CHAPTER - 12

AWT CONTROLS

12. Introduction

Controls are components that allow a user to interact with our application in various ways. A layout manager automatically positions components with in a container. The appearance of window is determined by a combination of the controls that it contains and the layout manger used to position them. The Layout managers manage manual positions of placing the components.

The AWT supports the following types of controls:

Labels

Push buttons

Check boxes

Choice lists

Lists

Scroll bars

Text editing

12.1 Adding Controls:

First created an instance of the desired control and then add it to a window by calling **add( )** method, which is defined by **Container.**

Syntax: Component add(Component obj)

obj is an instance of the control that we want to add. It returns the object of the control. Once, the control is added, it will automatically be visible whenever its parent window is displayed.

12.2 Removing Controls:

We want to remove a control from a window when the control is no longer needed, call the method remove( ). This method is also defined by Container.

Syntax: void remove(Component obj)

obj is a reference to the control that we want to remove.

To remove all the controls from the window used the following syntax as

Syntax: void removeAll( )

12.3 Labels:

Label is a text information that display on the GUI as a stirng. These 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 how)

The first form creates a blank label

The second form creates a label that contains the string specified by str. The string is left-justified

The third version crates 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:

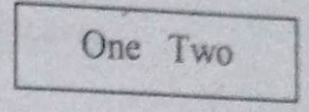
1. void setText(String str): To change the text specified by the str argument
2. String getText( ): It returns the text from the label
3. void setAlignment(int how): To set the position of the label specified by the how constant
4. int getAlignment( ): It returns the alignment

```
// Example program to create two Labels
import java.awt.*;
class Frame1 extends Frame
```



```
{
    Frame1()
    {
    setTitle("Demo");
    setSize(250,450);
    setVisible(true);
    setLayout(new FlowLayout());
    Label L1=new Label("One");
    Label L2=new Label("Two");
    add(L1);
    add(L2);
    }
}
class Labeldemo
{
    public static void main(String[] args)
    {
     Frame1 f=new Frame1();

    }
}
```



12.4 Buttons:

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. Button defines these two constructors:

Button( )

Button(String str)

The first form creates an empty button

The second form creates a button that contains str as a label

Methods:

1. void setLabel(String str): To set a label on the push button
2. String getLabel( ): It returns the label of the push button as String

Handling Buttons:

***ActionEvent* Class:** An ActionEvent is generated when a button is pressed, a list item is double-clicked, or menu item is selected.

Hierachy: java.lang.Object

↓

java.util.EventObject

↓

java.awt.AWTEvent

↓

java.awt.event.ActionEvent

Methods:

1. Object getSource(): It returns the object on which event initially occurred
2. String ***getActionCommand():*** It returns the command string associated with this action

ActionListener Interface: Each time a button is pressed, an action event is generated. This is sent to any listeners. Each listener implements the ActionListener interface. That interface defines the actionPerformed() metho, which called when an event occurs. An ActionEvent object is supplied as the argument to this method.

Method: void actionPerformed(ActionEvent obj)

// Example Program to perform arithmetic operations with textfields and command buttons

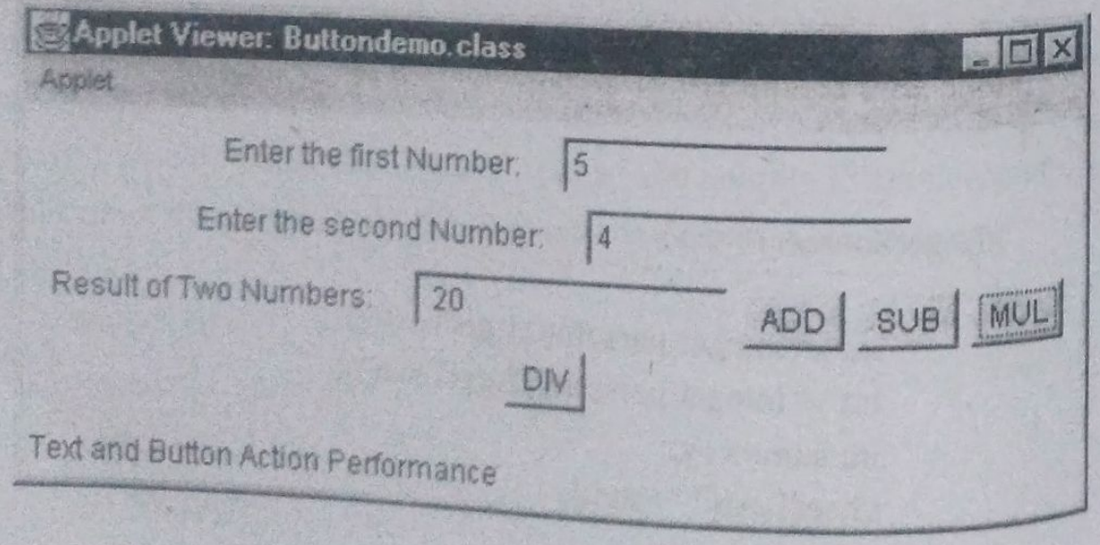
```
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;
/*<applet code="Buttondemo.class" width=400 height=350> </Applet> */
public class Buttondemo extends Applet implements ActionListener
{
    TextField t1,t2,t3;
    Button b1,b2,b3,b4;
    Label L1,L2,L3;
    String msg=" ";
    public void init()
```



```
{
 L1=new Label("Enter the first Number:");
 add(L1);
 t1=new TextField(15);
 add(t1);
 L2=new Label("Enter the second Number:");
 add(L2);
 t2=new TextField(15);
 add(t2);
 L3=new Label("Result of Two Numbers:");
 add(L3);
 t3=new TextField(15);
 add(t3);
 b1=new Button("ADD");
 add(b1);
 b1.addActionListener(this);
 b2=new Button("SUB");
 add(b2);
 b2.addActionListener(this);
 b3=new Button("MUL");
 add(b3);
 b3.addActionListener(this);
 b4=new Button("DIV");
 add(b4);
 b4.addActionListener(this);
}
public void actionPerformed(ActionEvent e)
{

 if(e.getSource()==b1)
 {
        int x=Integer.parseInt(t1.getText());
        int y=Integer.parseInt(t2.getText());
        int sum=x+y;
        t3.setText(" "+sum);
 }
```

```
if(e.getSource()==b2)
{
        int x=Integer.parseInt(t1.getText());
        int y=Integer.parseInt(t2.getText());
        int sub=x-y;
        t3.setText(String.valueOf(sub));
}
if(e.getSource()==b3)
{
        int x=Integer.parseInt(t1.getText());
        int y=Integer.parseInt(t2.getText());
        int m=x*y;
        t3.setText(" "+m);
}
if(e.getSource()==b4)
{
        int x=Integer.parseInt(t1.getText());
        int y=Integer.parseInt(t2.getText());
        int d=x/y;
        t3.setText(" "+d);
    }
    showStatus("Text and Button Action Performance");
    repaint();
    }
}
```



12.5 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 mark or not. There is a label associated with each check box that describes the option that box represents. We can change the state of the check box by clicking it. Check boxes can be used individually or as a part of a group. Check boxes are objects of the Checkbox class. Checkbox supports these constructors:

Checkbox( )

Checkbox(String str)

Checkbox(String str, boolean on)

Checkbox(String str, boolean on, CheckboxGroup cb)

The first form creates a check box with empty label and the state is unchecked The second form creates a check box with the label str and the state is unchecked The third form creates a check box with the label str and the state is checked (√) when the boolean argument on is true The fourth form creates a check box with the label str and the state is checked (√) when the boolean argument on is true whose group is specified by cb.

Methods:

1. boolean getState( ): Determines whether this check box is in the "on" or "off" state.
2. void setState(boolean on): Sets the state of this check box to the specified state
3. String getLabel( ): Gets the label of this check box
4. void setLabel( ): Sets this check box's label to be the string argument

// Example program to create different check boxes with individual group

```
import java.awt.*;
class Frame1 extends Frame
{
    Frame1()
    {
    setTitle("Demo");
    setSize(350,200);
    setVisible(true);
```

```
        setLayout(new FlowLayout());
        Checkbox c1=new Checkbox();
        Checkbox c2=new Checkbox("DOS");
        Checkbox c3=new Checkbox("Window",true);
        add(c1);
        add(c2);
        add(c3);
        }
}
class Cboxdemo
{
        public static void main(String[] args)
        {
          Frame1 f=new Frame1();
        }
}
```

Handling Checkboxes:

***ItemEvent* Class:** An ItemEvent is generated when a checkbox or a list item is clicked or when a checkable menu item is selected or deselected.

Hierarchy: java.lang.Object

↓

java.util.EventObject

↓

java.awt.AWTEvent

↓

java.awt.event.ItemEvent

Methods:

1. Object getItem(): returns the item effected by the event
2. int getStateChange(): returns the type of state change

***ItemListener Interface*:** Each time a Checkbox is selected or deselected, an item event is generated. This is sent to any listeners. Each listener implements the ItemListener interface. That interface defines the

itemStateChanged() method. An ItemEvent object is supplied as the argument to this method.

Method: void itemStateChanged (ActionEvent obj)

```
// Example program to create Checkboxes and print the status of the selection
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;
class Frame1 extends Frame implements ItemListener
{
        String msg=" ";
        Checkbox c1,c2,c3;
        Frame1()
        {
         setSize(400,350);
         setVisible(true);
         setLayout(new FlowLayout());
         c1=new Checkbox("Dos");
         c2=new Checkbox("Windows");
         c3=new Checkbox("Unix");
         add(c1);
         add(c2);
         add(c3);
         c1.addItemListener(this);
         c2.addItemListener(this);
         c3.addItemListener(this);
        }
public void itemStateChanged(ItemEvent e)
{
 repaint();
}
public void paint(Graphics g)
{
 msg="State:";
 g.drawString(msg,50,150);
 msg="DOS:"+c1.getState();
 g.drawString(msg,50,170);
```

```
        msg="WINDOWS:"+c2.getState();
        g.drawString(msg,50,190);
        msg="UNIX:"+c3.getState();
        g.drawString(msg,50,210);
      }
}
class Checkdemo
{
      public static void main(String[] args)
      {
        Frame1 obj=new Frame1();
      }
}
```

12.6 CheckboxGroup:

Collection of check boxes is placed under one group is considered as CheckboxGroup. These Checkboxes are often called radio buttons, since one choice is selected at any one time. To create a set of mutually exclusive check boxes, first define the group to which they will belong and then specify that group when we construct the check boxes. Check box groups are objects of type CheckboxGroup.

```
// Example program for Check boxes
import java.awt.*;
class Frame1 extends Frame
{
      Checkbox c1,c2,c3;
      Frame1()
      {
      setTitle("Demo");
      setSize(350,200);
      setVisible(true);
      setLayout(new FlowLayout());
      CheckboxGroup cbg=new CheckboxGroup();
      c1=new Checkbox("DOS",cbg,true);
      c2=new Checkbox("Windows",cbg,false);
```



```
      c3=new Checkbox("UNIX",cbg,false);
      add(c1);
      add(c2);
      add(c3);
      }
}
class Chgroup
{
      public static void main(String[] args)
      {
        Frame1 f=new Frame1();
        }
}
```

Handling CheckboxGroups:

```
// Example program to change the background color of the applet window by selecting different colors placed in checkbox group
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;
public class Checkboxdemo extends Applet implements ItemListener
{
      String msg=" ";
      Checkbox c1,c2,c3;
      CheckboxGroup cg;
      public void init()
      {
        cg=new CheckboxGroup();
        c1=new Checkbox("RED",cg,false);
        c2=new Checkbox("GREEN",cg,false);
        c3=new Checkbox("BLUE",cg,false);
        add(c1);
        add(c2);
```

```
    add(c3);
    c1.addItemListener(this);
    c2.addItemListener(this);
    c3.addItemListener(this);
  }
public void itemStateChanged(ItemEvent e)
{
  repaint();
}
public void paint(Graphics g)
{
  msg=cg.getSelectedCheckbox().getLabel();
  if(msg.equals("RED"))
        setBackground(Color.red);
  else if(msg.equals("GREEN"))
        setBackground(Color.green);
  else if(msg.equals("BLUE"))
        setBackground(Color.blue);
  showStatus("Selected Color:"+msg);
  }
}
/* <applet code="Checkboxdemo.class" width=400 height=350></Applet>*/
```

12.7 Choice Control:

The **Choice** class is used to create a pop-up list of items from which the user may choose. A Choice control is a form of menu. Choice only defines the default constructor, which defines an empty list.

Choice( )

Methods:

1. void add(String name): To add a selection of the list specified by the name argument
2. String getSelectedItem( ): Returns the string containing the name of the item

3. int getSelectedIndex( ): Returns the index of the selected item. The first item is at index 0.
4. int getItemCount( ): Returns the number of items in the list
5. void select(int index): To select the item in the choice depending upon the index value
6. void select(String name): To select the item in the choice depending upon the name argument
7. String getItem(int index): Returns the name of the item that contain the value of index

```
// Example program to create a Choice list
import java.awt.*;
class Frame1 extends Frame
{
      Frame1()
      {
      setTitle("Demo");
      setSize(350,200);
      setVisible(true);
      setLayout(new FlowLayout());
      Choice c=new Choice();
      add(c);
      c.add("Red");
      c.add("Green");
      c.add("Blue");
      }
}
class Choicedemo
{
      public static void main(String[] args)
      {
        Frame1 f=new Frame1();
      }
}
```

Handling ChoiceLists:

```
// Example program to change the background color of the applet window by selecting different colors from choice control
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;
/*<applet code="Choicedemo.class" width=400 height=350></Applet> */
public class Choicedemo extends Applet implements ItemListener
{
    String msg=" ";
    Choice ch;
    public void init()
    {
     ch=new Choice();
     ch.add("RED");
     ch.add("GREEN");
     ch.add("BLUE");
     add(ch);
     ch.addItemListener(this);
    }
    public void itemStateChanged(ItemEvent e)
    {
     repaint();
    }
    public void paint(Graphics g)
    {
     msg=ch.getSelectedItem();
     if(msg.equals("RED"))
            setBackground(Color.red);
     else if(msg.equals("GREEN"))
            setBackground(Color.green);
     else if(msg.equals("BLUE"))
            setBackground(Color.blue);
     g.drawString("Selection:"+msg,50,200);
     showStatus("List Action Performance");
    }
}
```


12.8 Lists:

The List class provides a compact, multiple-choice, scrolling section list. The 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.

List()
List(int nrows)
List(int nrows, boolean select)

The first form creates a List control that allows one item to be selected at any one time.

In the second form, the value of nrows specifies the number of entries in the list that will always visible.

In the third form, if select is true, then the user may select two or more items at a time. If its is false, then only one item may be selected.

To add a selection to the list, call add() method

Syntax: void add(String name)
Void add(String name,int in)

Here, name is the item that is added to the list. It adds the items at the end of the list.

The second form adds the item at the index specified by the in argument. Indexing starts at zero. Specify –1 to add the item to the end of the list.

Methods:

1. String getSelectedItem(): returns the currently selected item as string. If more than one item is selected, it returns null
2. int getSelectedIndex(): returns the index value of the currently selected item. If more than one item is selected, it returns -1
3. String[] getSelectedItems(): returns the multiple selection of items as string array
4. int[] getSelectedIndexes(): returns an array containing the indexes of the currently selected items
5. int getItemCount(): returns the number of items in the list
6. void select(int in): set the item as selection specified by the in index argument
7. String getItem(int in): returns the name associated with item by calling its index with in argument

Handling Lists:

```
// Example program to select different items by creating a list box
import java.awt.*;
import java.applet.Applet;
```

```
import java.awt.event.*;
/*<applet code="Listdemo.class" width=400 height=350> </Applet> */
public class Listdemo extends Applet implements ActionListener
{
    String msg=" ";
    List ch,browser;
    public void init()
    {
     ch=new List(3,true);
     browser=new List(4);
     ch.add("DOS");
     ch.add("WINDOWS");
     ch.add("UNIX");
     browser.add("Explorer");
     browser.add("Firefox");
     browser.add("Opera");
     browser.select(1);
     add(ch);
     add(browser);
     ch.addActionListener(this);
     browser.addActionListener(this);
    }
public void actionPerformed(ActionEvent e)
    {
     repaint();
    }
    public void paint(Graphics g)
    {
     msg="Selected OS:";
     int id[];
     id=ch.getSelectedIndexes();
     for(int i=0;i<id.length;i++)
     msg=msg+ch.getItem(id[i])+" ";
     g.drawString(msg,50,100);
     msg="Current:";
     msg+=browser.getSelectedItem();
     g.drawString(msg,50,150);
     showStatus("List Action Performance");
    }
}
```


12.9 Scroll Bars:

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 for the scroll bar. The user can drag the slider box to a new position. The scroll bar will then reflect this value.

Scrollbar defines the following constructors:

Scrollbar()

Scrollbar(int style)

Scrollbar(int style, int ivalue,int ssize, int min,int max)

The first form creates a vertical scroll bar

The second form and third form allow to specify the style of the scrollbar. If style is Scrollbar.VERTICAL, a vertical scrollbar is created. If style is Scrollbar.HORIZONTAL, the scrollbar is horizontal

In the third from the initial value is specified by ivalue, the number of units are specified by ssize and minimum, maximum values for the scrollbar are specified by min, max arguments.

Methods:

1. void setValues(int ivalue, int ssize, int min, int max): Used to set the parameters
2. int getValue(): returns the current setting value of the scrollbar
3. void setValue(int newvalue): Set with the newvalue
4. int getMinimum(): returns the minimum value of the scrollbar
5. int getMaximum(): return the maximum value of the scrollbar

TextField: The **TextField** class implements a single-line text entry area, usually called an edit control. Text fields are used to enter stings to edit the information. TextField is a subclass of TextComponent. TextField defines the following constructors:

TextField()

TextField(int num)

TextField(String str)

TextField(String str, int num)

The first form creates a default text field.

The second form creates a text field that is num characters wide

The third form creates a text field with the string specified by str
The fourth form initializes a text field and sets its width

Methods:

1. Sting getText(): returns the information currently contained in the text field as string
2. void setText(String str): sets the string information into the text field
3. String getSelectedText(): returns the currently selected text
4. void selected(int sindex, int eindex): selects the characters beginning at sindex and ending at eindex-1
5. boolean isEditable(): return true if the text may be changed and false if not
6. void setEditable(boolean b): if b is true, the text may be changed, otherwise, the text cannot be altered
7. void getEchoChar(char ch): it disable the echoing characters on the screen and display the characters as specified by the ch argument
8. boolean echoCharIsSet(): returns true, if echochar is set otherwise, returns false
9. char getEchoChar(): returns the echo character

Handling Text Fields:

```
//Example program to create an Applet to enter the data in Textfields and display sum of the two numbers
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;
/*<applet code="Tdemo.class" width=400 height=350> </Applet> */
public class Tdemo extends Applet implements ActionListener
{
      TextField t1,t2,t3;
      Label L1,L2,L3;
      public void init()
      {
       L1=new Label("Enter the first Number:");
       add(L1);
       t1=new TextField(15);
       add(t1);
       L2=new Label("Enter the second Number:");
       add(L2);
       t2=new TextField(15);
       add(t2);
```



```
       t1.addActionListener(this);
       t2.addActionListener(this);
      }
      public void actionPerformed(ActionEvent e)
      {
       repaint();
      }

public void paint(Graphics g)
      {
       int x=Integer.parseInt(t1.getText());
       int y=Integer.parseInt(t2.getText());
       int sum=x+y;
       g.drawString("Total="+sum,100,100);
      }
}
```

12.10 TextArea:

TextField is useful to enter only one line of information. If we want to enter more than one line of data it is not possible with the TextField. To handle these situaltions, the AWT includes a simple multiline editor called TextArea. Constructors of the TextArea are:

TextArea()

TextArea(int nlines, int nchars)

TextArea(String str)

TextArea(String str, int nlines, int nchars)

TextArea(String str, int nlines, int nchars, int sbars)

nlines specifies the height in lines, nchars specifies the width in charactes

str specifies the string filled with TextArea

sbars specifies the scroll bars that are associated with the TextArea. Sbars must have one of these values:

SCROLLBARS_BOTH

SCROLLBARS_NONE

SCROLLBARS_HORNIZONTAL_ONLY

SCROLLBARS_VERTICAL_ONLY

It supports the **getText(), setText(), getSelectedText(), select(), isEditable() and stEditable()** methods. In addition it also supports the following methods

1. void append(String str): it appends the string specified by str to the end of current text
2. void insert(String str, int index): inserts the string specified by str at the specified index
3. void replaceRange(String str, int sindex, int eindex): replaces the characters with str from sindex to eindex-1

Example Program:

```
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;
/*<applet code="TextAreademo.class" width=400 height=350> </Applet> */

public class TextAreademo extends Applet
{
    public void init()
    {
      String str="Java Language\n"+

      TextArea t=new TextArea(str,10,15);
      add(t);
    }
}
```

CHAPTER - 13

LAYOUT MANAGER

13. Layout Manager

A layout manageris an instance of any class that implements the **LayoutManager** interface. 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 layoutObj)

Each layout manager keeps track of a list of components that are stored by their names. The layout manager is notified each time you add a component to a container. Whenever the container needs to be resized, the layout manager is consulted via its **minimumLayoutSize()** and **preferredLayoutSize( )** methods. Each component that is being managed by a layout manager contains the **getPreferredSize( )** and **getMinimumSize( )** methods.

LAYOUTS are following types

1) FlowLayout 2) BorderLayout 3) GridLayout 4)Card Layout

13.1 FlowLayout

the default layout manager is **FlowLayout.**

FlowLayout implements a simple layout style, which is similar to how words flow in a text editor. Components are laid out from the upper-left corner, left to right and top to bottom. When no more components fit on a line, the next one appears on the next line. A small space is left between each component, above and below, as well as left and right.

The constructors for **FlowLayout**:

FlowLayout( )

FlowLayout(int *how*)

FlowLayout(int *how*, int *horz*, int *vert*)

Alignment as follows

FlowLayout.LEFT

FlowLayout.CENTER

FlowLayout.RIGHT

These values specify left, center, and right alignment, respectively.

Example on FLOWLAYOUT

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="FlowLayoutDemo" width=250 height=200>
</applet>
*/
public class FlowLayoutDemo extends Applet
implements ItemListener {
String msg = "";
Checkbox IT, CSE,ECE,CSSE;
public void init() {
// set left-aligned flow layout
setLayout(new FlowLayout(FlowLayout.LEFT));
IT = new Checkbox("IT", null, true);
CSE = new Checkbox("CSE");
ECE = new Checkbox("ECE");
CSSE = new Checkbox("CSSE");
add(IT);
add(CSE);
add(ECE);
add(CSSE);

IT.addItemListener(this);
CSE.addItemListener(this);
ECE.addItemListener(this);
CSSE.addItemListener(this);
}
public void itemStateChanged(ItemEvent ie) {
repaint();
}
public void paint(Graphics g) {
msg = "Current state: ";
g.drawString(msg, 6, 80);
msg = " IT: " + IT.getState();
g.drawString(msg, 6, 100);
msg = " CSE " + CSE.getState();
g.drawString(msg, 6, 120);
msg = "ECE " + ECE.getState();
```

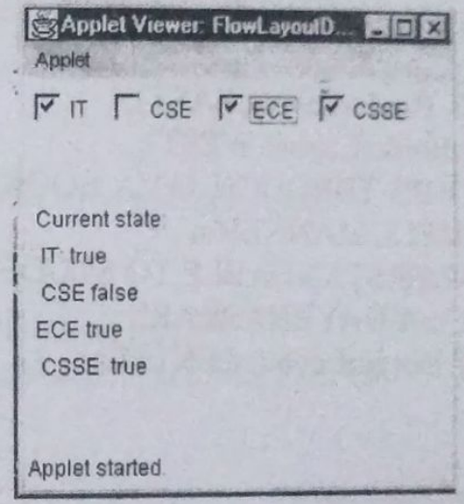


```
g.drawString(msg, 6, 140);
msg = " CSSE: " + CSSE.getState();
g.drawString(msg, 6, 160);
}
}
```

OUTPUT

C:VIJAY>javac FlowLayoutDemo .java

C:\vijay>appletviewer FlowLayoutDemo .java



13.2 BorderLayout

The **BorderLayout** class implements a common layout style for top-level windows. It has four narrow, fixed-width components at the edges and one large area in the center. The four sides are referred to as north, south, east, and west. The middle area is called the center. Here are the constructors defined by **BorderLayout:**

BorderLayout( )

BorderLayout(int horz, int vert)

The first form 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

When adding components, you will use these constants with the following form of **add( )**, which is defined by **Container**: void add(Component *compObj*, Object *region*);

Here, *compObj* is the component to be added, and *region* specifies where the component will be added.

Here is an example of a **BorderLayout** with a component in each layout area:

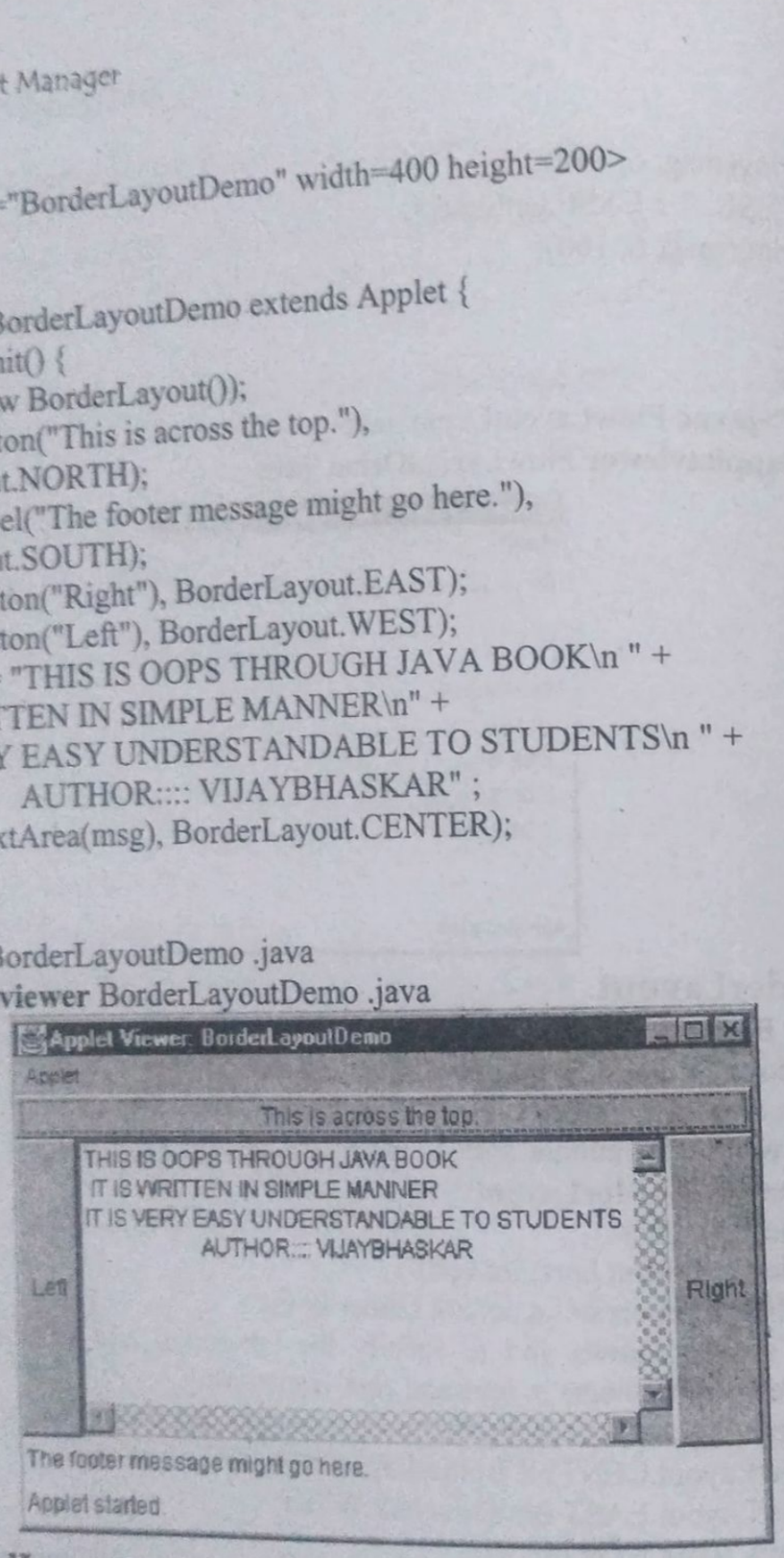
```
import java.awt.*;
import java.applet.*;
import java.util.*;
```

```
/*
<applet code="BorderLayoutDemo" width=400 height=200>
</applet>
*/
public class BorderLayoutDemo extends Applet {
public void init() {
setLayout(new BorderLayout());
add(new Button("This is across the top."),
BorderLayout.NORTH);
add(new Label("The footer message might go here."),
BorderLayout.SOUTH);
add(new Button("Right"), BorderLayout.EAST);
add(new Button("Left"), BorderLayout.WEST);
String msg = "THIS IS OOPS THROUGH JAVA BOOK\n " +
"IT IS WRITTEN IN SIMPLE MANNER\n" +
"IT IS VERY EASY UNDERSTANDABLE TO STUDENTS\n " +
"                AUTHOR:::: VIJAYBHASKAR" ;
add(new TextArea(msg), BorderLayout.CENTER);
}}
```

OUTPUT

C: >javac BorderLayoutDemo .java

C:\ >appletviewer BorderLayoutDemo .java



13.3 GridLayout

GridLayout lays out components in a two-dimensional grid. When you instantiate a **GridLayout**, you define the number 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*)

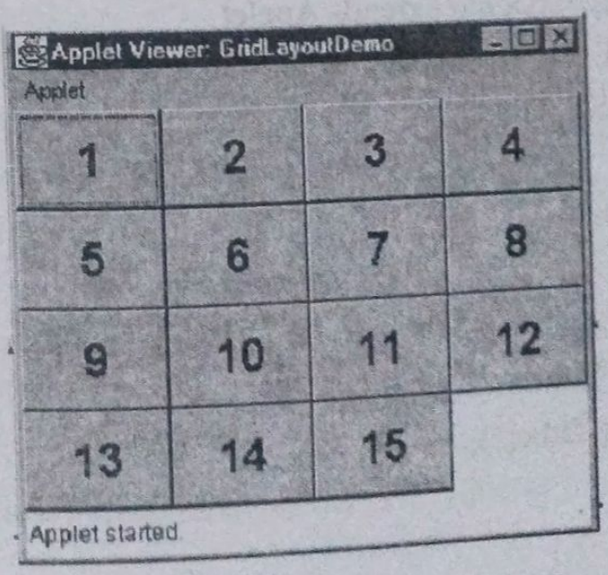
The first form creates a single-column grid layout. The second form creates a grid layout with the specified number of rows and columns. The third form allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively. This is a sample program that creates a 4×4 grid and fills it in with 15 buttons, each labeled with its index:

```
// Demonstrate GridLayout
import java.awt.*;
import java.applet.*;
/*
<applet code="GridLayoutDemo" width=300 height=200>
</applet>
*/
public class GridLayoutDemo extends Applet {
static final int n = 4;
public void init() {
setLayout(new GridLayout(n, n));
setFont(new Font("SansSerif", Font.BOLD, 24));
for(int i = 0; i < n; i++) {
for(int j = 0; j < n; j++) {
int k = i * n + j;
if(k > 0)
add(new Button("" + k));
}
}
}
}
```

OUTPUT

C: >javac GridLayoutDemo .java

C:\ >appletviewer GridLayoutDemo .java



13.4 CardLayout

The ***CardLayout*** class is stores several different layouts. Each layout can be thought of as being on a separate index card in a deck that can be shuffled so that any card is on top at a given time.

CardLayout provides these two constructors:

CardLayout( )

CardLayout(int *horz*, int *vert*)

When card panels are added to a panel, they are usually given a name. Thus, most of the time, you will use this form of **add( )** when adding cards to a panel: void add(Component *panelObj*, Object *name*); Here, *name* is a string that specifies the name of the card whose panel is specified by *panelObj*. 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*)

```
// Demonstrate CardLayout.
// Demonstrate CardLayout.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="CardLayoutDemo" width=300 height=100>
</applet>
*/
public class CardLayoutDemo extends Applet
implements ActionListener, MouseListener {
Checkbox CSE ,CSSE, IT,ECE;
Panel osCards;
CardLayout cardLO;
Button Win, Other;
public void init() {
Win = new Button("Windows");
Other = new Button("Other");
add(Win);
add(Other);
```



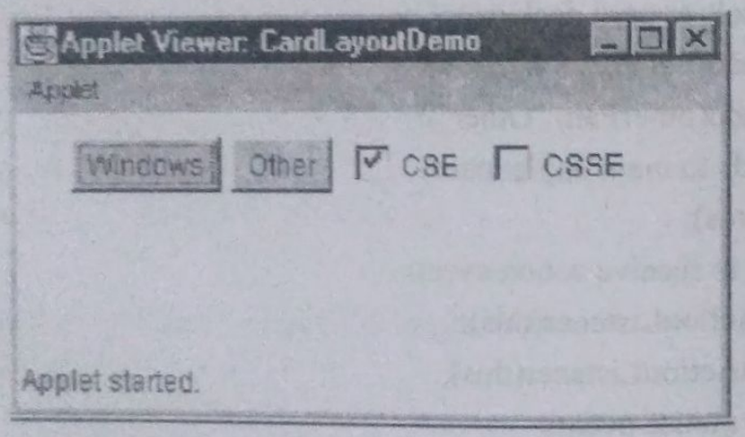
```
cardLO = new CardLayout();
osCards = new Panel();
osCards.setLayout(cardLO); // set panel layout to card layout
CSE = new Checkbox("CSE", null, true);
CSSE = new Checkbox("CSSE");
IT = new Checkbox("IT");
ECE = new Checkbox("ECE");
// add Windows check boxes to a panel
Panel winPan = new Panel();
winPan.add(CSE);
winPan.add(CSSE);
// Add other OS check boxes to a panel
Panel otherPan = new Panel();
otherPan.add(IT);
otherPan.add(ECE);
// add panels to card deck panel
osCards.add(winPan, "Windows");
osCards.add(otherPan, "Other");
// add cards to main applet panel
add(osCards);
// register to receive action events
Win.addActionListener(this);
Other.addActionListener(this);
// register mouse events
addMouseListener(this);
}
// Cycle through panels.
public void mousePressed(MouseEvent me) {
cardLO.next(osCards);
}
// Provide empty implementations for the other MouseListener methods.
public void mouseClicked(MouseEvent me) {
}
public void mouseEntered(MouseEvent me) {
}
```

```
public void mouseExited(MouseEvent me) {
}
public void mouseReleased(MouseEvent me) {
}
public void actionPerformed(ActionEvent ae) {
if(ae.getSource() == Win) {
cardLO.show(osCards, "Windows");
}
else {
cardLO.show(osCards, "Other");
}
}}
```

OUTPUT

C:VIJAY>javac CardLayoutDemo .java

C:\vijay>appletviewer CardLayoutDemo .java



CHAPTER - 14

EVENT HANDLING

14. Introduction

Events: An event is an object that describes a state change in a source. Even driven is a consequence interaction of the user with the GUI. Some of the common interactions are moving the mouse, clicking the mouse, clicking a button, typing in a textfield etc.,

Event Sources: A source is an object that generates an event. This occurs when the internal state of that object changes in some way. Sources may generate more than one type of event.

14.1 Delegation Event Model:

In java1.0 the action() method is used to perform any actions that are provided with the controls. But the action() method is now deprecated. The new technique in java1.1 and java2 is the Event Delegation Model.

This method defines a consistent mechanism to generate and process the events. The process is: **a source generates an event and sends it to one or more listeners.** In this scheme, the listener simply waits until it receives an event. Once an event is received, the listener processes the event and then returns.

14.2 Event Listeners:

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

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

The methods that receive and process events are defined in a set of interfaces found in java.awt.event package.

Commonly used Event Listener Interfaces are

ActionListener, AdjustmentListener, ComponentListener, ContainerListener, FocusListener, ItemListener, KeyListener, MouseListener, MouseMotionListener, MouseWheelListener, TextListener, WindowFocusListener, WindowListener.

14.3 Event Classes:

Event classes are used to handle Java event handling mechanism. At the root of the java event class hierarchy is **EventObject** , which is in java.util package. It is the superclass for all events. It provides a constructor as

EventObject(Object x)

Methods provided by the class are

1. Object getSource(): returns the source of the event
2. Stirng toString() : returns the string equivalent of the event

The class AWTEvent, defined with in the java.awt package, is subclass of EventObject. It is the superclass of all AWT-based events used by the delegation model.

Commonly used Event classes are

ActionEvent, AdjustmentEvent, ComponentEvent, ContainerEvent, FocusEvent, InputEvent, ItemEvent, KeyEvent, MouseEvent, MouseWheelEvent, TextEvent, WindowEvent.

14.4 MouseEvent Class:

The MouseEvent class defines eight types of mouse events. It defines the following integer constants that can be used to identify them:

MOUSE_CLICKED	:	The user clicked the mouse
MOUSE_DRAGGED	:	The user dragged the mouse
MOUSE_ENTERED	:	The mouse entered a component
MOUSE_EXITED	:	The mouse exited from a component
MOUSE_MOVED	:	The mouse moved
MOUSE_PRESSED	:	The mouse was pressed
MOUSE_RELEASED	:	The mouse was released
MOUSE_WHEEL	:	The mouse wheel was moved

14.4.1 MouseEvent is a subclass of InputEvent

Hierarchy: java.lang.Object

↓

java.util.EventObject

↓

java.awt.AWTEvent

↓

java.awt.event

↓

java.awt.event.InputEvent

↓

java.awt.event.MouseEvent

Methods:

1. int getX(): returns the X coordinate of the mouse when an event occurred
2. int getY():returns the Y coordinate of the mouse when an event occurred
3. Point getPoint(): returns the coordinates of the mouse

14.4.2 MouseListener Interface:

This interface defines five methods.

1. void mouseClicked(MouseEvent me): it invokes if the mouse is pressed and released at the same point
2. void mouseEntered(MouseEvent me): it invokes when the mouse enters a component
3. void mouseExited(MouseEvent me): it invokes when the mouse leaves the component
4. void mousePressed(MouseEvent me): it invokes when the mouse is pressed
5. void mouseReleased(MouseEvent me): it invokes when the mouse is released

14.4.3 MouseMotionListener Interface:

This interface defines two methods

1. void mouseDragged(MouseEvent me): it invokes when multiple times as the mouse is dragged
2. void mouseMoves(MouseEvent me): it invokes when multiple times as the mouse is moved

14.4.4 Handling MouseEvents:

// Example program to draw rectangles with mouse clicking

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

```
import java.awt.event.*;
import java.applet.*;
/* <applet code="Lab15.class" width=400 height=350>
            </Applet> */
public class Lab15 extends Applet implements
MouseListener,MouseMotionListener

{
      String msg=" ";
      int x=0,y=0,w=0,h=0;
      public void init()
      {
            addMouseListener(this);
            addMouseMotionListener(this);
      }

public void mouseClicked(MouseEvent me)
      {
            x=me.getX();
            y=me.getY();
            repaint();
}
public void mouseEntered(MouseEvent me)
{
      x=me.getX();
      y=me.getY();
      repaint();
}
public void mouseExited(MouseEvent me)
{
      x=me.getX();
      y=me.getY();
      repaint();
}
public void mousePressed(MouseEvent me)
{
      x=me.getX();
      y=me.getY();
      repaint();
```



```
}
public void mouseReleased(MouseEvent me)
{
      w=me.getX();
      h=me.getY();
      repaint();
}
public void mouseDragged(MouseEvent me)
{
      w=me.getX();
      h=me.getY();
      repaint();
}
public void mouseMoved(MouseEvent me)
{
      showStatus("Mouse Operation");
}
public void paint(Graphics g)
{
      g.setColor(Color.red);
      g.fillRect(x,y,w,h);
}
}
```

14.5 KeyEvent Class:

A KeyEvent is generated when keyboard input occurs. There are three types of key events, which are identified by these integer constants:

KEY_PRESSED
KEY_RELEASED
KEY_TYPED

The first two events are generated when any key is pressed or released. The last event occurs when a character is generated.

Methods:

1. char getKeyChar(): returns the character that was entered
2. int getKeyCode(): returns the character code

14.5.1 KeyListener Interface:

The interface defines three methods.

1. void keyPressed(KeyEvent me): it invokes when a key is pressed
2. void keyReleased(KeyEvent me): it invokes when a key is released
3. void keyTyped(KeyEvent me): it invokes when a key is typed

14.5.2 Handling KeyEvents:

```
//Example program:
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/* <applet code="Keydemo.class" width=400 height=350>
        </Applet> */
public class Keydemo extends Applet implements KeyListener
{
    String msg=" ";
    int x=50,y=50;
    public void init()
    {
        addKeyListener(this);
        requestFocus();
    }
    public void keyPressed(KeyEvent me)
    {
        showStatus("Key Down");
    }
    public void keyReleased(KeyEvent me)
    {
        showStatus("Key Released");
    }
    public void keyTyped(KeyEvent me)
    {
        showStatus("Key Typed");
        msg=msg+me.getKeyChar();
        repaint();
    }
    public void paint(Graphics g)
    {
        g.setColor(Color.red);
        g.drawString(msg,x,y);
    }
}
```


14.6 Adapter Classes

An adapter class provides an empty implementation of all methods in an event listener interface. Adapter classes are useful when we want to receive and process only some of the events that are handled by a particular event listener interface. For this, we can define a new class to act as an event listener by extending one of the adapter classes and implementing only those events in which you are interested. Consider, the **MouseMotionAdapter** class has two methods, **mouseDragged() and mouseMoved()**. If we are interested in only mouse drag events, then we extend **mouseMotionAdapter** and implement **mouseDragged().** The empty implementation of **mouseMoved()** would handle the mouse motion events.

Adapter classes are provided by **java.awt.event** package. Some of the Adapter classes are

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

// Example program

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/* <applet code="Adapterdemo.class" width=400 height=350>
</Applet> */

public class Adapterdemo extends Applet
{
    public void init()
    {
        addMouseListener(new my(this));
        addMouseMotionListener(new myadd(this));
    }
```

```
}
class my extends MouseAdapter
{
        Adapterdemo a;
        public my(Adapterdemo p)
        {
                this.a=p;
        }
        public void mouseClicked(MouseEvent me)
        {
                a.showStatus("Mouse Clicked");
        }
}
class myadd extends MouseMotionAdapter
{
        Adapterdemo a;
        public myadd(Adapterdemo p)
        {
                this.a=p;
        }
        public void mouseDragged(MouseEvent me)
        {
                a.showStatus("Mouse Dragged");
        }
}
```