UNIT-2
USING AWT CONTROLS, LAYOUT MANAGERS AND MENUS

3.1 Control Fundamentals

Controls are components that allow a user to interact with your application in various ways for example, a commonly used control is the push button. A layout manager automatically positions components within a container. Thus, the appearance of a window is determined by a combination of the controls that it contains and the layout manager used to position them. The AWT supports the following types of controls:

- Labels
- Push buttons
- Check boxes
- Choice lists
- Lists
- Scroll bars
- Text editing

These controls are subclasses of Component.

Adding and Removing Controls: To include a control in a window, you must add it to the window. To do this, you must first create an instance of the desired control and then add it to a window by calling add( ), which is defined by Container. The add( ) method has several forms. The following form is the one that is used for the first part of this chapter:

Component add(Component compObj)
Here, compObj is an instance of the control that you want to add. A reference to compObj is returned. Once a control has been added, it will automatically be visible whenever its parent window is displayed.

Sometimes you will want to remove a control from a window when the control is no longer needed. To do this, call remove( ). This method is also defined by Container. It has this general form:

void remove(Component obj)
Here, obj is a reference to the control you want to remove. We can remove all controls by calling **removeAll( ).**

Responding to Controls: Except for labels, which are passive, all controls generate events when they are accessed by the user. For example, when the user clicks on a push button, an event is sent that identifies the push button. In general, a program simply implements the appropriate interface and then registers an event listener for each control that we need to monitor.

The HeadlessException: Mos of the AWT controls have constructors that can throw a HeadlessException when an attempt is made to instantiate a GUI component in a non-interactive environment (such as one in which no display, mouse, or keyboard is present).

3.1.1 Labels

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

Label( ) throws HeadlessException
Label(String str) throws HeadlessException
Label(String str, int how) throws HeadlessException

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**. You can set or change the text in a label by using the setText( ) method. You can obtain the current label by calling getText( ). These methods are shown here:

void setText(String str)
String getText( )

For setText( ), str specifies the new label. For getText( ), the current label is returned. You can set the alignment of the string within the label by calling setAlignment( ). To obtain the current alignment, call getAlignment( ). The methods are as follows:

void setAlignment(int how)
int getAlignment( )

Here, *how* must be one of the alignment constants. The following example creates three labels and adds them to an applet window.

```
// Demonstrate Labels
import java.awt.*;
import java.applet.*;
/*
<applet code="LabelDemo" width=300 height=200>
</applet>
*/
public class LabelDemo extends Applet {
      public void init( ) {
            Label one = new Label("One");
            Label two = new Label("Two");
            Label three = new Label("Three");
            // add labels to applet window
            add(one);
            add(two);
            add(three);
        }
 }
```

The labels are organized in the window by the default layout manager.

3.1.2 Buttons

Perhaps the most widely used control is the push button. 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( ) throws HeadlessException

Button(String str) throws HeadlessException

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( ). You can retrieve its label by calling getLabel( ). These methods are as follows:

void setLabel(String str)

String getLabel( )

Here, str becomes the new label for the button.

Handling Buttons: Each time a button is pressed, an action event is generated. This is sent to any listeners that previously registered an interest in receiving action event notifications from that component. Each listener implements the ActionListener interface. That interface defines the actionPerformed( ) method, which is called when an event occurs. An ActionEvent object is supplied as the argument to this method. It contains both a reference to the button that generated the event and a reference to the action command string associated with the button. By default, the action command string is the label of the button. Usually, either the button reference or the action command string can be used to identify the button.

Here is an example that creates three buttons labeled ―Yes‖, ―No‖, and ―Undecided‖. Each time one is pressed, a message is displayed that reports which button has been pressed. In this version, the action command of the button (which, by default, is its label) is used to determine which button has been pressed. The label is obtained by calling the getActionCommand( ) method on the ActionEvent object passed to **actionPerformed( ).**

// Demonstrate Buttons

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="ButtonDemo" width=250 height=150>
</applet>
*/
public class ButtonDemo extends Applet implements ActionListener {
      String msg = "";
      Button yes, no, maybe;
      public void init() {
      yes = new Button("Yes"
      no = new Button("No");
      maybe = new Button("Undecided");
      add(yes);
      add(no);
      add(maybe);
      yes.addActionListener(this);
      no.addActionListener(this);
      maybe.addActionListener(this);
   }
```

```
    public void actionPerformed(ActionEvent ae) {
        String str = ae.getActionCommand();
        if(str.equals("Yes")) {
           msg = "You pressed Yes.";
        }
       else if(str.equals("No")) {
           msg = "You pressed No.";
       }
       else {
              msg = "You pressed Undecided.";
           }
      repaint( );
     }
     public void paint(Graphics g) {
      g.drawString(msg, 6, 100);
     }
}
```

3.1.3 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. Check boxes are objects of the Checkbox class. Checkbox supports these constructors:

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

The first form creates a check box whose label is initially blank. The state of the check box is unchecked. The second form creates a check box whose label is specified by str. The state of the check box is unchecked. The third form allows you to set the initial state of the check box. If on is true, the check box is initially checked; otherwise, it is cleared. The fourth and fifth forms create a check box whose label is specified by str and whose group is specified by cbGroup. If this check box is not part of a group, then cbGroup must be null. The value of on determines the initial state of the check box.

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( ). These methods are as follows:

boolean getState( )
void setState(boolean on)
String getLabel( )
void setLabel(String str)

Here, if on is true, the box is checked. If it is false, the box is cleared. The string passed in str becomes the new label associated with the invoking check box.

Handling Checkboxes: Each time a check box is selected or deselected, an item event is generated. This is sent to any listeners that previously registered an interest in receiving item event notifications from that component. Each listener implements the ItemListener interface. That interface defines the **itemStateChanged( )** method. An ItemEvent object is supplied as the argument to this method. It contains information about the event (for example, whether it was a selection or deselection)

The following program creates four check boxes. The initial state of the first box is checked. The status of each check box is displayed. Each time you change the state of a check box, the status display is updated.

```
// Demonstrate check boxes.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="CheckboxDemo" width=250 height=200>
</applet>
*/
public class CheckboxDemo extends Applet implements ItemListener {
       String msg = "";
       Checkbox winXP, winVista, solaris, mac;
       public void init() {
       winXP = new Checkbox("Windows XP", null, true);
       winVista = new Checkbox("Windows Vista");
       solaris = new Checkbox("Solaris");
       mac = new Checkbox("Mac OS");
       add(winXP);
       add(winVista);
       add(solaris);
       add(mac);
       winXP.addItemListener(this);
       winVista.addItemListener(this);
       solaris.addItemListener(this);
       mac.addItemListener(this);
      }
 public void itemStateChanged(ItemEvent ie) {
    repaint();
}
   // Display current state of the check boxes.
   public void paint(Graphics g) {
      msg = "Current state: ";
      g.drawString(msg, 6, 80);
      msg = " Windows XP: " + winXP.getState();
      g.drawString(msg, 6, 100);
      msg = " Windows Vista: " + winVista.getState();
      g.drawString(msg, 6, 120);
      msg = " Solaris: " + solaris.getState();
      g.drawString(msg, 6, 140);
      msg = " Mac OS: " + mac.getState();
      g.drawString(msg, 6, 160);
```

```
  }
}
```

3.1.4 Checkbox Group

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. Check box groups are objects of type CheckboxGroup. Only the default constructor is defined, which creates an empty group. You can determine which check box in a group is currently selected by calling getSelectedCheckbox( ). You can set a check box by calling setSelectedCheckbox( ). These methods are as follows:

Checkbox getSelectedCheckbox( )
void setSelectedCheckbox(Checkbox which)

Here, which is the check box that you want to be selected. The previously selected check box will be turned off. Here is a program that uses check boxes that are part of a group:

```
// Demonstrate check box group.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="CBGroup" width=250 height=200>
</applet>
*/
   public class CBGroup extends Applet implements ItemListener {
     String msg = "";
     Checkbox winXP, winVista, solaris, mac;
     CheckboxGroup cbg;
     public void init() {
          cbg = new CheckboxGroup();
     winXP = new Checkbox("Windows XP", cbg, true);
     winVista = new Checkbox("Windows Vista", cbg, false);
     solaris = new Checkbox("Solaris", cbg, false);
     mac = new Checkbox("Mac OS", cbg, false);
     add(winXP);
     add(winVista);
     add(solaris);
     add(mac);
     winXP.addItemListener(this);
     winVista.addItemListener(this);
     solaris.addItemListener(this);
     mac.addItemListener(this);
   }
    public void itemStateChanged(ItemEvent ie) {
       repaint();
    }
   // Display current state of the check boxes.
         public void paint(Graphics g) {
```

```
        msg = "Current selection: ";
        msg += cbg.getSelectedCheckbox().getLabel();
        g.drawString(msg, 6, 100);
    }
  }
```

3.1.5 Choice Controls

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. To add a selection to the list, call add( ). It has this general form:

```
void add(String name)
```

Here, name is the name of the item being added. Items are added to the list in the order in which calls to add( ) occur. To determine which item is currently selected, you may call either **getSelectedItem( )** or **getSelectedIndex( ).** These methods are shown here:

String getSelectedItem( )
int getSelectedIndex( )

The getSelectedItem( ) method returns a string containing the name of the item. getSelectedIndex( ) returns the index of the item. The first item is at index 0. By default, the first item added to the list is selected.

To obtain the number of items in the list, call getItemCount( ). You can set the currently selected item using the select( ) method with either a zero-based integer index or a string that will match a name in the list. These methods are shown here:

int getItemCount( )
void select(int index)
void select(String name)

Given an index, you can obtain the name associated with the item at that index by calling getItem( ), which has this general form:

```
String getItem(int index)
```

Each time a choice is selected, an item event is generated. This is sent to any listeners that previously registered an interest in receiving item event notifications from that component. Each listener implements the ItemListener interface. That interface defines the **itemStateChanged( )** method. An ItemEvent object is supplied as the argument to this method. Here is an example that creates two Choice menus. One selects the operating system.

The other selects the browser.

```
// Demonstrate Choice lists.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="ChoiceDemo" width=300 height=180>
```

```
</applet>
*/
public class ChoiceDemo extends Applet implements ItemListener {
      Choice os, browser;
      String msg = " ";
      public void init() {
      os = new Choice( );
      browser = new Choice();
      // add items to os list
      os.add("Windows XP");
      os.add("Windows Vista");
      os.add("Solaris");
      os.add("Mac OS");
      // add items to browser list
      browser.add("Internet Explorer");
      browser.add("Firefox");
      browser.add("Opera");
      // add choice lists to window
      add(os);
      add(browser);
     // register to receive item events
     os.addItemListener(this);
     browser.addItemListener(this);
  }
   public void itemStateChanged(ItemEvent ie) {
      repaint();
   }
   // Display current selections.
    public void paint(Graphics g) {
         msg = "Current OS: ";
         msg += os.getSelectedItem();
         g.drawString(msg, 6, 120);
         msg = "Current Browser: ";
         msg += browser.getSelectedItem();
         g.drawString(msg, 6, 140);
    }
}
```

3.1.6 Lists

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:

List( ) throws HeadlessException
List(int numRows) throws HeadlessException
List(int numRows, boolean multipleSelect) throws HeadlessException
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

multipleSelect 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. To add a selection to the list, call add( ). It has the following two forms:

void add(String name)
void add(String name, int index)

Here, name is the name of the item added to the list. The first form adds items to the end of the list. The second form adds the item at the index specified by index. Indexing begins at zero. You can specify –1 to add the item to the end of the list. For lists that allow only single selection, you can determine which item is currently selected by calling either getSelectedItem( ) or getSelectedIndex( ). These methods are shown here:

String getSelectedItem( )
int getSelectedIndex( )

The getSelectedItem( ) method returns a string containing the name of the item. If more than one item is selected, or if no selection has yet been made, null is returned. getSelectedIndex( )
returns the index of the item. The first item is at index 0. If more than one item is selected, or if no selection has yet been made, –1 is returned.

getSelectedItems( ) returns an array containing the names of the currently selected items. getSelectedIndexes( ) returns an array containing the indexes of the currently selected items. To obtain the number of items in the list, call getItemCount( ). You can set the currently selected item by using the select( ) method with a zero-based integer index. These methods are shown here:

int getItemCount( )
void select(int index)

Given an index, you can obtain the name associated with the item at that index by calling getItem( ), which has this general form:

String getItem(int index)

Here, index specifies the index of the desired item.

To process list events, you will need to implement the ActionListener interface. Each time a List item is double-clicked, an ActionEvent object is generated. Its getActionCommand( ) method can be used to retrieve the name of the newly selected item. Also, each time an item is selected or deselected with a single click, an ItemEvent object is generated. Its

getStateChange( )

method can be used to determine whether a selection or deselection triggered this event. **getItemSelectable( )** returns a reference to the object that triggered this event. Here is an example that converts the Choice controls in the preceding section into List components, one multiple choice and the other single choice

```
// Demonstrate Lists.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="ListDemo" width=300 height=180>
</applet>
*/
```

```
public class ListDemo extends Applet implements ActionListener {
    List os, browser;
    String msg = "";
    public void init() {
    os = new List(4, true);
    browser = new List(4, false);
    // add items to os list
    os.add("Windows XP");
    os.add("Windows Vista");
    os.add("Solaris");
    os.add("Mac OS");
   // add items to browser list
   browser.add("Internet Explorer");
   browser.add("Firefox");
   browser.add("Opera");
   browser.select(1);
   // add lists to window
   add(os);
   add(browser);
   // register to receive action events
   os.addActionListener(this);
   browser.addActionListener(this);
  }
   public void actionPerformed(ActionEvent ae) {
       repaint();
  }
    // Display current selections.
  public void paint(Graphics g) {
      int idx[ ];
      msg = "Current OS: ";
      idx = os.getSelectedIndexes();
      for(int i=0; i<idx.length; i++)
      msg += os.getItem(idx[i]) + " ";
      g.drawString(msg, 6, 120);
      msg = "Current Browser: ";
      msg += browser.getSelectedItem();
      g.drawString(msg, 6, 140);
   }
}
```

3.1.7 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 (or thumb) for the scroll bar. The slider box can be dragged by the user to a new position. The scroll bar will then reflect this value. In the background space on either side of the thumb, the user can click to cause the thumb to jump in that direction by some increment larger than 1. Typically, this

action translates into some form of page up and page down. Scroll bars are encapsulated by the Scrollbar class. Scrollbar defines the following constructors:

Scrollbar( ) throws HeadlessException
Scrollbar(int style) throws HeadlessException
Scrollbar(int style, int initialValue, int thumbSize, int min, int max)
throws HeadlessException

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.

If you construct a scroll bar by using one of the first two constructors, then you need to set its parameters by using setValues( ), shown here, before it can be used:
void **setValues(int initialValue, int thumbSize, int min, int max)**
The parameters have the same meaning as they have in the third constructor just described. To obtain the current value of the scroll bar, call getValue( ). It returns the current setting. To set the current value, call setValue( ). These methods are as follows:
int getValue( )
void setValue(int newValue)
Here, newValue specifies the new value for the scroll bar. When you set a value, the slider box inside the scroll bar will be positioned to reflect the new value. You can also retrieve the minimum and maximum values via getMinimum( ) and getMaximum( ), shown here:
int getMinimum( )
int getMaximum( )
They return the requested quantity. By default, 1 is the increment added to or subtracted from the scroll bar each time it is scrolled up or down one line. You can change this increment by calling setUnitIncrement( ).
By default, page-up and page-down increments are 10. You can change this value by calling setBlockIncrement( ). These methods are shown here:
void setUnitIncrement(int newIncr)
void setBlockIncrement(int newIncr)
To process scroll bar events, you need to implement the AdjustmentListener interface. Each time a user interacts with a scroll bar, an AdjustmentEvent object is generated. Its getAdjustmentType( ) method can be used to determine the type of the adjustment. The types of adjustment events are as follows:
BLOCK_DECREMENT- A page-down event has been generated.
BLOCK_INCREMENT -A page-up event has been generated.
TRACK -An absolute tracking event has been generated.
UNIT_DECREMENT- The line-down button in a scroll bar has been pressed.
UNIT_INCREMENT-The line-up button in a scroll bar has been pressed.

// **Demonstrate scroll bars.**
```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
```

```
<applet code="SBDemo" width=300 height=200>
</applet>
*/
  public class SBDemo extends Applet
     implements AdjustmentListener, MouseMotionListener {
       String msg = " ";
       Scrollbar vertSB, horzSB;
       public void init() {
           int width = Integer.parseInt(getParameter("width"));
           int height = Integer.parseInt(getParameter("height"));
          vertSB = new Scrollbar(Scrollbar.VERTICAL, 0, 1, 0, height);
           horzSB = new Scrollbar(Scrollbar.HORIZONTAL, 0, 1, 0, width);
           add(vertSB);
           add(horzSB);
           // register to receive adjustment events
           vertSB.addAdjustmentListener(this);
           horzSB.addAdjustmentListener(this);
           addMouseMotionListener(this);
       }
      public void adjustmentValueChanged(AdjustmentEvent ae) {
         repaint( );
    }
   // Update scroll bars to reflect mouse dragging.
   public void mouseDragged(MouseEvent me) {
     int x = me.getX();
     int y = me.getY();
     vertSB.setValue(y);
     horzSB.setValue(x);
     repaint();
  }
   // Necessary for MouseMotionListener
        public void mouseMoved(MouseEvent me) {
     }
  // Display current value of scroll bars.
      public void paint(Graphics g) {
           msg = "Vertical: " + vertSB.getValue();
           msg += ", Horizontal: " + horzSB.getValue( );
           g.drawString(msg, 6, 160);
    // show current mouse drag position
         g.drawString("*", horzSB.getValue(),
         vertSB.getValue());
  }
 }
```

3.1.8 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. TextField defines the following constructors:

TextField( ) throws HeadlessException
TextField(int numChars) throws HeadlessException
TextField(String str) throws HeadlessException
TextField(String str, int numChars) throws HeadlessException
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.
TextField (and its superclass TextComponent) provides several methods that allow you to utilize a text field. To obtain the string currently contained in the text field, call getText( ). To set the text, call setText( ). These methods are as follows:

String getText( )
void setText(String str)
Here, str is the new string. The user can select a portion of the text in a text field. Also, you can select a portion of text under program control by using select( ). Your program can obtain the currently selected text by calling getSelectedText( ). These methods are shown here:

String getSelectedText( )
void select(int startIndex, int endIndex)
getSelectedText( ) returns the selected text. The select( ) method selects the characters beginning at startIndex and ending at endIndex–1.

You can control whether the contents of a text field may be modified by the user by calling setEditable( ). You can determine editability by calling isEditable( ). These methods are shown here:
boolean isEditable( )
void setEditable(boolean canEdit)
isEditable( ) returns true if the text may be changed and false if not. In setEditable( ), if canEdit is true, the text may be changed. If it is false, the text cannot be altered. There may be times when you will want the user to enter text that is not displayed, such as a password. You can disable the echoing of the characters as they are typed by calling setEchoChar( ). This method specifies a single character that the TextField will display when characters are entered (thus, the actual characters typed will not be shown). You can check a text field to see if it is in this mode with the echoCharIsSet( ) method. You can retrieve the echo character by calling the getEchoChar( ) method. These methods are as follows:
void setEchoChar(char ch)
boolean echoCharIsSet( )
char getEchoChar( )
Here, ch specifies the character to be echoed. Since text fields perform their own editing functions, your program generally will not respond to individual key events that occur within a text field. However, you may want to respond when the user presses ENTER. When this occurs, an action event is generated. Here is an example that creates the classic user name and password screen:

// **Demonstrate text field.**
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*

```
<applet code="TextFieldDemo" width=380 height=150>
</applet>
*/
    public class TextFieldDemo extends Applet
        implements ActionListener {
            TextField name, pass;
            public void init() {
                Label namep = new Label("Name: ", Label.RIGHT);
                Label passp = new Label("Password: ", Label.RIGHT);
                name = new TextField(12);
                pass = new TextField(8);
                pass.setEchoChar('?');
                add(namep);
    add(name);
    add(passp);
                add(pass);
                // register to receive action events
                    name.addActionListener(this);
                    pass.addActionListener(this);
                }
      // User pressed Enter.
     public void actionPerformed(ActionEvent ae) {
            repaint( );
     }
        public void paint(Graphics g) {
            g.drawString("Name: " + name.getText(), 6, 60);
          g.drawString("Selected text in name: " + name.getSelectedText(), 6, 80);
          g.drawString("Password: " + pass.getText(), 6, 100);
      }
}
```

3.1.9 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:

TextArea( ) throws HeadlessException
TextArea(int numLines, int numChars) throws HeadlessException
TextArea(String str) throws HeadlessException
TextArea(String str, int numLines, int numChars) throws HeadlessException
TextArea(String str, int numLines, int numChars, int sBars) throws HeadlessException

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: SCROLLBARS_BOTH, SCROLLBARS_NONE, SCROLLBARS_HORIZONTAL_ONLY, SCROLLBARS_VERTICAL_ONLY.

TextArea is a subclass of TextComponent. Therefore, it supports the getText( ), setText( ), getSelectedText( ), select( ), isEditable( ), and setEditable( ) methods described in the preceding section. TextArea adds the following methods

void append(String str)
void insert(String str, int index)
void replaceRange(String str, int startIndex, int endIndex)
The append( ) method appends the string specified by str to the end of the current text. insert( ) method inserts the string passed in str at the specified index. To replace text, call replaceRange( ).It replaces the characters fromstartIndex to endIndex–1, with the replacement text passed in str. Text areas are almost self-contained controls. Your program incurs virtually no management overhead. Text areas only generate got-focus and lost-focus events.

Normally, your program simply obtains the current text when it is needed. The following program creates a TextArea control:

```
// Demonstrate TextArea.
import java.awt.*;
import java.applet.*;
/*
<applet code="TextAreaDemo" width=300 height=250>
</applet>
*/
  public class TextAreaDemo extends Applet {
       public void init() {
           String val =
"Java SE 6 is the latest version of the most\n" +
"widely-used computer language for Internet programming.\n" +
"Building on a rich heritage, Java has advanced both\n" +
"the art and science of computer language design.\n\n" +
"One of the reasons for Java's ongoing success is its\n" +
"constant, steady rate of evolution. Java has never stood\n" +
"still. Instead, Java has consistently adapted to the\n" +
"rapidly changing landscape of the networked world.\n" +
"Moreover, Java has often led the way, charting the\n" +
"course for others to follow.";
           TextArea text = new TextArea(val, 10, 30);
           add(text);
       }
   }
```

3.2 Layout Managers

Layout manager automatically arranges controls within a window by using some type of algorithm. If you have programmed for other GUI environments, such as Windows, then you are accustomed to laying out your controls by hand. While it is possible to lay out Java controls by hand, too,you generally won‘t want to, for two main reasons. First, it is very tedious to manually layout a large number of components. Second, sometimes the width and height information is not yet available when you need to arrange some control, because the native toolkit components haven‘t been realized.

Each Container object has a layout manager associated with it. A layout manager is 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. Whenever a container is resized (or sized for the first time), the layout manager is used to position each of the components within it. The setLayout( ) method has the following general form:

void setLayout(LayoutManager layoutObj)

Here, layoutObj is a reference to the desired layout manager. If you wish to disable the layout manager and position components manually, pass null for layoutObj. If you do this, you will need to determine the shape and position of each component manually, using the setBounds( ) method defined by Component. Normally, you will want to use a layout manager.

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. These return the preferred and minimum size required to display each component. The layout manager will honor these requests if at all possible, while maintaining the integrity of the layout policy. You may override these methods for controls that you subclass. Default values are provided otherwise.

3.2.1 FlowLayout

FlowLayout is the default layout manager. FlowLayout implements a simple layout style, which is similar to how words flow in a text editor. The direction of the layout is governed by the container‘s component orientation property, which, by default, is left to right, top to bottom. Therefore, by default, components are laid out line-by-line beginning at the upper- left corner. In all cases, when a line is filled, layout advances to the next line. A small space is left between each component, above and below, as well as left and right. Here are the constructors for FlowLayout:

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

The first form creates the default layout, which centers components and leaves five pixels of space between each component. The second form lets you specify how each line is aligned. Valid values for how are as follows:

FlowLayout.LEFT
FlowLayout.CENTER
FlowLayout.RIGHT
FlowLayout.LEADING

FlowLayout.TRAILING
These values specify left, center, right, leading edge, and trailing edge alignment, respectively. The third constructor allows you to specify the horizontal and vertical space left between components in horz and vert, respectively. Here is a version of the CheckboxDemo applet shown earlier in this chapter, modified so that it uses left-aligned flow layout:

// Use left-aligned flow layout.

```
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 winXP, winVista, solaris, mac;
public void init( ) {
// set left-aligned flow layout
setLayout(new FlowLayout(FlowLayout.LEFT));
winXP = new Checkbox("Windows XP", null, true);
winVista = new Checkbox("Windows Vista");
solaris = new Checkbox("Solaris");
mac = new Checkbox("Mac OS");
add(winXP);
add(winVista);
add(solaris);
add(mac);
// register to receive item events
winXP.addItemListener(this);
winVista.addItemListener(this);
solaris.addItemListener(this);
mac.addItemListener(this);
}
// Repaint when status of a check box changes.
public void itemStateChanged(ItemEvent ie) {
repaint();
}
// Display current state of the check boxes.
public void paint(Graphics g) {
msg = "Current state: ";
g.drawString(msg, 6, 80);
msg = " Windows XP: " + winXP.getState();
g.drawString(msg, 6, 100);
msg = " Windows Vista: " + winVista.getState();
g.drawString(msg, 6, 120);
msg = " Solaris: " + solaris.getState();
g.drawString(msg, 6, 140);
msg = " Mac: " + mac.getState();
```

```
g.drawString(msg, 6, 160);
}
}
```

3.2.2 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 constants that specify the regions as 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:

```
// Demonstrate BorderLayout.
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 = "The reasonable man adapts " + "himself to the world;\n" +
       "the unreasonable one persists in " +
       "trying to adapt the world to himself.\n" +
       "Therefore all progress depends " +
       "on the unreasonable man.\n\n" + " - George Bernard Shaw\n\n";
        add(new TextArea(msg), BorderLayout.CENTER);
    }
}
```

3.2.3 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. Either numRows or numColumns can be zero. Specifying numRows as zero allows for unlimited- length columns. Specifying numColumns as zero allows for unlimited-length rows. Here 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));
                  }
               }
            }
 }
```

3.2.4 Menu Bars and Menus

A top-level window can have a menu bar associated with it. A menu bar displays a list of top-level menu choices. Each choice is associated with a drop-down menu. This concept is implemented in the AWT by the following classes:MenuBar,Menu, andMenuItem. In general, a menu bar contains one or more Menu objects. EachMenu object contains a list of MenuItem objects. Each MenuItem object represents something that can be selected by the user. SinceMenu is a subclass ofMenuItem, a hierarchy of nested submenus can be created. It is also possible to include checkable menu items. These are menu options of type CheckboxMenuItem and will have a check mark next to them when they are selected. To create a menu bar, first create an instance ofMenuBar. This class only defines the default

constructor. Next, create instances ofMenu that will define the selections displayed on the bar. Following are the constructors for Menu:

Menu( ) throws HeadlessException
Menu(String optionName) throws HeadlessException
Menu(String optionName, boolean removable) throws HeadlessException
Here, optionName specifies the name of the menu selection. If removable is true, the menu can be removed and allowed to float free. Otherwise, it will remain attached to the menu bar. (Removable menus are implementation-dependent.) The first form creates an empty menu. Individual menu items are of typeMenuItem. It defines these constructors:
MenuItem( ) throws HeadlessException
MenuItem(String itemName) throws HeadlessException
MenuItem(String itemName, MenuShortcut keyAccel) throws HeadlessException
Here, itemName is the name shown in the menu, and keyAccel is the menu shortcut for this item. You can disable or enable a menu item by using the setEnabled( ) method. Its form is shown here:
void setEnabled(boolean enabledFlag)
If the argument enabledFlag is true, the menu item is enabled. If false, the menu item is disabled. You can determine an item‘s status by calling **isEnabled( ).** This method is shown here:
boolean isEnabled( )
isEnabled( ) returns true if the menu item on which it is called is enabled. Otherwise, it returns false. You can change the name of a menu item by calling **setLabel( ).** You can retrieve the current name by using getLabel( ). These methods are as follows:
void setLabel(String newName)
String getLabel( )
Here, newName becomes the new name of the invoking menu item. getLabel( ) returns the current name. You can create a checkable menu item by using a subclass of MenuItem called CheckboxMenuItem. It has these constructors:
CheckboxMenuItem( ) throws HeadlessException
CheckboxMenuItem(String itemName) throws HeadlessException
CheckboxMenuItem(String itemName, boolean on) throws HeadlessException

Here, itemName is the name shown in the menu. Checkable items operate as toggles. Each time one is selected, its state changes. In the first two forms, the checkable entry is unchecked. In the third form, if on is true, the checkable entry is initially checked. Otherwise, it is cleared. You can obtain the status of a checkable item by calling getState( ). You can set it to a known state by using setState( ). These methods are shown here:
boolean getState( )
void setState(boolean checked)

If the item is checked, getState( ) returns true. Otherwise, it returns false. To check an item, pass true to setState( ). To clear an item, pass false. Once you have created a menu item, you must add the item to a Menu object by using add( ), which has the following general form:
MenuItem add(MenuItem item)

Here, item is the item being added. Items are added to a menu in the order in which the calls to add( ) take place. The item is returned. Once you have added all items to a Menu object, you can add that object to the menu bar by using this version of add( ) defined by MenuBar:

Menu add(Menu menu)

Here, menu is the menu being added. The menu is returned. Menus only generate events when an item of type MenuItem or CheckboxMenuItem is selected. They do not generate events when a menu bar is accessed to display a drop-down menu, for example. Each time a menu item is selected, an ActionEvent object is generated. By default, the action command string is the name of the menu item. However, you can specify a different action command string by calling setActionCommand( ) on the menu item. Each time a check box menu item is checked or unchecked, an ItemEvent object is generated. Thus, you must implement the ActionListener and/or ItemListener interfaces in order to handle these menu events. The getItem( ) method of ItemEvent returns a reference to the item that generated this event. The general form of this method is shown here:

Object getItem( )

Assignment 2

MCQ

1. Which is the container that doesn't contain title bar and MenuBars but it can have other components like button, textfield etc?

A. Window B. Frame C. Panel D. Container
Answer: Option C

2. Which package provides many event classes and Listener interfaces for event handling?

A. java.awt B. java.awt.Graphics
C. java.awt.event D. None of the above
Answer: Option C

3. In Graphics class which method is used to draws a rectangle with the specified width and height?

A. public void drawRect(int x, int y, int width, int height)
B. public abstract void fillRect(int x, int y, int width, int height)
C. public abstract void drawLine(int x1, int y1, int x2, int y2)
D. public abstract void drawOval(int x, int y, int width, int height)
Answer: Option A

4. Name the class used to represent a GUI application window, which is optionally resizable and can have a title bar, an icon, and menus.

A. Window
B. Panel
C. Dialog
D. Frame
Answer: Option D

5. To use the ActionListener interface it must be implemented by a class there are several ways to do that find in the following?

A. Creating a new class B. Using the class the graphical component
C. An anonymous inner class D. All mentioned above
Answer: Option D

6. Which is a component in AWT that can contain another component like buttons, text fields, labels etc.?

A. Window
B. Container
C. Panel
D. Frame
Answer: Option B

7. Which is used to store data and partial results, as well as to perform dynamic linking, return values for methods, and dispatch exceptions?

A. Window
B. Panel
C. Frame
D. Container
Answer: Option C

8. What are the different types of controls in AWT?

A. Labels
B. Pushbuttons
C. Checkboxes
D. Choice lists
E. All of these
Answer: Option E

9. Which class provides many methods for graphics programming?

A. java.awt
B. java.Graphics
C. java.awt.Graphics
D. java. awt.event
Answer: Option C

10. By which method You can set or change the text in a Label?

A. setText()
B. getText()
C. Both A & B
D. None of the above
Answer: Option A

11. Which class can be used to represent a checkbox with a textual label that can appear in a menu.

A. MenuBar
B. MenuItem
C. CheckboxMenuItem
D.Menu
Answer: Option C

12. How many types of controls does AWT supports these controls are subclasses of component?

A. 7
B. 6
C. 5
D. 8

Answer: Option A

13. Which are passive controls that do not support any interaction with the user?

A. Choice
B. List
C. Labels
D. Checkbox
Answer: Option C

14. Which of the following classes are derived from the Component class.

A. Container
B. Window
C. List
D.MenuItem
Answer: Option D

15. How many ways can we align the label in a container?

A. 1
B. 2
C. 3
D. 4
Answer: Option C

16. Which method is used to set the graphics current color to the specified color in the graphics class?
A. public abstract void setFont(Font font)
B. public abstract void setColor(Color c)

C. public abstract void drawString(String str, int x, int y)
D. None of the above
Answer: Option B

17. Which object can be constructed to show any number of choices in the visible window?
A. Labels
B. Choice
C. List
D. Checkbox
Answer: Option C

18. Which class is used for this Processing Method processActionEvent( )?
A. Button,List,MenuItem
B. Button,Checkbox,Choice
C. Scrollbar,Component,Button
D. None of the above
Answer: Option A

19. Which package provides many event classes and Listener interfaces for event handling?
A. java.awt
B. java.awt.Graphics
C. java.awt.event
D. None of the above
Answer: Option C

20. Name the class used to represent a GUI application window, which is optionally resizable and can have a title bar, an icon, and menus.

A. Window
B. Panel
C. Dialog
D. Frame
Answer: Option D

21. What does the following line of code do?
Textfield text = new Textfield(10);
A. Creates text object that can hold 10 rows of text.
B. Creates the object text and initializes it with the value 10.
C. The code is illegal.
D. Creates text object that can hold 10 columns of text.

Answer: Option D
22. Which of the following methods can be used to remove a java.awt.Component object from the display?
A. delete()
B. remove()
C. disappear()
D. hide()
Answer: Option D

23. The setBackground() method is part of the following class in java.awt package:
A. Component
B. Graphics
C. Applet
D. Container
Answer: Option A
24. When we invoke **repaint()** for a java.awt.Component object, the AWT invokes the method:
A. update()
B. draw()
C. show()
D. paint()

Answer: Option A
25. Where are the following four methods commonly used?

1) public void add(Component c)

2) public void setSize(int width,int height)
3) public void setLayout(LayoutManager m)
4) public void setVisible(boolean)

A. Graphics class B. Component class
C. Both A & B D. None of the above
Answer: Option B

Long Answer Questions

1. Illustrate the use of Label with suitable example.
2. List and explain various methods associated with Button.
3. Explain the purpose of Choice controls with an example.
4. Explain List control with an example.
5. Explain Scroll Bars.
6. Explain about Layout Managers.
7. With an example explain FlowLayout.
8. With an example explain BoarderLayout.
9. What is the purpose of layout managers? With an example explain the use of Grid Layout.
10. Name the controls required for creating a menu. With an example explain the creation of a menu.
11. List and explain different methods associated with check boxes.
12. What is the purpose of TextField class? Explain any three methods of TextField class with syntax and example.