Chapter - 19

AWT Controls

AWT Controls

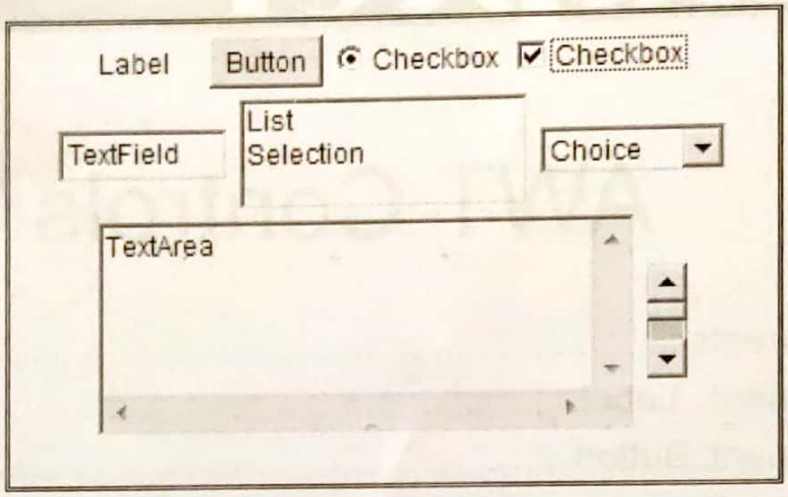
Components

Java's Abstract Windowing Toolkit provides many of the user interface objects we find in the Windows environment. These are called "Components" of the Java AWT. Most of the components we will use to create a graphical user interface (GUI) for our applets.

Components are graphical user interface (GUI) widgets like checkboxes, menus, windows, buttons, text fields, applets, and more.

In Java all components are subclasses of java.awt.Component. Subclasses of Component include

- Label
- TextField
- TextArea
- Button
- List
- Choice
- Checkbox
- Scrollbar



These components are used to develop GUI applications in applets. Components act as an interface between application and user. Components take input from the user and gives output. Components are also called as controls.

The following steps to be followed to develop GUI applications using controls.

1. Create objects to controls.
2. Add objects to containers.(panel, canvas, etc)
3. Register these objects(controls) with listeners.
4. Write the handlers to handle the events.

Component : Label

A Label object is a component for placing text in a container. A label displays a single line of read-only text. The text can be changed by the application, but a user cannot edit it directly. Label controls do not raise events therefore they need not be registered with any events.

The following table shows fields of Label class.

Fields	Meaning
static int **CENTER**	Indicates that the text on label should be centered.
static int **LEFT**	Indicates that the text on label should be left justified.
static int **RIGHT**	Indicates that the text on label should be right justified.

The following table shows constructors of **Label** class.

Constructors	Meaning
Label()	Constructs an empty label.
Label(String text)	Constructs a new label with the specified string of text, left justified.
Label(String text, int alignment)	Constructs a new label that presents the specified string of text with the specified alignment. Alignment can be LEFT, RIGHT, CENTER

The following table shows methods of **Label** class.

Methods	Meaning
void **addNotify**()	Creates the peer for this label.
AccessibleContext **getAccessibleContext**()	Gets the AccessibleContext associated with this Label.
int **getAlignment**()	Gets the current alignment of this label.
String **getText**()	Gets the text of this label.
protected String **paramString**()	Returns a string representing the state of this Label.
void **setAlignment**(int alignment)	Sets the alignment for this label to the specified alignment.
void **setText**(String text)	Sets the text for this label to the specified text.

setAlignment()

This method sets the alignment for the label to the specified alignment. Possible values are Label.LEFT, Label.RIGHT, and Label.CENTER.

public void **setAlignment**(int alignment)

getAlignment()

This method returns the current alignment of this label. Possible values are Label.LEFT, Label.RIGHT, and Label.CENTER.

public int **getAlignment**()

setText()

This method can be used to set the text for the label to the specified text.

public void **setText**(String text)

getText()

This method returns the text on the label.

public String **getText**()

A program to demonstrate Label control.

In this program, Three label controls are created and set with the text. These controls are then added to the applet container.

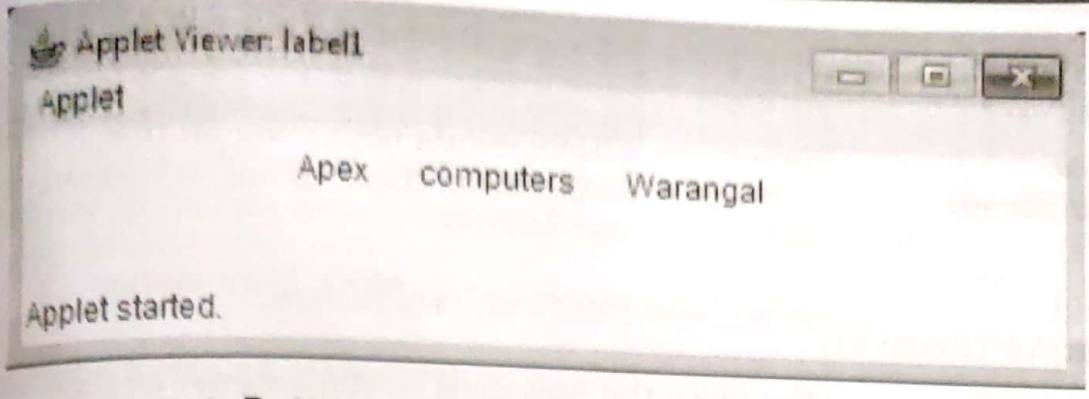
Bin>edit label1.java

```
import java.applet.*;
import java.awt.*;
public class label1 extends Applet
{
    Label l1,l2,l3;
    public void init()
    {
          l1=new Label("Apex");
          l2=new Label("computers");
          l3=new Label("Warangal");
          add(l1);
          add(l2);
          add(l3);
    }
}
```

Bin>javac label1.java

Bin>edit label1.html

```
<applet code="label1" width=400 height=400>
</applet>
```



Component: Button

The Button class can be used to create button component. The button component fires an ActionEvent when it is selected using mouse or keyboard. The button component should be registered with ActionListener using addActionListener() method so that it receives the ActionEvent raised on button.

The following table shows the constructors of Button class

Constructors	Meaning
Button()	Constructs a button with an empty string for its label.
Button(String label)	Constructs a button with the specified label.

The following table shows methods of Button class.

Methods	Meaning
void **addActionListener**(ActionListener l)	Adds the specified action listener to receive action events from this button.
void **addNotify**()	Creates the peer of the button.
AccessibleContext **getAccessibleContext**()	Gets the AccessibleContext associated with this Button.
String **getActionCommand**()	Returns the command name of the action event fired by this button.
ActionListener []**getActionListeners**()	Returns an array of all the action listeners registered on this button.
String **getLabel**()	Gets the label of this button.
<T extends EventListener> T[]**getListeners**(Class<T> listenerType)	Returns an array of all the objects currently registered as *Foo*Listeners upon this Button.

protected String **paramString**()	Returns a string representing the state of this Button.
protected void **processActionEvent**(ActionEvent e)	Processes action events occurring on this button by dispatching them to any registered ActionListener objects.
protected void **processEvent**(AWTEvent e)	Processes events on this button.
void **removeActionListener**(ActionListener l)	Removes the specified action listener so that it no longer receives action events from this button.
void **setActionCommand**(String command)	Sets the command name for the action event fired by this button.
void **setLabel**(String label)	Sets the button's label to be the specified string.

getLabel()

This method of Button class returns the label of button i.e. it returns the text present on the button.

public String **getLabel**()

setLabel()

This method of Button class sets the label on the button i.e. the text that should be displayed on the button.

public void **setLabel**(String label)

setActionCommand()

This method of Button class sets the command name(which returns by getActionCommand()) for the action event fired by the button. By default this action command is set to match the label of the button.

public void **setActionCommand**(String command)

The argument **command** is a string used to set the button's action command. If the string is null then the action command is set to match the label of the button.

getActionCommand()

The method of Button class Returns the command name of the action event fired by this button. If the command name is null (default) then this method returns the label of the button.

public String **getActionCommand**()

addActionListener()

This method of Button class can be used to add the specified action listener to receive action events from the button component.

public void addActionListener(ActionListener l)

removeActionListener()

This method of Button class removes the specified ActionListener from the button component so that the component does not receive events from button component.

public void **removeActionListener**(ActionListener l)

The ActionEvent class

A semantic event which indicates that a component-defined action occurred. This high-level event is generated when a button or menu item is pressed or selected. The event is passed to every **ActionListener** object that registered to receive such events using the component's **addActionListener** method. When the event is received by the ActionListener, it calls the actionPerformed() method and passes ActionEvent argument to it.

The following table shows the fields of **ActionEvent** class.

Fields	Meaning
static int **ACTION_FIRST**	The first number in the range of ids used for action events.
static int **ACTION_LAST**	The last number in the range of ids used for action events.
static int **ACTION_PERFORMED**	This event id indicates that a meaningful action occured.
static int **ALT_MASK**	The alt modifier.
static int **CTRL_MASK**	The control modifier.
static int **META_MASK**	The meta modifier.
static int **SHIFT_MASK**	The shift modifier.

Constructors	Meaning
ActionEvent(Object source, int id, String command)	Constructs an ActionEvent object.
ActionEvent(Object source, int id, String command, int modifiers)	Constructs an ActionEvent object with modifier keys.
ActionEvent(Object source, int id, String command, long when, int modifiers)	Constructs an ActionEvent object with the specified modifier keys and timestamp.

In the above constructors, source is the source object that generates event. The **id** represents the event type, command is the command associated with the event and modifier specifies which modifier key was pressed when the event was occurred.

The following table shows methods of **ActionEvent** class.

Methods	Meaning
String **getActionCommand**()	Returns the command string associated with this action.
int **getModifiers**()	Returns the modifier keys held down during this action event.
long **getWhen**()	Returns the timestamp of when this event occurred.
String **paramString**()	Returns a parameter string identifying this action event.

A program to demonstrate button component

In this program, two buttons (Red and Green) are created. When Red button is clicked, the background of applet is applied with red color. Similarly, when Green button is clicked then background of applet is applied with green color.

Bin>edit button1.java

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
public class button1 extends Applet implements ActionListener
{
    Button b1,b2;
```

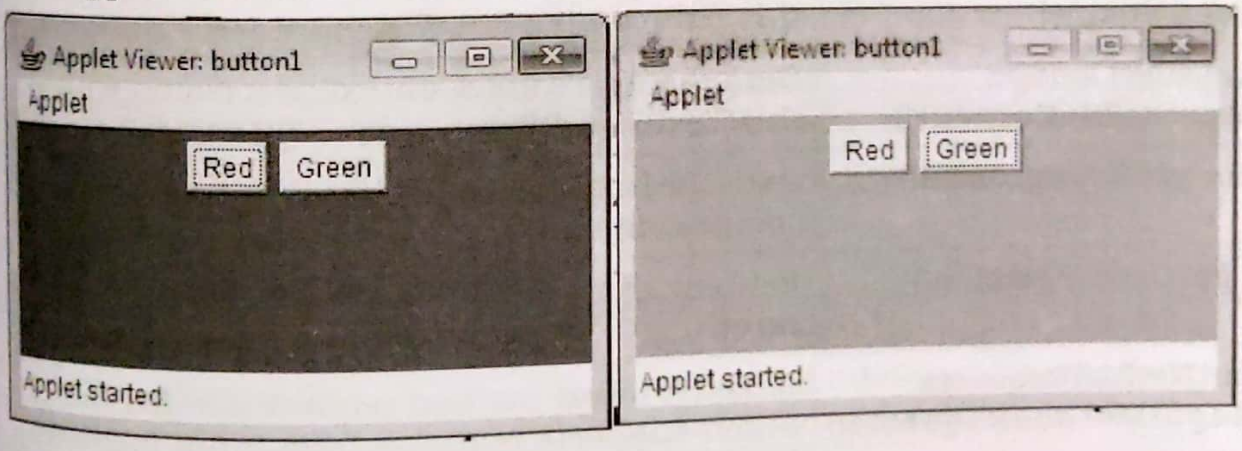
```
    public void init()
    {
          b1=new Button("Red");
          b2=new Button("Green");
          add(b1);add(b2);
          b1.addActionListener(this);
          b2.addActionListener(this);
    }
    public void actionPerformed(ActionEvent e)
    {
          String str=e.getActionCommand();
          if(str.equals("Red"))
                setBackground(Color.red);
          else if(str.equals("Green"))
                setBackground(Color.green);
    }
}
```

Bin>javac button1.java

Bin>copy con button1.html

```
<applet code="button1" width=400 height=400>
</applet>
```

Bin>appletviewer button1.html



Explanation:

When the **Red** button is clicked on the applet, the button raises **ActionEvent** which is received by **addActionListener(this);** and calls the **actionPerfomed(ActionEvent e)**. This method is passed with an argument of **ActionEvent** class. In the **actionPerformed()** method the statement **e.getActionCommand()** returns the label of button clicked i.e **Red** and assigns

to **str**. The first if condition is satisfied and set the background color as red. Same thing happens when Green button is clicked.

Class TextComponent

The **TextComponent** class is the superclass of any component that allows the editing of some text.

A text component embodies a string of text. The **TextComponent** class defines a set of methods that determine whether or not this text is editable. If the component is editable, it defines another set of methods that supports a text insertion caret.

In addition, the class defines methods that are used to maintain a current *selection* from the text. The text selection, a substring of the component's text, is the target of editing operations. It is also referred to as the *selected text*.

The following table shows methods of TextComponent class

Methods	Meaning
void **addNotify**()	Makes this Component displayable by connecting it to a native screen resource.
void **addTextListener**(TextListener l)	Adds the specified text event listener to receive text events from this text component.
void **enableInputMethods**(boolean enable)	Enables or disables input method support for this text component.
AccessibleContext **getAccessibleContext**()	Gets the AccessibleContext associated with this TextComponent.
Color **getBackground**()	Gets the background color of this text component.
int **getCaretPosition**()	Returns the position of the text insertion caret.
InputMethodRequests **getInputMethodRequests**()	Gets the input method request handler which supports requests from input methods for this component.
<T extends EventListener> T[]**getListeners**(Class<T> listenerType)	Returns an array of all the objects currently registered as *Foo*Listeners upon this TextComponent.
String **getSelectedText**()	Returns the selected text from the text that is presented by this text component.

int **getSelectionEnd**()	Gets the end position of the selected text in this text component.
int **getSelectionStart**()	Gets the start position of the selected text in this text component.
String **getText**()	Returns the text that is presented by this text component.
TextListener[] **getTextListeners**()	Returns an array of all the text listeners registered on this text component.
boolean **isEditable**()	Indicates whether or not this text component is editable.
protected String **paramString**()	Returns a string representing the state of this TextComponent.
protected void **processEvent**(AWTEvent e)	Processes events on this text component.
protected void **processTextEvent**(TextEvent e)	Processes text events occurring on this text component by dispatching them to any registered TextListener objects.
void **removeNotify**()	Removes the TextComponent's peer.
void **removeTextListener**(TextListener l)	Removes the specified text event listener so that it no longer receives text events from this text component If l is null, no exception is thrown and no action is performed.
void **select**(int selectionStart, int selectionEnd)	Selects the text between the specified start and end positions.
void **selectAll**()	Selects all the text in this text component.
void **setBackground**(Color c)	Sets the background color of this text component.
void **setCaretPosition**(int position)	Sets the position of the text insertion caret.
void **setEditable**(boolean b)	Sets the flag that determines whether or not this text component is editable.
void **setSelectionEnd**(int selectionEnd)	Sets the selection end for this text component to the specified position.
void **setSelectionStart**(int selectionStart)	Sets the selection start for this text component to the specified position.
void **setText**(String t)	Sets the text that is presented by this text component to be the specified text.

Component : TextField

The **TextField** class can be used to create textbox component. The **TextField** component takes the input from the user and gives the output. A text component allows for the editing of a single line of text.

The **TextField** class is a subclass of **TextComponent** class. The methods of **TextComponent** are available to **TextField** class.

Every time the user types a key in the text field, one or more key events are sent to the text field. A **KeyEvent** may be one of three types: **keyPressed**, **keyReleased**, or **keyTyped**.

The key event is passed to every **KeyListener** or **KeyAdapter** object which registered to receive such events using the component's **addKeyListener()** method.

It is also possible to fire an **ActionEvent**. If action events are enabled for the text field, they may be fired by pressing the Return key.

The **TextField** class's **processEvent** method examines the action event and passes it along to **processActionEvent**. The latter method redirects the event to any **ActionListener** objects that have registered to receive action events generated by this text field.

The following table shows the constructors of TextField class.

Constructors	Meaning
TextField()	Constructs a new text field and it will be empty.
TextField(int columns)	Constructs a new empty text field with the specified number of columns as width.
TextField(String text)	Constructs a new text field initialized with the specified text.
TextField(String text, int columns)	Constructs a new text field initialized with the specified text to be displayed, and wide enough to hold the specified number of columns.

The following table shows methods of TextField class.

Methods	Meaning
void **addActionListener**(ActionListener l)	Adds the specified action listener to receive action events from this text field.

void **addNotify**()	Creates the TextField's peer.
boolean **echoCharIsSet**()	Indicates whether or not this text field has a character set for echoing.
AccessibleContext **getAccessibleContext**()	Gets the AccessibleContext associated with this TextField.
ActionListener[] **getActionListeners**()	Returns an array of all the action listeners registered on this textfield.
int **getColumns**()	Gets the number of columns in this text field.
char **getEchoChar**()	Gets the character that is to be used for echoing.
<T extends EventListener> T[]**getListeners**(Class<T> listenerType)	Returns an array of all the objects currently registered as *Foo*Listeners upon this TextField.
Dimension **getMinimumSize**()	Gets the minumum dimensions for this text field.
Dimension **getMinimumSize**(int columns)	Gets the minumum dimensions for a text field with the specified number of columns.
Dimension **getPreferredSize**()	Gets the preferred size of this text field.
Dimension **getPreferredSize**(int columns)	Gets the preferred size of this text field with the specified number of columns.
protected void **processActionEvent**(ActionEvent e)	Processes action events occurring on this text field by dispatching them to any registered ActionListener objects.
protected void **processEvent**(AWTEvent e)	Processes events on this text field.
void **removeActionListener**(ActionListener l)	Removes the specified action listener so that it no longer receives action events from this text field.
void **setColumns**(int columns)	Sets the number of columns in this text field.
void **setEchoChar**(char c)	Sets the echo character for this text field.
void **setText**(String text)	Sets the given text to the text component.

setEchoChar()

This method of **TextField** class can be used to set the echo character for the text field component.

public void **setEchoChar**(char c)

getEchoChar()

This method of **TextField** class can be used to get the character that is to be used for echoing.

public char **getEchoChar**()

An echo character is useful for text fields where user input should not be echoed to the screen, as in the case of a text field for entering a password. Setting **echoChar = 0** allows user input to be echoed to the screen again.

setText()

The method of **TextField** class can be used to set the given text to the text field component.

public void **setText**(String text)

getText()

This method of **TextComponent** class can be used to get the text present in the text field component.

public String **getText**()

getSelectedText()

This method of **TextComponent** can be used to get the selected text from the text component.

public String **getSelectedText**()

select()

This method of **TextComponent** can be used to select the text between the specified start and end positions.

public void **select**(int selectionStart, int selectionEnd)

This method sets the start and end positions of the selected text, enforcing the restriction that the start position must be greater than or equal to zero. The end position must be greater than or equal to the start position, and less than or equal to the length of the text component's text. The character

positions are indexed starting with zero. The length of the selection is **endPosition - startPosition**, so the character at **endPosition** is not selected. If the start and end positions of the selected text are equal, all text is deselected.

A program to demonstrate TextField class.

In this program, two texfields and button components are created. When the button is clicked, the text of first textfield component is copied into second textfield component.

Bin>edit text1.java

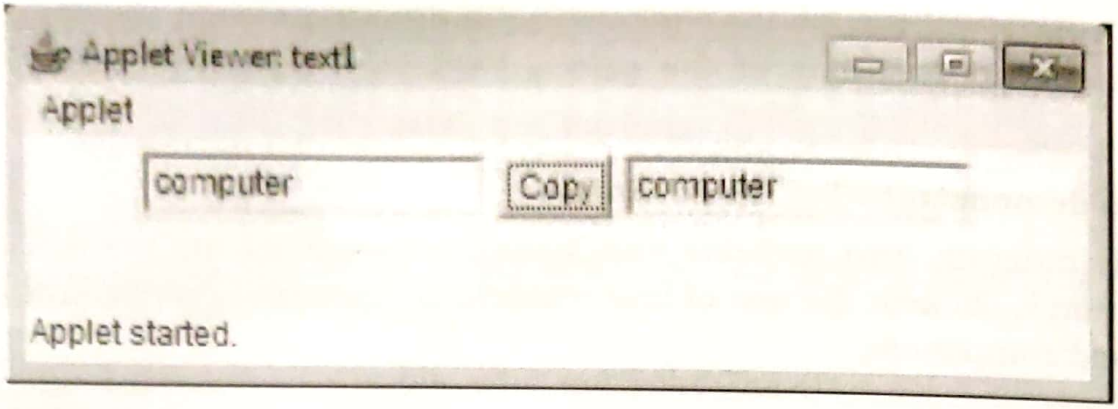
```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class text1 extends Applet implements ActionListener
{
    TextField t1,t2;
    Button b1;
    public void init()
    {
          t1=new TextField(15);
          t2=new TextField(15);
          b1=new Button("Copy");
          add(t1);add(b1);add(t2);
          b1.addActionListener(this);
    }
    public void actionPerformed(ActionEvent e)
    {
          String str=e.getActionCommand();
          if(str.equals("Copy"))
          {
                String s=t1.getText();
                t2.setText(s);
          }
    }
}
```

Bin>javac text1.java

Bin>copy con text1.html

```
<applet code="text1" width=400 height=400>
</applet>
```

Bin>appletviewer text1.html



Explanation:

When button is clicked, the **actionPerformed()** method is executed where **t1.getText()** returns the text of **t1** object and assigns to **s**. The **t2.setText(s)** sets the **s** text to **t2** object as shown in above output.

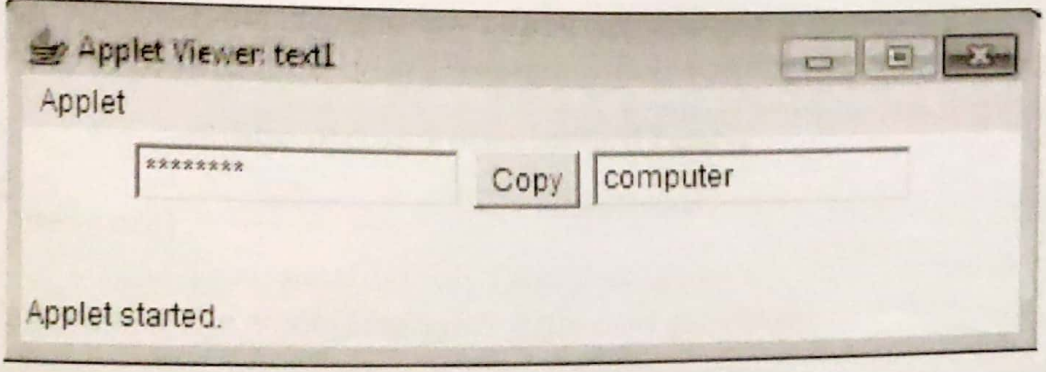
Creation of Password TextField

In the above program, t1 TextField object can be converted into password textfield component using setEchoChar() method.

Execute the above program replacing the statements in init() as

```
t1=new TextField(15);
t1.setEchoChar('*');
t2=new TextField(15);
b1=new Button("Copy");
add(t1);add(b1);add(t2);
b1.addActionListener(this);
```

We get the output as:



Component: Checkbox

The Checkbox class of java.awt package can be used to create checkbox component. This component allows to select or de-select an option. More than one checkbox can be selected or de-selected. Clicking on a checkbox changes its state from "on" (true) to "off" (false) , or from "off" to "on."

When a Checkbox component is selected or de-selected either using keyboard or mouse, the checkbox object generates an ItemEvent. Therefore CheckBox component object should be registered to receive ItemEvent and this can be done using the addItemListener() method. When the ItemEvent is received by the ItemListener it immediately calls itemStateChanged() method by passing an argument of ItemEvent class.

The following table shows constructors of **Checkbox** class.

Constructors	Meaning
Checkbox()	Creates a check box with an empty string for its label.
Checkbox(String label)	Creates a check box with the specified label.
Checkbox(String label, boolean state)	Creates a check box with the specified label and sets the specified state.
Checkbox(String label, boolean state, CheckboxGroup group)	Constructs a Checkbox with the specified label, set to the specified state, and in the specified check box group.
Checkbox(String label, CheckboxGroup group, boolean state)	Creates a check box with the specified label, in the specified check box group, and set to the specified state.

The following table shows methods of **Checkbox** class.

Method	Meaning
void **addItemListener**(ItemListener l)	Adds the specified item listener to receive item events from this check box.
void **addNotify**()	Creates the peer of the Checkbox.
AccessibleContext **getAccessibleContext**()	Gets the AccessibleContext associated with this Checkbox.
CheckboxGroup **getCheckboxGroup**()	Determines this check box's group.
ItemListener[] **getItemListeners**()	Returns an array of all the item listeners registered on this checkbox.
String **getLabel**()	Gets the label of this check box.
<T extends EventListener> T[]**getListeners**(Class<T> listenerType)	Returns an array of all the objects currently registered as *Foo*Listeners upon this Checkbox.

Object[] **getSelectedObjects**()	Returns an array (length 1) containing the checkbox label or null if the checkbox is not selected.
boolean **getState**()	Determines whether this check box is in the "on" or "off" state.
protected String **paramString**()	Returns a string representing the state of this Checkbox.
protected void **processEvent**(AWTEvent e)	Processes events on this check box.
protected void **processItemEvent**(ItemEvent e)	Processes item events occurring on this check box by dispatching them to any registered ItemListener objects.
void **removeItemListener**(ItemListener l)	Removes the specified item listener so that the item listener no longer receives item events from this check box.
void **setCheckboxGroup**(CheckboxGroup g)	Sets this check box's group to the specified check box group.
void **setLabel**(String label)	Sets this check box's label to be the string argument.
void **setState**(boolean state)	Sets the state of this check box to the specified state.

getLabel()

This method of Checkbox class returns the label of the check box.

public String **getLabel**()

setLabel()

This method of Checkbox class can be used to set the label of checkbox.

public void **setLabel**(String label)

getState()

This method of Checkbox class returns whether the check box is in the "on" or "off" state. It returns **true** if checkbox is selected and **false** if it is de-selected.

public boolean **getState**()

setState()

This method of Checkbox class can be used to set the state of the checkbox to select or de-select. The boolean value true indicates to select the checkbox and false indicates to de-select.

public void **setState**(boolean state)

addItemListener()

This method of Checkbox class can be used to add the specified item listener to receive item events from the checkbox component. Item events are sent to listeners in response to user input, but not in response to calls to setState().

public void **addItemListener**(ItemListener l)

removeItemListener

This method of Checkbox class removes the specified ItemListener for the checkbox component so that the ItemListener does not receive the events form the component.

public void **removeItemListener**(ItemListener l)

The ItemEvent class

A semantic event which indicates that an item was selected or deselected. This high-level event is generated by List or CheckBox when an item is selected or deselected by the user. The event is passed to every ItemListener object which registered to receive such events using the component's addItemListener() method.

The following table shows fields of ItemEvent class.

Fields	Meaning
static int **DESELECTED**	This state-change-value indicates that a selected item was deselected.
static int **ITEM_FIRST**	The first number in the range of ids used for item events.
static int **ITEM_LAST**	The last number in the range of ids used for item events.
static int **ITEM_STATE_CHANGED**	This event id indicates that an item's state changed.
static int **SELECTED**	This state-change value indicates that an item was selected.

The following table shows constructors of ItemEvent class.

Constructors	Meaning
ItemEvent(ItemSelectable source, int id, Object item, int stateChange)	Constructs an ItemEvent object.

The following table shows methods of ItemEvent class.

Methods	Meaning
Object **getItem**()	Returns the item affected by the event.
ItemSelectable **getItemSelectable**()	Returns the originator of the event.
int **getStateChange**()	Returns the type of state change (selected or deselected).
String **paramString**()	Returns a parameter string identifying this item event.

A program to demonstrate Checkbox component.

In this program, two checkboxes and textfields components are created. The state of checkboxes are shown in the textfields i.e. checkboxes are selection mode or de-selection mode.

Bin>edit check1.java

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class check1 extends Applet implements ItemListener
{
    Checkbox c1,c2;
    TextField t1,t2;
    public void init()
    {
        c1=new Checkbox("Check1",true);
        c2=new Checkbox("Check2",false);
        t1=new TextField(20);
        t2=new TextField(20);
        add(c1);add(t1);add(c2);add(t2);
        c1.addItemListener(this);
```

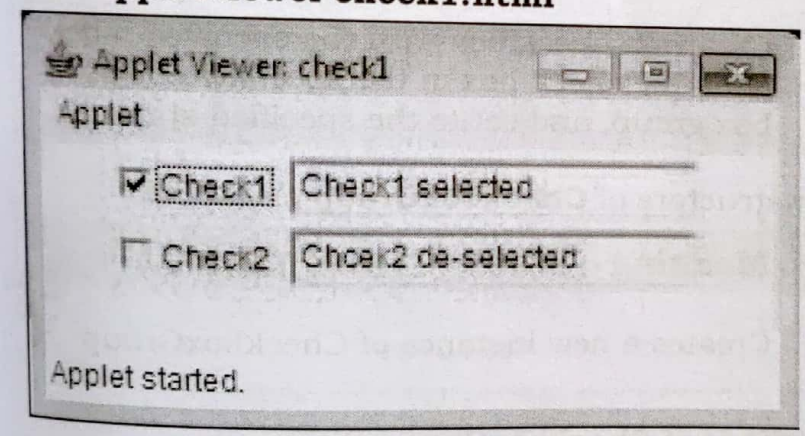
```
        c2.addItemListener(this);
    }
    public void itemStateChanged(ItemEvent i)
    {
        if(c1.getState()==true)
            t1.setText("Check1 selected");
        else
            t1.setText("Chcek1 de-selected");
        if(c2.getState()==true)
            t2.setText("Check2 selected");
        else
            t2.setText("Chcek2 de-selected");
    }
}
```

Bin>javac check1.java

Bin>copy con check1.html

```
<applet code="check1" width=400 height=400>
</applet>
```

Bin>appletviewer check1.html



Explanation:

When the check1 component is selected an **ItemEvent** event is generated as a result the **itemStateChanged()** method is called. In the method, the first if statement get true because **c1.getState()** returns **true** and the true statement executes as a result the **t1** textbox is set with the text **Check1 selected**. Where as **c2.getState()** returns **false** and false statement executes which sets **t2** textbox with **Check2 de-selected**.

Component:Radiobutton

Radiobutton component is similar to checkbox component with a difference that at any given time only one radio button can be selected. Radiobutton components can be created using Checkbox class. The checkbox components if grouped together then the checkbox components behave like radio buttons. To group the checkboxes Java provides a class CheckboxGroup.

For radio buttons also use ItemEvent class and ItemListener interface.

Class CheckboxGroup

The CheckboxGroup class is used to group together a set of Checkbox buttons. Exactly one check box button in a CheckboxGroup can be selected at any given time. Selection of any button forces any other button that was in selection mode become de-selected.

To make the check box join in the group, the object of check box has to pass to the constructor of Checkbox class. The following constructors of Checkbox class can be used to make the check box as radio button.

Methods	Meaning
Checkbox(String label, boolean state, CheckboxGroup group)	Constructs a Checkbox with the specified label, set to the specified state, and adds the checkbox in the specified check box group.
Checkbox(String label, CheckboxGroup group, boolean state)	Creates a check box with the specified label, adds the check box in the specified check box group, and set to the specified state.

The following table shows constructors of **CheckboxGroup** class.

Constructors	Meaning
CheckboxGroup()	Creates a new instance of CheckboxGroup

The following table shows methods of **CheckboxGroup** class.

Methods	Meaning
Checkbox **getSelectedCheckbox**()	Gets the current choice from this check box group.
void **setSelectedCheckbox**(Checkbox box)	Sets the currently selected check box in this group to be the specified check box.
String **toString**()	Returns a string representation of this check box group, including the value of its current selection.

getSelectedCheckbox()

This method of CheckboxGroup returns the selected check box object reference from the group.

public Checkbox **getSelectedCheckbox**()

A program to demonstrate radio buttons

Bin>edit radio1.java

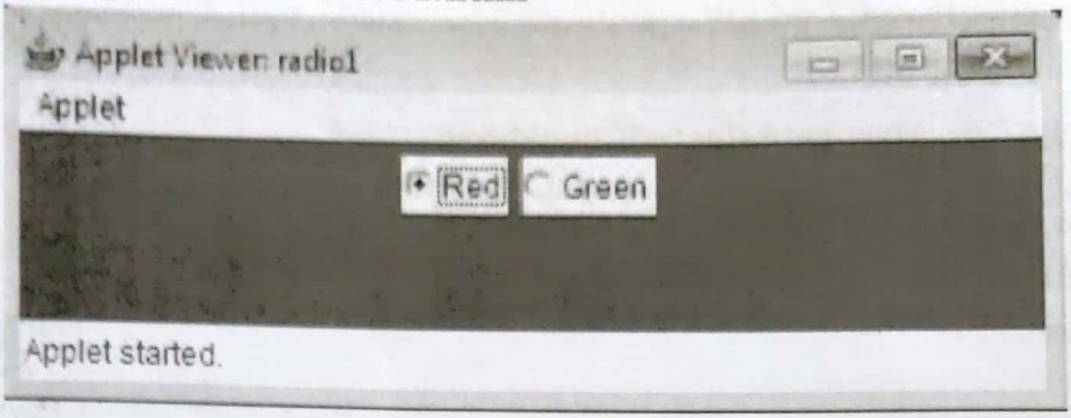
```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class radio1 extends Applet implements ItemListener
{
    Checkbox c1,c2;
    CheckboxGroup cb;
    public void init()
    {
          cb=new CheckboxGroup();
          c1=new Checkbox("Red",cb,true);
          c2=new Checkbox("Green",cb,false);
          add(c1);add(c2);
          c1.addItemListener(this);
          c2.addItemListener(this);
    }
    public void itemStateChanged(ItemEvent i)
    {
          Checkbox c=cb.getSelectedCheckbox();
          String str=c.getLabel();
          if(str.equals("Red"))
                setBackground(Color.red);
          else if(str.equals("Green"))
                setBackground(Color.green);
    }
}
```

Bin>javac radio1.java

Bin>copy con radio1.html

```
<applet code="radio1" width=400 height=400>
</applet>
```

Bin>appletviewer radio1.html



Explanation:

In the above program, in the **init()** method an object of **CheckboxGroup** is created and passes the same object to the two checkboxes, that means the checkboxes are grouped and becomes radio buttons. These components are registered with **ItemListener**. When the Red radio button is selected the **ItemEvent** is raised and calls the **itemStateChanged()** method. Inside the method, the **cb.getSelectedCheckbox()** returns the reference of selected checkbox(**Red**) to **c** reference variable. The **c.getLabel()** returns the label **"Red"** and assigns to **str**. The first condition gets true and the red color is applied to applet.

Component: Choice

The Choice class of java.awt can be used to create a popup menu with a fixed position. The choice component is a collection of items and at any given time only one item can be selected. When clicked on this component, it pops up a menu where it shows all items in it. This component shows large number of items in minimum space.

The Choice component should be registered with ItemListener using addItemListener() method so that it receives the events raised on choice.

The following table shows constructors of Choice class.

Constructors	Meaning
public **Choice**()	Creates a new choice menu. The menu initially has no items in it. By default, the first item added to the choice menu becomes the selected item, until a different selection is made.

The following table shows methods of **Choice** class.

Methods	Meaning
void **add**(String item)	Adds an item to this Choice menu.
void **addItem**(String item)	Obsolete as of Java 2 platform v1.1.
void **addItemListener**(ItemListener l)	Adds the specified item listener to receive item events from this Choice menu.
void **addNotify**()	Creates the Choice's peer.
int **countItems**()	**Deprecated.** *As of JDK version 1.1, replaced by getItemCount().*
AccessibleContext **getAccessibleContext**()	Gets the AccessibleContext associated with this Choice.
String	Gets the string at the specified index in this Choice menu.
int **getItemCount**()	Returns the number of items in this Choice menu.
ItemListener[] **getItemListeners**()	Returns an array of all the item listeners registered on this choice.
<T extends EventListener> T[]**getListeners**(Class<T> listenerType)	Returns an array of all the objects currently registered as *Foo*Listeners upon this Choice.
int **getSelectedIndex**()	Returns the index of the currently selected item.
String **getSelectedItem**()	Gets a representation of the current choice as a string.
Object[]**getSelectedObjects** ()	Returns an array (length 1) containing the currently selected item.
void **insert**(String item, int index)	Inserts the item into this choice at the specified position.
protected String **paramString**()	Returns a string representing the state of this Choice menu.
protected void **processEvent**(AWTEvent e)	Processes events on this choice.
protected void **processItemEvent**(ItemEvent e)	Processes item events occurring on this Choice menu by dispatching them to any registered ItemListener objects.

void **remove**(int position)	Removes an item from the choice menu at the specified position.
void **remove**(String item)	Removes the first occurrence of item from the Choice menu.
void **removeAll**()	Removes all items from the choice menu.
void **removeItemListener**(ItemListener l)	Removes the specified item listener so that it no longer receives item events from this Choice menu.
void **select**(int pos)	Sets the selected item in this Choice menu to be the item at the specified position.
void **select**(String str)	Sets the selected item in this Choice menu to be the item whose name is equal to the specified string.

getItemCount()

This method of Choice class returns the number of items in the Choice menu.

public int **getItemCount**()

getItem()

This method of Choice class returns the string at the specified index in the Choice menu.

public String **getItem**(int index)

add()

This method of Choice class can be used to add an item to the Choice menu.

public void **add**(String item)

insert()

This method of Choice class can be used to insert an item into the choice at the specified index position. Existing items at an index greater than or equal to index are shifted up by one to accommodate the new item. If index is greater than or equal to the number of items in this choice, item is added to the end of this choice.

If the item is the first one being added to the choice, then the item becomes selected. Otherwise, if the selected item was one of the items shifted, the first item in the choice becomes the selected item. If the selected item was no among those shifted, it remains the selected item.

public void **insert**(String item,int index)

remove()

This method of Choice class can be used to delete the first occurrence of item from the Choice menu. If the item being removed is the currently selected item, then the first item in the choice becomes the selected item. Otherwise, the currently selected item remains selected and the selected index is updated accordingly.

public void **remove**(String item)

remove()

This an overloaded method of Choice class that deletes an item from the choice menu at the specified index. If the item being removed is the currently selected item, then the first item in the choice becomes the selected item. Otherwise, the currently selected item remains selected and the selected index is updated accordingly.

public void **remove**(int index)

removeAll()

This method of Choice class can be used to delete all items from the choice menu.

public voi d **removeAll**()

getSelectedItem()

This method of Choice class returns the item of the currently selected item.

public String **getSelectedItem**()

getSelectedIndex()

This method of Choice class returns the index of the currently selected item. If nothing is selected, returns -1.

public int **getSelectedIndex**()

select()

This method of Choice class can be used to select an item at the given index in the Choice menu.

public void **select**(int index)

select()

The method is an overloaded method of Choice class that can be used to set the selected item in the Choice menu to be the item whose name is

equal to the specified string. If more than one item matches (is equal to) the specified string, the one with the smallest index is selected.

public void **select**(String str)

addItemListener()

This method of Choice class can be used to register the choice component with ItemListener to receive item events from the Choice menu.

public void **addItemListener**(ItemListener l)

removeItemListener()

This method of Choice class can be used to remove the ItemListener for the choice component listener so that the ItemListener does not receive the events from choice component.

public void **removeItemListener**(ItemListener l)

A program to demonstrate the Choice component

In this program, a choice control, three textfields and a button is created. The choice is added with three items(Red, Green and Blue). The first textfield is displayed with selected item and second textfield is displayed with selected index and when button is clicked the number of items is displayed in textfield3. Color is also displayed based on the selected item.

Bin>edit choice1.java

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class choice1 extends Applet implements
ItemListener, ActionListener
{
    Choice c;
    TextField t1,t2,t3;
    Button b1;
    public void init()
    {
          c=new Choice();
          c.add("Red");
          c.add("Green");
          c.add("Blue");
          t1=new TextField(10);
```

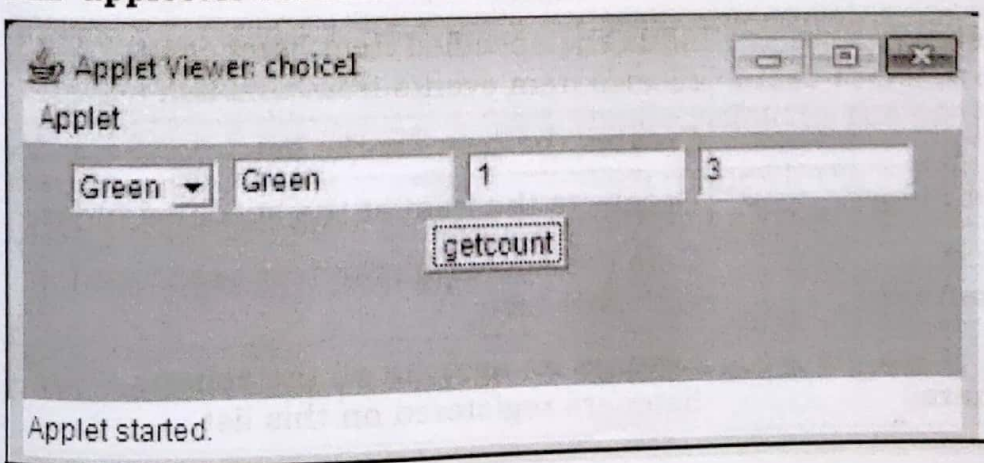
```
        t2=new TextField(10);
        t3=new TextField(10);
        b1=new Button("getcount");
        add(c);add(t1);add(t2);add(t3);
        add(b1);
        c.addItemListener(this);
        b1.addActionListener(this);
    }
    public void itemStateChanged(ItemEvent i)
    {
        String str=c.getSelectedItem();
        t1.setText(str);
        t2.setText(""+c.getSelectedIndex());
        if(str.equals("Red"))
            setBackground(Color.red);
        else if(str.equals("Green"))
            setBackground(Color.green);
        else if(str.equals("Blue"))
            setBackground(Color.blue);
    }
    public void actionPerformed(ActionEvent e)
    {
        t3.setText(""+c.getItemCount());
    }
}
```

Bin>javac choice1.java

Bin>copy con choice1.html

```
<applet code="choice1" width=400 height=400>
</applet>
```

Bin>appletviewer choice1.html



Component: List

The List class of java.awt can be used to create a list component. The List component is a collection of items similar to choice. List component occupies more space than choice. List component can show more than one item and remaining items can be selected by scrolling items. List allows to select one or more items at a time.

The List component should be registered with ItemListener or ActionListener using **addItemListener()** or **addActionListener()** method so that they receives the events raised on list.

The following table shows constructors of **List** class.

Constructors	Meaning
List()	Creates a new scrolling list.
List(int rows)	Creates a new scrolling list initialized with the specified number of visible lines.
List(int rows, boolean multipleMode)	Creates a new scrolling list initialized to display the specified number of rows.

The following table shows methods of **List** class.

Methods	Meaning
void **add**(String item)	Adds the specified item to the end of scrolling list.
void **add**(String item, int index)	Adds the specified item to the the scrolling list at the position indicated by the index.
void **addActionListener**(ActionListener l)	Adds the specified action listener to receive action events from this list.
void **addItemListener**(ItemListener l)	Adds the specified item listener to receive item events from this list.
void **addNotify**()	Creates the peer for the list.
void **deselect**(int index)	Deselects the item at the specified index.
AccessibleContext **getAccessibleContext**()	Gets the AccessibleContext associated with this List.
ActionListener[] **getActionListeners**()	Returns an array of all the action listeners registered on this list.

String **getItem**(int index)	Gets the item associated with the specified index.
int **getItemCount**()	Gets the number of items in the list.
ItemListener[] **getItemListeners**()	Returns an array of all the item listeners registered on this list.
String[]**getItems**()	Gets the items in the list.
<T extends EventListener> T[]**getListeners**(Class<T> listenerType)	Returns an array of all the objects currently registered as *Foo*Listeners upon this List.
Dimension **getMinimumSize**()	Determines the minimum size of this scrolling list.
Dimension **getMinimumSize**(int rows)	Gets the minumum dimensions for a list with the specified number of rows.
Dimension **getPreferredSize**()	Gets the preferred size of this scrolling list.
Dimension **getPreferredSize**(int rows)	Gets the preferred dimensions for a list with the specified number of rows.
int **getRows**()	Gets the number of visible lines in this list.
int **getSelectedIndex**()	Gets the index of the selected item on the list,
int[]**getSelectedIndexes**()	Gets the selected indexes on the list.
String **getSelectedItem**()	Gets the selected item on this scrolling list.
String[]**getSelectedItems**()	Gets the selected items on this scrolling list.
Object[]**getSelectedObjects**()	Gets the selected items on this scrolling list in an array of Objects.
int **getVisibleIndex**()	Gets the index of the item that was last made visible by the method makeVisible.
boolean **isIndexSelected**(int index)	Determines if the specified item in this scrolling list is selected.
boolean **isMultipleMode**()	Determines whether this list allows multiple selections.
void **makeVisible**(int index)	Makes the item at the specified index visible.

protected String **paramString**()	Returns the parameter string representing the state of this scrolling list.
protected void **processActionEvent**(ActionEvent e)	Processes action events occurring on this component by dispatching them to any registered ActionListener objects.
protected void **processEvent**(AWTEvent e)	Processes events on this scrolling list.
protected void **processItemEvent**(ItemEvent e)	Processes item events occurring on this list by dispatching them to any registered ItemListener objects.
void **remove**(int position)	Removes the item at the specified position from this scrolling list.
void **remove**(String item)	Removes the first occurrence of an item from the list.
void **removeActionListener**(ActionListener l)	Removes the specified action listener so that it no longer receives action events from this list.
void **removeAll**()	Removes all items from this list.
void **removeItemListener**(ItemListener l)	Removes the specified item listener so that it no longer receives item events from this list.
void **removeNotify**()	Removes the peer for this list.
void **replaceItem**(String newValue, int index)	Replaces the item at the specified index in the scrolling list with the new string.
void **select**(int index)	Selects the item at the specified index in the scrolling list.
Void **setMultipleMode**(boolean b)	Sets the flag that determines whether this list allows multiple selections.

List()

This default constructor of List class creates a new scrolling list which is empty. The list will be created with a default of four rows as its height

List(int rows)

This one argument constructor of List class creates a new scrolling list initialized with the specified number of rows as its height.

List(int rows,boolean multipleMode)

This two arguments constructor of List class creates a new scrolling list initialized to display the specified number of rows as its height. Note that if zero rows are specified, then the list will be created with a default of four rows. Also note that the number of visible rows in the list cannot be changed after it has been created. If the value of multipleMode is true, then the user can select multiple items from the list. If it is false, only one item at a time can be selected.

getItems()

The getItems() method of List class returns items present in the list component as an array of String objects.

public String[] **getItems**()

getSelectedIndexes()

This method of List class returns selected items indexes of list component as an array of integer. This method should be used if multi-selection mode of list component is set to true.

public int[] **getSelectedIndexes**()

getSelectedItems()

This method of List class returns selected items of list component as an array of String objects. This method should be used if multi-selection mode of list component is set to true.

public String[] **getSelectedItems**()

Note: Most of the methods of List class are similar to Choice class which was discussed in choice component.

A program to demonstrate the List component

In this program, a list, a textfield and a button control are created. When the button is clicked the item given in the textfield is added to list control(items should be color names). When a color item is selected in the list box then the selected color is applied to background of applet.

Bin>edit list1.java

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

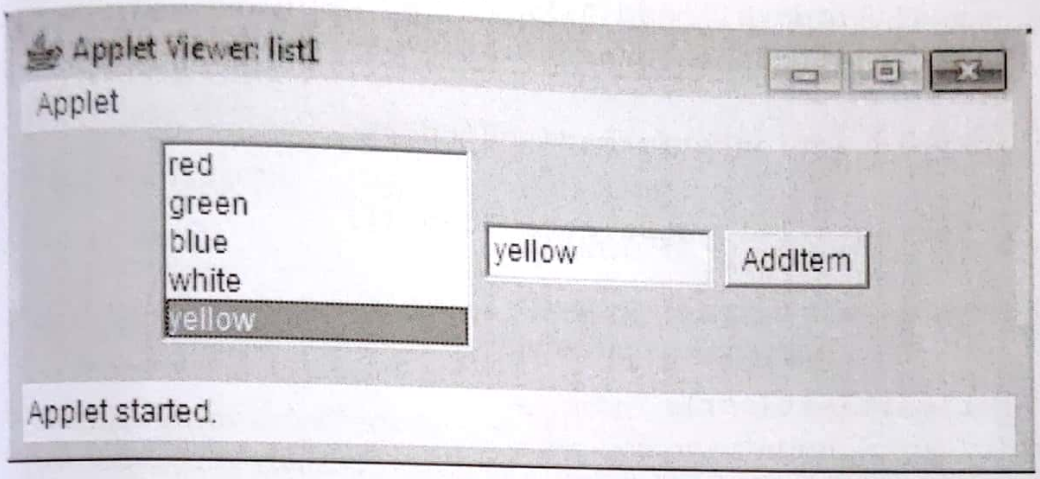
```
public class list1 extends Applet implements
ItemListener, ActionListener
{
    List l;
    TextField t1;
    Button b1;
    public void init()
    {
          l=new List(5);
          l.add("red");
          l.add("green");
          l.add("blue");
          t1=new TextField(10);
          b1=new Button("AddItem");
          add(l);add(t1);add(b1);
          l.addItemListener(this);
          b1.addActionListener(this);
    }
    public void itemStateChanged(ItemEvent i)
    {
          String str=l.getSelectedItem();
          if(str.equals("red"))
                setBackground(Color.red);
          else if(str.equals("green"))
                setBackground(Color.green);
          else if(str.equals("blue"))
                setBackground(Color.blue);
          else if(str.equals("pink"))
                setBackground(Color.pink);
          else if(str.equals("white"))
                setBackground(Color.white);
          else if(str.equals("black"))
                setBackground(Color.black);
          else if(str.equals("yellow"))
                setBackground(Color.yellow);
    }
    public void actionPerformed(ActionEvent e)
    {
          l.add(""+t1.getText());
    }
}
```

Bin>javac list1.java

Bin>copy con list1.html

```
<applet code="list1" width=400 height=400>
</applet>
```

Bin>appletviewer list1.html



Note: You can implement all pre-defined colors present in Color class in the above program.

A program to demonstrate list component that allows to select multiple items.

In this program , a list, textfield and a button controls are created. Select multiple items by clicking on them and click on button. The multiple items selected are shows in textfield.

Bin>edit list2.java

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class list2 extends Applet implements ActionListener
{
    List l;
    TextField t1;
    Button b1;
    public void init()
    {
          l=new List(5,true);
          l.add("red");
```

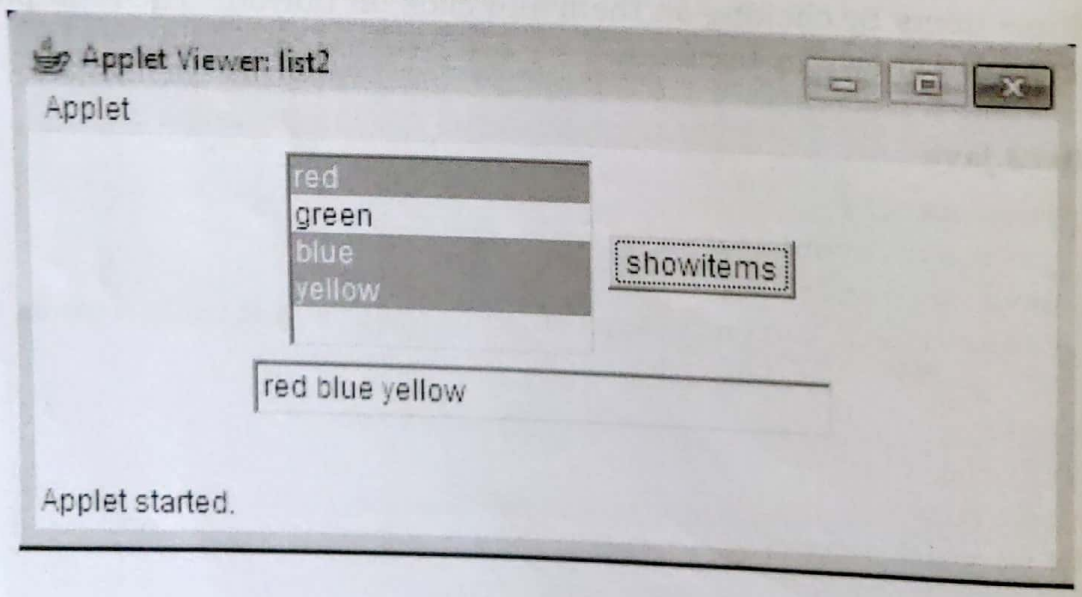
```
            l.add("green");
            l.add("blue");
            l.add("yellow");
            t1=new TextField(30);
            b1=new Button("showitems");
            add(l);add(b1);add(t1);
            b1.addActionListener(this);
      }
      public void actionPerformed(ActionEvent e)
      {
            String s[]=l.getSelectedItems();
            String str="";
            for(int i=0;i<s.length;i++)
                  str=str+ s[i]+" ";
            t1.setText(str);
      }
}
```

Bin>javac list2.java

Bin>copy con list2.html

```
<applet code="list2" width=400 height=400>
</applet>
```

Bin>appletviewer list2.html



Component : Scrollbar

Lists, TextAreas come with ready made scrollbars. However if we want to scroll any other object we have to use a java.awt.Scrollbar. Scrollbars have many uses. At their most basic they are used for moving the visible area. They can also be used to set a value between two numbers. Or they can be used to flip through a number of screens as in a database operation that looks at successive records.

There are three constructors:

Constructors	Meaning
Scrollbar()	Constructs a new vertical scroll bar.
Scrollbar(int orientation)	Constructs a new scroll bar with the specified orientation.
Scrollbar(int orientation, int value, int visible, int min, int max)	Constructs a new scroll bar with the specified orientation, initial value, visible amount, and minimum and maximum values.

The orientation argument is one of the mnemonic constants, **Scrollbar.HORIZONTAL** or **Scrollbar.VERTICAL**. As you expect this determines whether the Scrollbar is laid out from left to right or top to bottom.

A **Scrollbar** has an **int** value at all times. This value will be between the **minimum** and the **maximum** value set by the last two arguments to the constructor. When a **Scrollbar** is created, its value is given by the value argument. The default is **0**.

Finally **visible** represents the size of the visible portion of the scrollable area in pixels. The **Scrollbar** uses this when moving up or down a page.

A **Scrollbar** fires an adjustment event when its value changes. We register an adjustment listener to catch this event. This class needs an **adjustmentValueChanged()** method with this signature:

```
public void adjustmentValueChanged(AdjustmentEvent e)
```

When the user changes the value of the scroll bar, the scroll bar receives an instance of **AdjustmentEvent**. The scroll bar processes this event, passing it along to any registered listeners.

The AdjustmentEvent class defines five types of adjustment event, listed here:

→ **AdjustmentEvent.TRACK** is sent out when the user drags the scroll bar's bubble.

→ **AdjustmentEvent.UNIT_INCREMENT** is sent out when the user clicks in the left arrow of a horizontal scroll bar, or the top arrow of a vertical scroll bar, or makes the equivalent gesture from the keyboard.

→ **AdjustmentEvent.UNIT_DECREMENT** is sent out when the user clicks in the right arrow of a horizontal scroll bar, or the bottom arrow of a vertical scroll bar, or makes the equivalent gesture from the keyboard.

→ **AdjustmentEvent.BLOCK_INCREMENT** is sent out when the user clicks in the track, to the left of the bubble on a horizontal scroll bar, or above the bubble on a vertical scroll bar. By convention, the **Page Up** key is equivalent, if the user is using a keyboard that defines a **Page Up** key.

→ **AdjustmentEvent.BLOCK_DECREMENT** is sent out when the user clicks in the track, to the right of the bubble on a horizontal scroll bar, or below the bubble on a vertical scroll bar. By convention, the **Page Down** key is equivalent, if the user is using a keyboard that defines a **Page Down** key.

The following table shows methods of **Scrollbar** class.

Methods	Meaning
void **addAdjustmentListener**(AdjustmentListener l)	Adds the specified adjustment listener to receive instances of AdjustmentEvent from this scroll bar.
void **addNotify**()	Creates the Scrollbar's peer.
AccessibleContext **getAccessibleContext**()	Gets the AccessibleContext associated with this Scrollbar.
AdjustmentListener[]**getAdjustmentListeners**()	Returns an array of all the adjustment listeners registered on this scrollbar.
int **getBlockIncrement**()	Gets the block increment of this scroll bar.
<T extends EventListener> T[]**getListeners**(Class<T> listenerType)	Returns an array of all the objects currently registered as *Foo*Listeners upon this Scrollbar.
int **getMaximum**()	Gets the maximum value of this scroll bar.
int **getMinimum**()	Gets the minimum value of this scroll bar.
int **getOrientation**()	Returns the orientation of this scroll bar.
int **getUnitIncrement**()	Gets the unit increment for this scrollbar.

int **getValue**()	Gets the current value of this scroll bar.
boolean **getValueIsAdjusting**()	Returns true if the value is in the process of changing as a result of actions being taken by the user.
int **getVisibleAmount**()	Gets the visible amount of this scroll bar.
protected String **paramString**()	Returns a string representing the state of this Scrollbar.
protected void **processAdjustmentEvent**(AdjustmentEvent e)	Processes adjustment events occurring on this scrollbar by dispatching them to any registered AdjustmentListener objects.
protected void **processEvent**(AWTEvent e)	Processes events on this scroll bar.
void **removeAdjustmentListener**(AdjustmentListener l)	Removes the specified adjustment listener so that it no longer receives instances of AdjustmentEvent from this scroll bar.
void **setBlockIncrement**(int v)	Sets the block increment for this scroll bar.
void **setMaximum**(int newMaximum)	Sets the maximum value of this scroll bar.
void **setMinimum**(int newMinimum)	Sets the minimum value of this scroll bar.
void **setOrientation**(int orientation)	Sets the orientation for this scroll bar.
void **setUnitIncrement**(int v)	Sets the unit increment for this scroll bar.
void **setValue**(int newValue)	Sets the value of this scroll bar to the specified value.
void **setValueIsAdjusting**(boolean b)	Sets the valueIsAdjusting property.
void **setValues**(int value, int visible, int minimum, int maximum)	Sets the values of four properties for this scroll bar: value, visibleAmount, minimum, and maximum.
void **setVisibleAmount**(int newAmount)	Sets the visible amount of this scroll bar.

getValue()

This method of Scrollbar class returns the current value of the scroll bar.

public int **getValue**()

setValue()

This method of Scrollbar class can be used to set the value of the scroll bar to the specified value.

public void **setValue**(int Value)

If the value supplied is less than the current minimum or greater than the current maximum - visibleAmount, then either minimum or maximum - visibleAmount is substituted, as appropriate.

getMinimum()

This method of Scrollbar class can be used to get the minimum value of the scroll bar.

public int **getMinimum**()

setMinimum()

This method of Scrollbar class can be used to set the minimum value of the scroll bar.

public void **setMinimum**(int newMinimum)

When setMinimum() is called, the minimum value is changed, and other values (including the maximum, the visible amount, and the current scroll bar value) are changed to be consistent with the new minimum.

getMaximum()

This method of Scrollbar class can be used to get the maximum value of the scroll bar.

public int **getMaximum**()

setMaximum()

This method of Scrollbar class can be used to set the maximum value of the scroll bar.

public void **setMaximum**(int newMaximum)

setUnitIncrement()

This method of Scrollbar can be used to set the unit increment for the scroll bar.

public void **setUnitIncrement**(int v)

getUnitIncrement()

This method of Scrollbar can be used to get the unit increment for the scrollbar.

public int **getUnitIncrement**()

setBlockIncrement()

This method of Scrollbar class can be used to set the block increment for the scroll bar.

public void **setBlockIncrement**(int v)

getBlockIncrement()

This method of Scrollbar class can be used to get the block increment of the scroll bar.

public int **getBlockIncrement**()

setValues()

This method of Scrollbar class can be used to set the values of four properties for the scroll bar: value, visibleAmount, minimum, and maximum.

public void **setValues**(int value,int visible,int minimum,int maximum)

The AdjustmentEvent class

The adjustment event is generated by a scroll bar.

The following table shows fields of **AdjustmentEvent** class

Fields	Meaning
static int**ADJUSTMENT_VALUE_CHANGED**	The adjustment value changed event.
static int **BLOCK_DECREMENT**	The block decrement adjustment type.
static int **BLOCK_INCREMENT**	The block increment adjustment type.
static int **TRACK**	The absolute tracking adjustment type.
static int **UNIT_DECREMENT**	The unit decrement adjustment type.
static int **UNIT_INCREMENT**	The unit increment adjustment type.

The following table shows constructors of **AdjustmentEvent** class.

Constructors	Meaning
AdjustmentEvent(Adjustable source, int id, int type, int value)	Constructs an AdjustmentEvent object with the specified Adjustable source, event type, adjustment type, and value.
AdjustmentEvent(Adjustable source, int id, int type, int value, boolean isAdjusting)	Constructs an AdjustmentEvent object with the specified Adjustable source, event type, adjustment type, and value.

The following table shows methods of **AdjustmentEvent** class.

Methods	Meaning
Adjustable **getAdjustable**()	Returns the Adjustable object where this event originated.
int **getAdjustmentType**()	Returns the type of adjustment which caused the value changed event.
int **getValue**()	Returns the current value in the adjustment event.
boolean **getValueIsAdjusting**()	Returns true if this is one of multiple adjustment events.
String **paramString**()	Returns a string representing the state of this Event.

A program to demonstrate Scrollbar.

In this program, a horizontal scrollbar and a text field components is created. When the scroll bar is scrolled using mouse or keyboard, the scroll value is read and is displayed in text field.

Bin>edit scroll1.java

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class scroll1 extends Applet implements AdjustmentListener
{
 TextField t;
 Scrollbar sb;
 public void init()
```

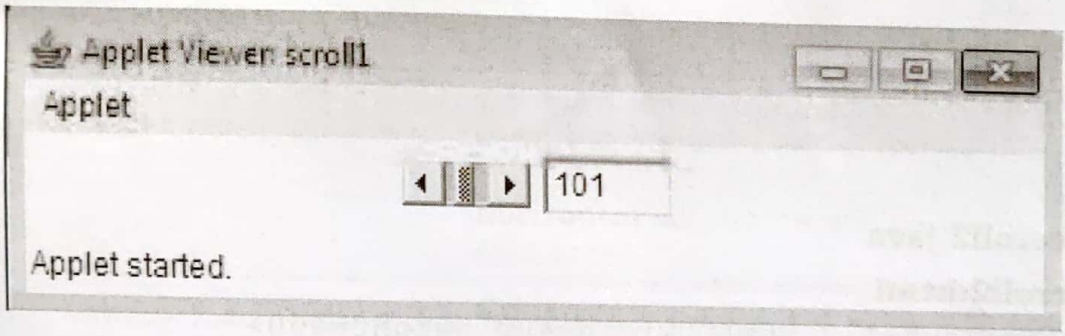
```
{
 int initialValue = 1;
 sb=new Scrollbar(Scrollbar.HORIZONTAL,initialValue,100,0,255);
 sb.addAdjustmentListener(this);
 add(sb);
 t = new TextField(4);
 add(t);
}
public void adjustmentValueChanged(AdjustmentEvent e)
{
   int val = sb.getValue();
   t.setText(""+val);
 }
}
```

Bin>javac scroll1.java

Bin>edit scroll1.html

```
<applet code="scroll1.html" width=400 height=400>
</applet>
```

Bin>appletviewer scroll1.html



A program to create new RGB colors using Scrollbars.

Bin>edit scroll2.java

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class scroll2 extends Applet implements AdjustmentListener
 {
  TextField t1,t2,t3;
  Scrollbar sb1,sb2,sb3;
  public void init()
  {
```

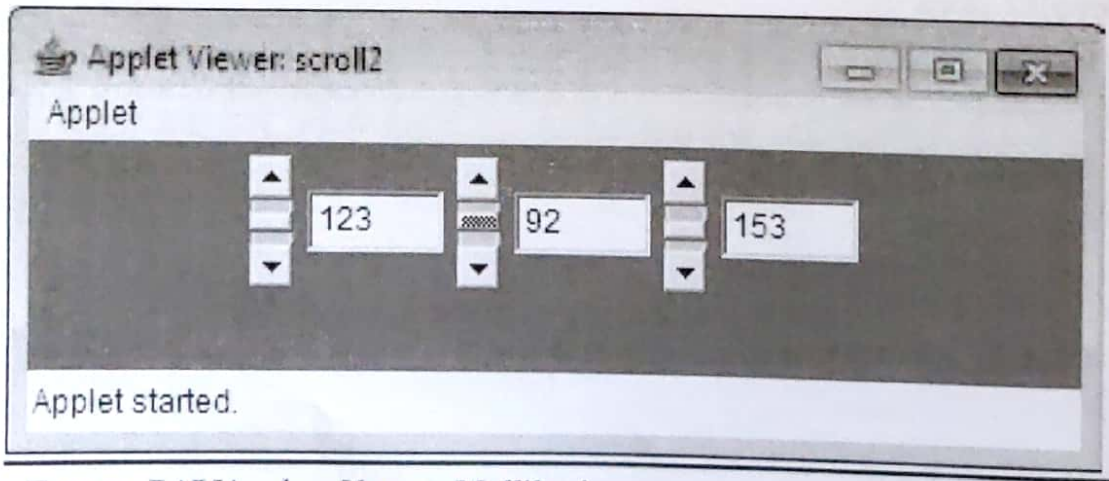
```
    sb1=new Scrollbar(Scrollbar.VERTICAL,0,10,0,255);
    sb2=new Scrollbar(Scrollbar.VERTICAL,0,10,0,255);
    sb3=new Scrollbar(Scrollbar.VERTICAL,0,10,0,255);
    t1 = new TextField(4);
    t2 = new TextField(4);
    t3 = new TextField(4);
    add(sb1);add(t1);
    add(sb2);add(t2);
    add(sb3);add(t3);
    sb1.addAdjustmentListener(this);
    sb2.addAdjustmentListener(this);
    sb3.addAdjustmentListener(this);
  }
  public void adjustmentValueChanged(AdjustmentEvent e)
  {
     int red=sb1.getValue();
     int green=sb2.getValue();
     int blue=sb3.getValue();
     t1.setText(""+red);
     t2.setText(""+green);
     t3.setText(""+blue);
     Color c=new Color(red,green,blue);
     setBackground(c);
  }
}
```

Bin>javac scroll2.java

Bin>edit scroll2.html

```
<applet code="scroll2.html" width=400 height=400>
</applet>
```

Bin>appletviewer scroll2.html



Component : TextArea

TextArea class can be used create text area component which is an extension of TextField component. Text area component is a multiline text field and allows the user to edit the text. We can add scroll bars that allow scrolling the text.

The following table shows fields of **TextArea** class.

Fields	Meaning
static int **SCROLLBARS_BOTH**	Create and display both vertical and horizontal scrollbars.
static int **SCROLLBARS_HORIZONTAL_ONLY**	Create and display horizontal scrollbar only.
static int **SCROLLBARS_NONE**	Do not create or display any scrollbars for the text area.
static int **SCROLLBARS_VERTICAL_ONLY**	Create and display vertical scrollbar only.

The following table shows constructors of **TextArea** class.

Methods	Meaning
TextArea()	Constructs a new text area with the empty string as text. This text area is created with scrollbar visibility equal to SCROLLBARS_BOTH, so both vertical and horizontal scrollbars will be visible for this text area.
TextArea(int rows, int columns)	Constructs a new text area with the specified number of rows and columns and the empty string as text. A column is an approximate average character width that is platform-dependent. The text area is created with scrollbar visibility equal to SCROLLBARS_BOTH, so both vertical and horizontal scrollbars will be visible for this text area.
TextArea(String text)	Constructs a new text area with the specified text. This text area is created with scrollbar visibility equal to SCROLLBARS_BOTH, so both vertical and horizontal scrollbars will be visible for this text area.

TextArea(String text, int rows, int columns)	Constructs a new text area with the specified text, and with the specified number of rows and columns. A column is an approximate average character width that is platform-dependent. The text area is created with scrollbar visibility equal to SCROLLBARS_BOTH, so both vertical and horizontal scrollbars will be visible for this text area.
TextArea(String text, int rows, int columns, int scrollbars)	Constructs a new text area with the specified text, and with the rows, columns, and scroll bar visibility as specified. All TextArea constructors defer to this one. The TextArea class defines several constants that can be supplied as values for the scrollbars argument: SCROLLBARS_BOTH, SCROLLBARS_VERTICAL_ONLY, SCROLLBARS_HORIZONTAL_ONLY, SCROLLBARS_NONE.

The following table shows methods of TextArea class.

Methods	Meaning
void **addNotify**()	Creates the TextArea's peer.
void **append**(String str)	Appends the given text to the text area's current text.
AccessibleContext **getAccessibleContext**()	Returns the AccessibleContext associated with this TextArea.
int **getColumns**()	Returns the number of columns in this text area.
Dimension	Determines the minimum size of this text area.

Dimension **getMinimumSize**(int rows, int columns)	Determines the minimum size of a text area with the specified number of rows and columns.
Dimension **getPreferredSize**()	Determines the preferred size of this text area.
Dimension **getPreferredSize**(int rows, int columns)	Determines the preferred size of a text area with the specified number of rows and columns.
int **getRows**()	Returns the number of rows in the text area.
int **getScrollbarVisibility**()	Returns an enumerated value that indicates which scroll bars the text area uses.
void **insert**(String str, int pos)	Inserts the specified text at the specified position in this text area.
void **replaceRange**(String str, int start, int end)	Replaces text between the indicated start and end positions with the specified replacement text.
Void **setColumns**(int columns)	Sets the number of columns for this text area.
void **setRows**(int rows)	Sets the number of rows for this text area.

insert()

This method of TextArea class can be used to insert the specified text at the specified position in this text area.

public void **insert**(String str,int pos)

append()

This method of TextArea class can be used to append the given text to the text area's current text.

public void **append**(String str)

getRows()

This method of TextArea class can be used to get the number of rows in the text area.

public int **getRows**()

Note:

TextArea class inherits from **TextComponent** class, therefore the methods of **TextComponent** can be used with **TextArea** class object. The methods of **TextComponent** were discussed earlier before **TextField** component.

A program to demonstrate text area component.

Bin>edit textarea1.java

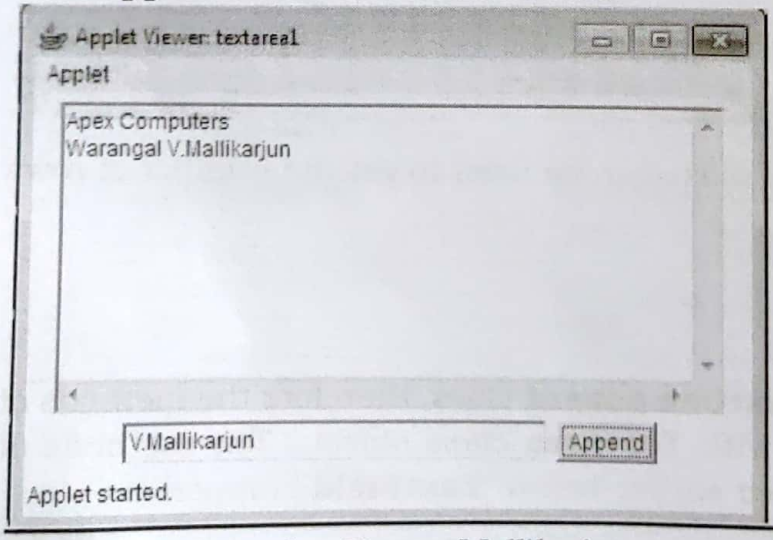
```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class textarea1 extends Applet implements ActionListener
{
    TextArea ta;
    TextField t1;
    Button b1;
    public void init()
    {
        ta=new TextArea("Apex Computers\nWarangal",10,50);
        t1=new TextField(30);
        b1=new Button("Append");
        add(ta);add(t1);add(b1);
        b1.addActionListener(this);
    }
    public void actionPerformed(ActionEvent e)
    {
        String str=t1.getText();
        ta.append(str);
    }
}
```

Bin>javac textarea1.java

Bin>edit textarea1.html

```
<applet code="textarea1" width=400 height=400>
</applet>
```

Bin>appletviewer textarea1.html



Chapter - 20

Layout Managers

Layout Managers

What is a Layout Manager?

A layout manager is an object that controls the size and position (layout) of components inside a Container object. For example, a window is a container that contains components such as buttons and labels. The layout manager in effect for the window determines how the components are sized and positioned inside the window.

A container can contain another container. For example, a window can contain a panel, which is itself a container. As you can see in the figure, the layout manager in effect for the window determines how the two panels are sized and positioned inside the window, and the layout manager in effect for each panel determines how components are sized and positioned inside the panels.

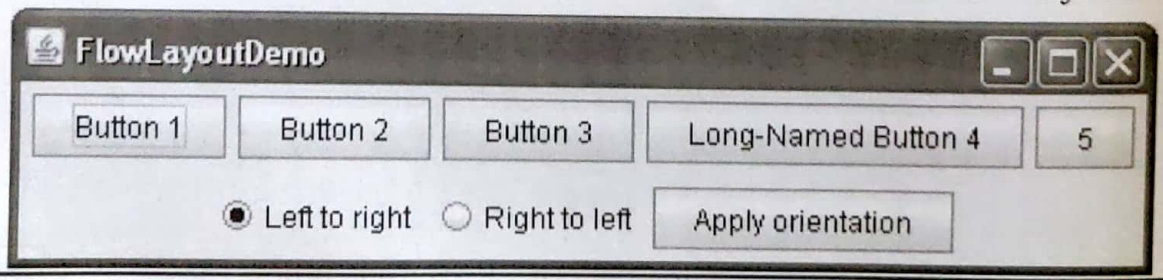
List of Layout Managers

The java.awt package provides the following predefined layout managers that implement the java.awt.LayoutManager interface. Every Abstract Window Toolkit (AWT) has a predefined layout manager as its default. We can change the layout manager of the container using container.setLayout( ) method, and we can define our own layout manager by implementing the java.awt.LayoutManager interface. The following are the predefined layout managers.

FlowLayout
BorderLayout
GridLayout
CardLayout
GridBagLayout

Flowlayout Manager

The FlowLayout class provides a very simple layout manager that is used in default, by the Panel objects(Applet that extends Panel class). The following figure represents a snapshot of an application that uses the flow layout:



The FlowLayout class puts components in a row, sized at their preferred size. If the horizontal space in the container is too small to put all the components in one row, the FlowLayout class uses multiple rows. If the container is wider than necessary for a row of components, then the controls are centered horizontally within the container by default. To specify that the row is to aligned either to the left or right, use a FlowLayout constructor that takes an alignment argument. Another constructor of the FlowLayout class specifies how much vertical or horizontal padding is put around the components.

The following table shows fields present in **FlowLayout** class.

Fields	Meaning
static int **CENTER**	This value indicates that each row of components should be centered.
static int **LEADING**	This value indicates that each row of components should be justified to the leading edge of the container's orientation, for example, to the left in left-to-right orientations.
static int **LEFT**	This value indicates that each row of components should be left-justified.
static int **RIGHT**	This value indicates that each row of components should be right-justified.
static int **TRAILING**	This value indicates that each row of components should be justified to the trailing edge of the container's orientation, for example, to the right in left-to-right orientations.

The following table lists constructors of the **FlowLayout** class.

Constructors	Meaning
FlowLayout()	Constructs a new FlowLayout object with a centered alignment and horizontal and vertical gaps with the default size of 5 pixels.
FlowLayout(int *align*)	Creates a new flow layout manager with the indicated alignment and horizontal and vertical gaps with the default size of 5 pixels. The alignment argument can be FlowLayout.LEFT,FlowLayout.RIGHT,FlowLayout.LEADING, FlowLayout.CENTER, or FlowLayout.TRAILING. When the

	FlowLayout object controls a container with a left-to right component orientation (the default), the LEADING value specifies the components to be left-aligned and the TRAILING value specifies the components to be right-aligned.
FlowLayout (int *align*, int *hgap*, int *vgap*)	Creates a new flow layout manager with the indicated alignment and the indicated horizontal and vertical gaps. The hgap and vgap arguments specify the number of pixels to put between components.

A program to demonstrate how to organize components using FlowLayout manager.

Bin>edit flowlayout1.java

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
public class flowlayout1 extends Applet implements ActionListener
{
    Button b1,b2,b3,b4;
    public void init()
    {
        FlowLayout fl=new FlowLayout(FlowLayout.LEFT,15,15);
        setLayout(fl);
        b1=new Button("Red");
        b2=new Button("Green");
        b3=new Button("Blue");
        b4=new Button("Yellow");
        add(b1);add(b2);add(b3);add(b4);
        b1.addActionListener(this);
        b2.addActionListener(this);
        b3.addActionListener(this);
        b4.addActionListener(this);
    }
    public void actionPerformed(ActionEvent e)
    {
        String str=e.getActionCommand();
        if(str.equals("Red"))
            setBackground(Color.red);
        else if(str.equals("Green"))
            setBackground(Color.green);
```

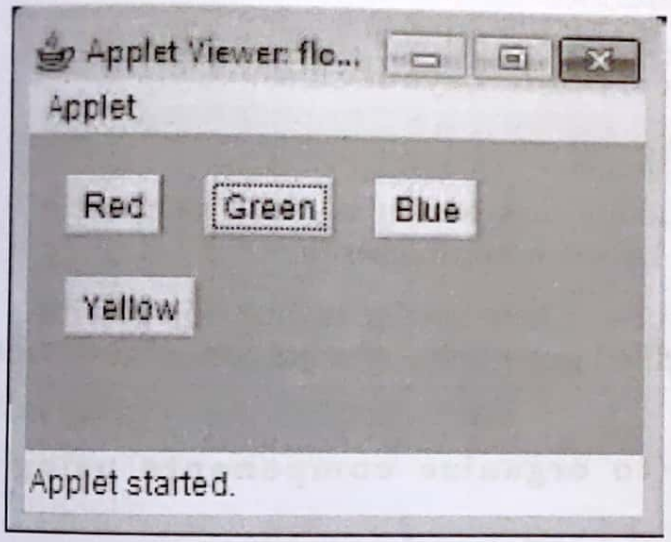
```
            else if(str.equals("BLUE"))
                 setBackground(Color.blue);
            else if(str.equals("Yellow"))
                 setBackground(Color.yellow);
    }
}
```

Bin>javac flowlayout1.java

Bin>edit flowlayout1.html

```
<applet code="flowlayout1" width=400 height=400>
</applet>
```

Bin>appletviewer flowlayout1.html



BorderLayout Manager

The border layout lays out a container, arranging and resizing its components to fit in five regions: north, south, east, west, and center. Each region may contain no more than one component, and is identified by a corresponding constant: NORTH, SOUTH, EAST, WEST, and CENTER. When adding a component to a container with a border layout, use one of these five constants, for example

```
Panel p = new Panel();
p.setLayout(new BorderLayout());
Button b=new Button("Ok");
p.add(b , BorderLayout.SOUTH);
```

The following table shows fields of **BorderLayout** class.

Fields	Meaning
static String **CENTER**	The center layout constraint (middle of container).
static String **EAST**	The east layout constraint (right side of container).
static String **NORTH**	The north layout constraint (top of container).
static String **SOUTH**	The south layout constraint (bottom of container).
static String **WEST**	The west layout constraint (left side of container).

The following table shows constructors of **BorderLayout** class.

Constructors	Meaning
BorderLayout()	Constructs a new border layout object with no gaps between components.
BorderLayout(int hgap, int vgap)	Constructs a new border layout object with the specified gaps between components.

A program to demonstrate how to organize components using BorderLayout manager.

Bin>edit borderlayout1.java

```
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;
public class borderlayout1 extends Applet
implements ActionListener
{
    Button b1,b2,b3,b4;
    TextArea t;
    public void init()
    {
        BorderLayout bl=new BorderLayout();
        setLayout(bl);
        b1=new Button("North");
```

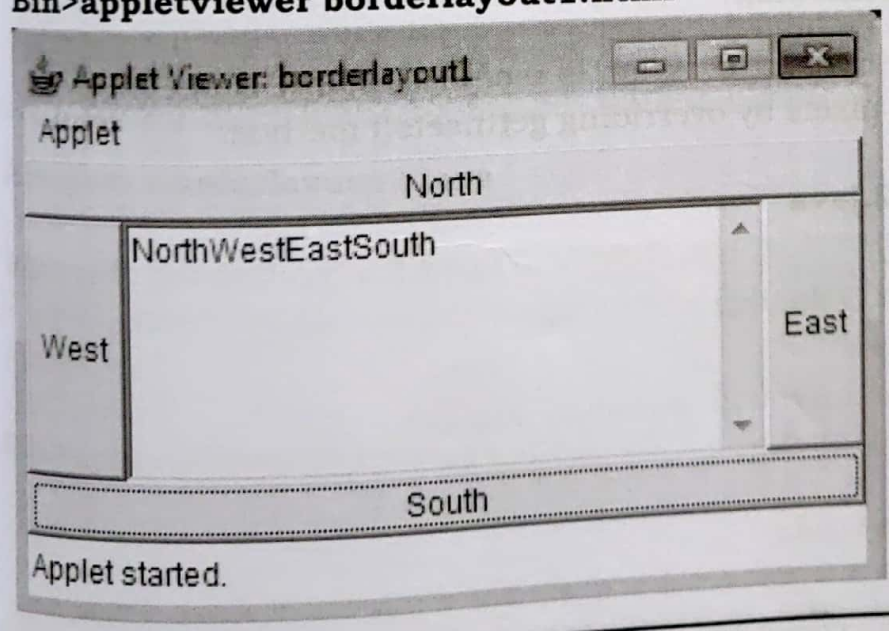
```
        b2=new Button("South");
        b3=new Button("East");
        b4=new Button("West");
        t=new TextArea(20,50);
        add(b1,BorderLayout.NORTH);
        add(b2,BorderLayout.SOUTH);
        add(b3,BorderLayout.EAST);
        add(b4,BorderLayout.WEST);
        add(t,BorderLayout.CENTER);
        b1.addActionListener(this);
        b2.addActionListener(this);
        b3.addActionListener(this);
        b4.addActionListener(this);
    }
    public void actionPerformed(ActionEvent e)
    {
        String str=e.getActionCommand();
        t.append(str);
    }
}
```

Bin>java borderlayout1.java

Bin>edit borderlayout1.html

```
<applet code="borderlayout1" width=400 height=400>
</applet>
```

Bin>appletviewer borderlayout1.html



Class Insets

An Insets object is a representation of the borders of a container. It specifies the space that a container must leave at each of its edges. The space can be a border, a blank space, or a title. The Insets object that contains top, bottom, left, right inset to be used when the container is displayed. These values are used by the layout manager to inset the components when it lays out the window.

The constructor of Insets class is

public **Insets**(int top, int left, int bottom, int right)

The values passed to top, left, bottom, and right specifies the amount of space that should be left between the container and the enclosing window.

getInsets()

The getInsets() method of Container class determines the insets of the container, which indicate the size of the container's border.
A Frame object, for example, has a top inset that corresponds to the height of the frame's title bar.

When we want to leave some space between the container and the window then we should override getInsets(). The getInsets() method returns an Insets object that contains top, bottom, left, right inset to be used when the container is displayed. These values are used by the layout manager to inset the components when it lays out the window.

The signature of the getInsets() method of Container class is:

public Insets getInsets()

The above border layout program is modified to add space from top, left, right, bottom of 10 pixels by overriding getInsets() method.

Bin>edit borderlayout2.java

```
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;
public class borderlayout2 extends Applet
implements ActionListener
{
    Button b1,b2,b3,b4;
    TextArea t;
```

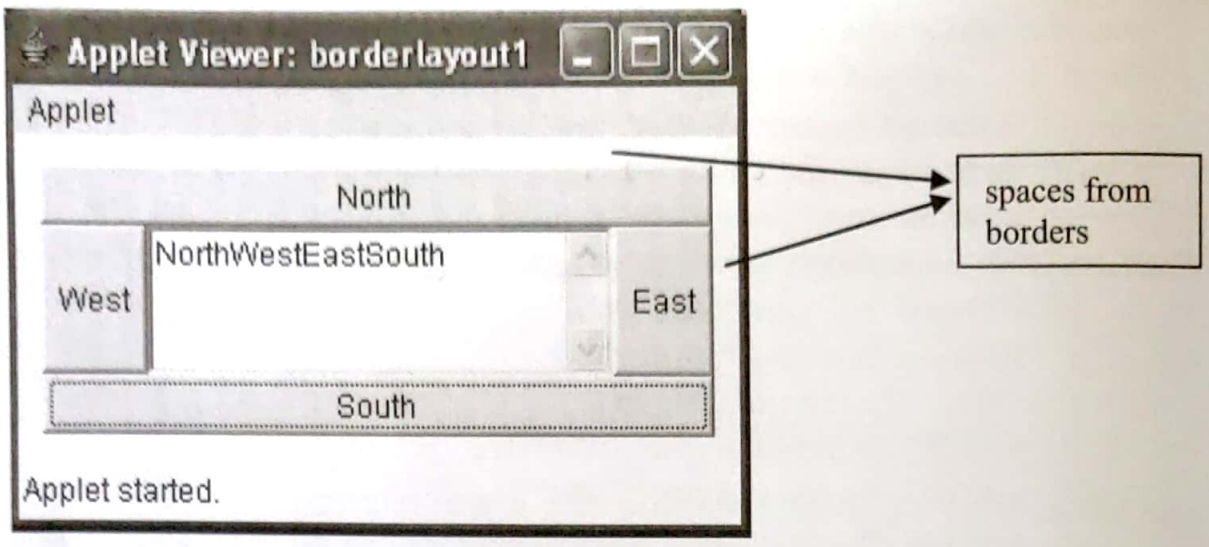
```
    public void init()
    {
        BorderLayout bl=new BorderLayout();
        setLayout(bl);
        b1=new Button("North");
        b2=new Button("South");
        b3=new Button("East");
        b4=new Button("West");
        t=new TextArea(20,50);
        add(b1,BorderLayout.NORTH);
        add(b2,BorderLayout.SOUTH);
        add(b3,BorderLayout.EAST);
        add(b4,BorderLayout.WEST);
        add(t,BorderLayout.CENTER);
        b1.addActionListener(this);
        b2.addActionListener(this);
        b3.addActionListener(this);
        b4.addActionListener(this);
    }
    public void actionPerformed(ActionEvent e)
    {
        String str=e.getActionCommand();
        t.append(str);
    }
    public Insets getInsets()
    {
        return new Insets(10,10,10,10);
    }
}
```

Bin>java borderlayout2.java

Bin>edit borderlayout2.html

<applet code="borderlayout2" width=400 height=400>
</applet>

Bin>appletviewer borderlayout2.html



GridLayout Manager

The GridLayout class is a layout manager that lays out a container's components in a rectangular grid(rows and columns). The container is divided into equal-sized rectangle cells, and one component is placed in each cell.

The following table shows constructors of **GridLayout** class

Constructors	Meaning
GridLayout()	Creates a grid layout with a default of one column per component, in a single row.
GridLayout(int rows, int cols)	Creates a grid layout with the specified number of rows and columns.
GridLayout(int rows, int cols, int hgap, int vgap)	Creates a grid layout with the specified number of rows and columns. The horizontal and vertical gaps are set to the specified values. Horizontal gaps are placed between each of the columns. Vertical gaps are placed between each of the rows.

A program to demonstrate how to organize components using GridLayout manager.

Bin>edit gridlayout1.java

```
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;
public class gridlayout1 extends Applet implements ActionListener
{
   Button b;
```

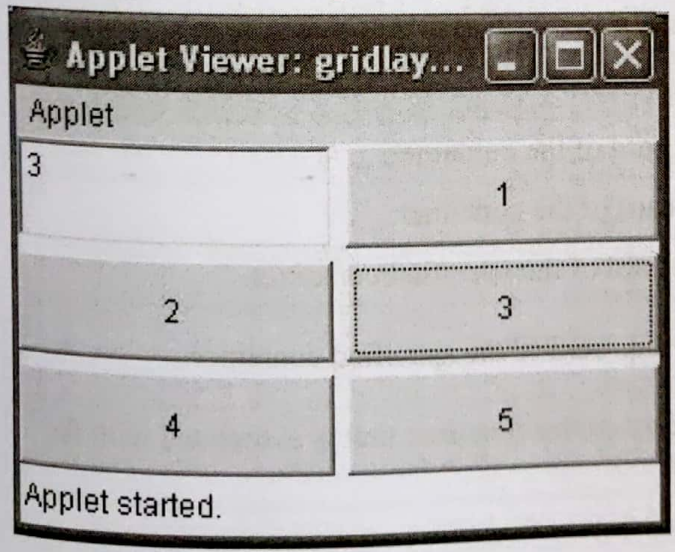
```
    TextField t;
    public void init()
    {
      GridLayout gl=new GridLayout(3,2,5,5);
      setLayout(gl);
      t=new TextField(5);
      add(t);
      for(int i=1;i<=5;i++)
      {
            b=new Button(""+i);
            add(b);
            b.addActionListener(this);
      }
    }
    public void actionPerformed(ActionEvent e)
    {
      String str=e.getActionCommand();
      t.setText(""+str);
    }
}
```

Bin>javac gridlayout1.java

Bin>edit gridlayout1.html

```
<applet code="gridlayout1" width=400 height=400>
</applet>
```

Bin>appletviewer gridlayout1.html



CardLayout Manager

A CardLayout object is a layout manager for a container. CardLayout manager manages the cards present in the container. Each card is a container and it can contain components. Each card can set with different layout manager to organize the components. Each card can be associated with a name. The container acts as a stack of cards. The CardLayout manager, at any given time shows only one card and it can also show any specified card based on the name of the card that is given using show() method.

Cards can be defined using Panel class. Panel is a class defined in java.awt package which acts as a container. We have discussed about this class in the Windows chapter.

The advantage of this layout manager is, in the limited space we can display different group of components. Example of card layout is the properties window of desktop, which contains different tabs such as background, appearance, settings etc. Based on the tab selection the concerned group of components is shown.

The following are the constructors of **CardLayout** class.

Constructors	Meaning
`CardLayout()`	Creates a new card layout with gaps of size zero.
`CardLayout(int hgap, int vgap)`	Creates a new card layout with the specified horizontal and vertical gaps.

The following are the methods of **CardLayout** class.

Methods	Meaning
void **first**(Container parent)	shows first card of the container.
void **last**(Container parent)	shows last card of the container.
void **next**(Container parent)	shows next card of the specified container.
void **previous**(Container parent)	shows previous card of the specified container.
void **show**(Container parent, String name)	shows the card of the container that is associated with the given name.

A program to demonstrate CardLayout manager.

Bin>edit cardlayout1.java

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
public class cardlayout1 extends Applet implements ActionListener
{
    Panel mainpanel;
    CardLayout c1;
    Button b1,b2;
    public void init()
    {
        b1=new Button("subpanel1");
        b2=new Button("subpanel2");
        add(b1);
        add(b2);
        mainpanel=new Panel();
        c1=new CardLayout();
        mainpanel.setLayout(c1);
                    // mainpanel is set with cardlayout
        Panel subpanel1,subpanel2;
        subpanel1=new Panel();
        subpanel2=new Panel();
        subpanel1.setBackground(Color.yellow);
        subpanel2.setBackground(Color.blue);
        TextField t1=new TextField(15);;
        Checkbox c1=new Checkbox("check1",true);
        subpanel1.add(t1);
        subpanel1.add(c1);
        Button b3=new Button("mybutton");
        Label l1=new Label("Label1:");
        subpanel2.add(b3);
        subpanel2.add(l1);
        mainpanel.add(subpanel1,"A");
                    //subpanel1 is added to
                    //mainpanel with the name A
        mainpanel.add(subpanel2,"B");
                    //subpanel2 is added to mainpanel
                    //with the name B
```

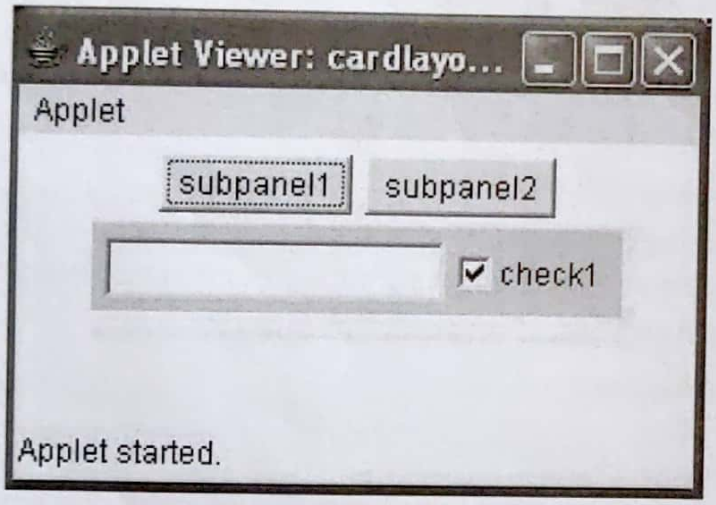
```
            add(mainpanel);
            b1.addActionListener(this);
            b2.addActionListener(this);
      }
      public void actionPerformed(ActionEvent e)
      {
            String str=e.getActionCommand();
            if(str.equals("subpanel1"))
                  cl.show(mainpanel,"A");   //shows subpanel1
            else if(str.equals("subpanel2"))
                  cl.show(mainpanel,"B");   //shows subpanel2
      }
}
```

Bin>javac cardlayout1.java

Bin>edit cardlayout1.html

```
<applet code="cardlayout1" width=400 height=400>
</applet>
```

Bin>appletviewer cardlayout1.html



The Null Layout

The above layout managers can be used to organize the components into predefined manner. The components organized using layout managers' floats in the window as the size of the window changes. If we want to fix the components on the container according to our requirements then we have to make the layout manager as null. This can be done using **setLayout(null)**. After applying null layout we can use **setBounds()** method to position the components on the container.

setBounds()

The **setBounds()** method of Window class can be used to set the position and size of the component in the container.

public void setBounds(int x,int y,int width,int height)

Where x and y specifies position of x and y co-ordinates of the container where the component to be shown. Width and height specifies width and height of the component.

A program that positions the button at 100(x),150(y) co-ordinates of the container.

Bin>edit buttonpos.java

```
import java.awt.*;
import java.awt.event.*;
class mywindow extends Frame implements ActionListener
{
     Label l1,l2,l3;
     TextField t1,t2,t3;
     Button b1,b2;
     public mywindow(String title)
     {
            super(title);
            setLayout(null);
            l1=new Label("Number1 :");
            l2=new Label("Number3 :");
            l3=new Label("Result  :");
            t1=new TextField(10);
            t2=new TextField(10);
            t3=new TextField(10);
            b1=new Button("Add");
            b2=new Button("Sub");
            l1.setBounds(100,50,60,20);
            l2.setBounds(100,80,60,20);
            l3.setBounds(100,110,60,20);
            t1.setBounds(180,50,50,20);
            t2.setBounds(180,80,50,20);
            t3.setBounds(180,110,50,20);
            b1.setBounds(100,140,40,20);
            b2.setBounds(160,140,40,20);

            add(l1);add(t1);
```

```
                add(l2);add(t2);
                add(l3);add(t3);
                add(b1);add(b2);
                b1.addActionListener(this);
                b2.addActionListener(this);
                addWindowListener(new mywindowhandler());
        }
        public void actionPerformed(ActionEvent e)
        {
                int a=0,b=0,c=0;
                a=Integer.parseInt(t1.getText());
                b=Integer.parseInt(t2.getText());
                String str=e.getActionCommand();
                if(str.equals("Add"))
                        c=a+b;
                else if(str.equals("Sub"))
                        c=a-b;
                t3.setText(""+c);
        }
        class mywindowhandler extends WindowAdapter
        {
                public void windowClosing(WindowEvent e)
                {
                        dispose();
                }
        }
}

class buttonpos
{
        public static void main(String argv[])
        {
                mywindow w=new mywindow("Frame Window");
                w.setSize(400,400);
                w.setVisible(true);
        }
}
```

Bin>javac buttonpos.java

Bin>java buttonpos