CHAPTER-12

Abstract Window Toolkit

Java AWT

Java AWT (Abstract Windowing Toolkit) is an API to develop GUI or window-based application in java.

Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system. AWT is heavyweight i.e. its components uses the resources of system.

AWT Packages

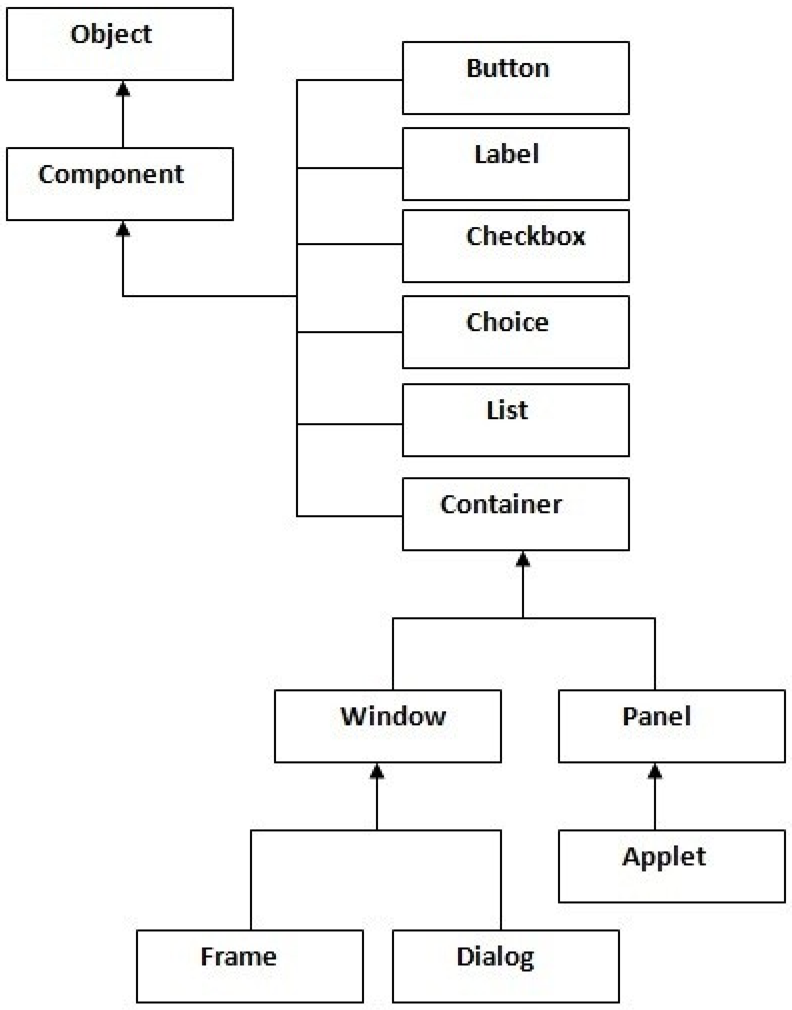
AWT is huge! It consists of 12 packages (Swing is even bigger, with 18 packages as of JDK 1.7!). Fortunately, only 2 packages - java.awt and java.awt.event - are commonly-used.

1. The java.awt package contains the *core* AWT graphics classes:
   - GUI Component classes (such as Button, TextField, and Label),
   - GUI Container classes (such as Frame, Panel, Dialog and ScrollPane),
   - Layout managers (such as FlowLayout, BorderLayout and GridLayout & CardLayout),
   - Custom graphics classes (such as Graphics, Color and Font).
2. The java.awt.event package supports event handling:
   - Event classes (such as ActionEvent, MouseEvent, KeyEvent and WindowEvent),
   - Event Listener Interfaces (such as ActionListener, MouseListener, KeyListener and WindowListener),
   - Event Listener Adapter classes (such as MouseAdapter, KeyAdapter, and WindowAdapter).

AWT provides a *platform-dependent* and *device-independent* interface to develop graphic programs that runs on all platforms, such as Windows, Mac, and Linux.

Java AWT Hierarchy

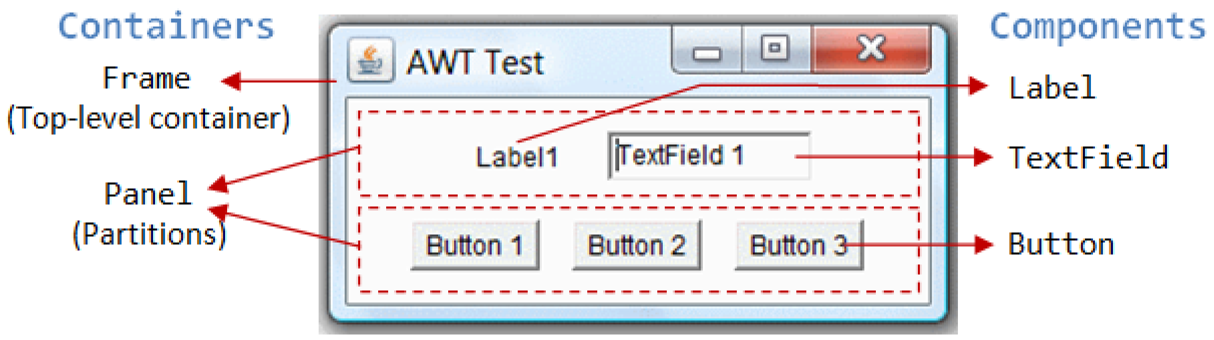
The hierarchy of Java AWT classes are given below.



Containers and Components

There are two types of GUI elements:

1. Container: Containers (such as Frame, Panel and Applet) are used to *hold components in a specific layout* (such as flow or grid). A container can also hold sub-containers.
2. Component: Components are elementary GUI entities (such as Button, Label, and TextField.)

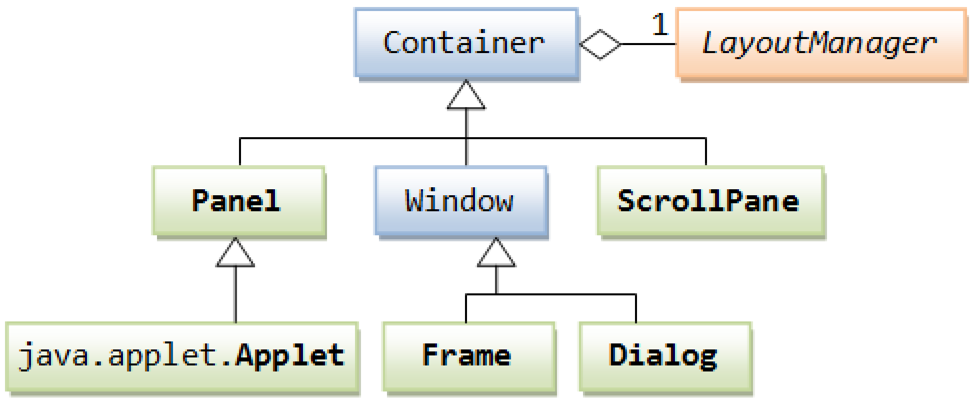


In the above figure, there are three containers: a Frame and two Panels. A Frame is the *top-level container* of an AWT program. A Frame has a title bar (containing an icon, a title, and the minimize/maximize/close buttons), an optional menu bar and the content display area. A Panel is a *rectangular area* used to group related GUI components in a certain layout. In the above figure, the top-level Frame contains two Panels. There are five components: a Label (providing description), a TextField (for users to enter text), and three Buttons (for user to trigger certain programmed actions).

Container

The Container is a component in AWT that can contain another components like buttons, textfields, labels etc. The classes that extends Container class are known as container. The AWT provides four container classes. They are class Window and its two subtypes -- class Frame and class Dialog -- as well as the Panel class. In addition to the containers provided by the AWT, the Applet class is a container -- it is a subtype of the Panel class and can therefore hold components. Brief descriptions of each container class provided by the AWT are provided below.Window

The hierarchy of the AWT Container classes is as follows:



Window

A top-level display surface (a window). An instance of the Window class is not attached to nor embedded within another container. An instance of the Window class has no border and no title. You must use frame, dialog or another window for creating a window.

Panel

The Panel is the container that doesn't contain title bar and menu bars. It can have other components like button, textfield etc.

Frame

The Frame is the container that contain title bar and can have menu bars. It can have other components like button, textfield etc.

Dialog

A generic container for holding components. An instance of the Panel class provides a container to which to add components.

Useful Methods of Component class

Method	Description
public void add(Component c)	inserts a component on this component.
public void setSize(int width,int height)	sets the size (width and height) of the component.
public void setLayout(LayoutManager m)	defines the layout manager for the component.
public void setVisible(boolean status)	changes the visibility of the component, by default false.

Creating Container and Adding GUI Components

To create simple awt example, you need a frame. There are two ways to create a frame in AWT.

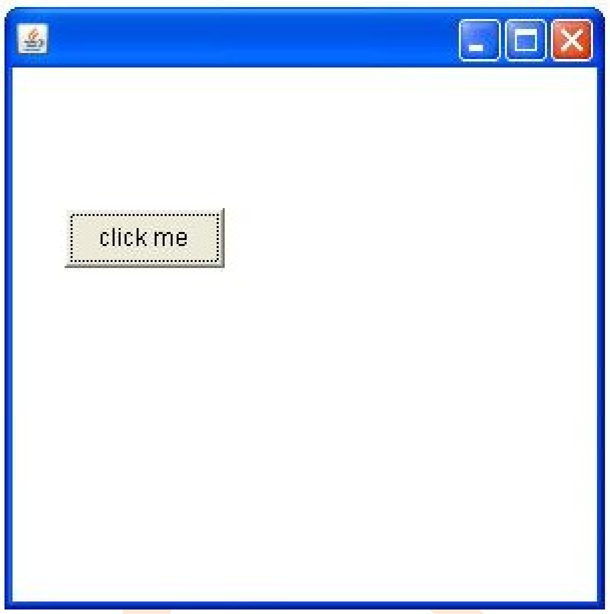
- By extending Frame class (inheritance)
- By creating the object of Frame class (association)

Simple example of AWT by inheritance

```
import java.awt.*;
class First extends Frame
{
   First()
   {
      Button b=new Button("click me");
      b.setBounds(30,100,80,30);// setting button position

      add(b);//adding button into frame
      setSize(300,300);//frame size 300 width and 300 height
      FlowLayout l = new FlowLayout();
      setLayout(l);//no layout manager
      setVisible(true);//now frame will be visible, by default not visible
```

```
        }
        public static void main(String args[])
        {
            First f=new First();
        }
    }
```

The setBounds(int xaxis, int yaxis, int width, int height) method is used in the above example that sets the position of the awt button.

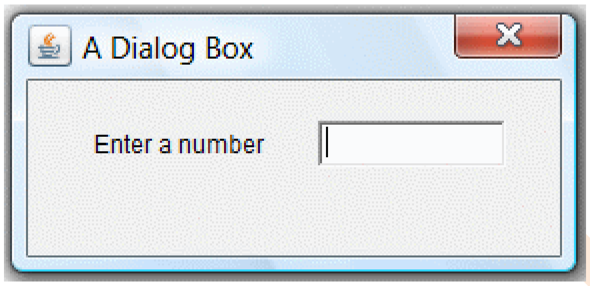


Simple example of AWT by association

```
    import java.awt.*;
    class First2
    {
        First2()
        {
            Frame f=new Frame();
            Button b=new Button("click me");
            b.setBounds(30,50,80,30);

            f.add(b);
            f.setSize(300,300);
            f.setLayout(null);
            f.setVisible(true);
        }
```

```
            public static void main(String args[])
            {
               First2 f=new First2();
            }
        }
```

An AWT Dialog is a *"pop-up window"* used for interacting with the users. A Dialog has a title-bar (containing an icon, a title and a close button) and a content display area, as illustrated.



 An AWT Applet (in package java.applet) is the top-level container for an applet, which is a Java program running inside a browser. Applet will be already discussed in the previous chapter.

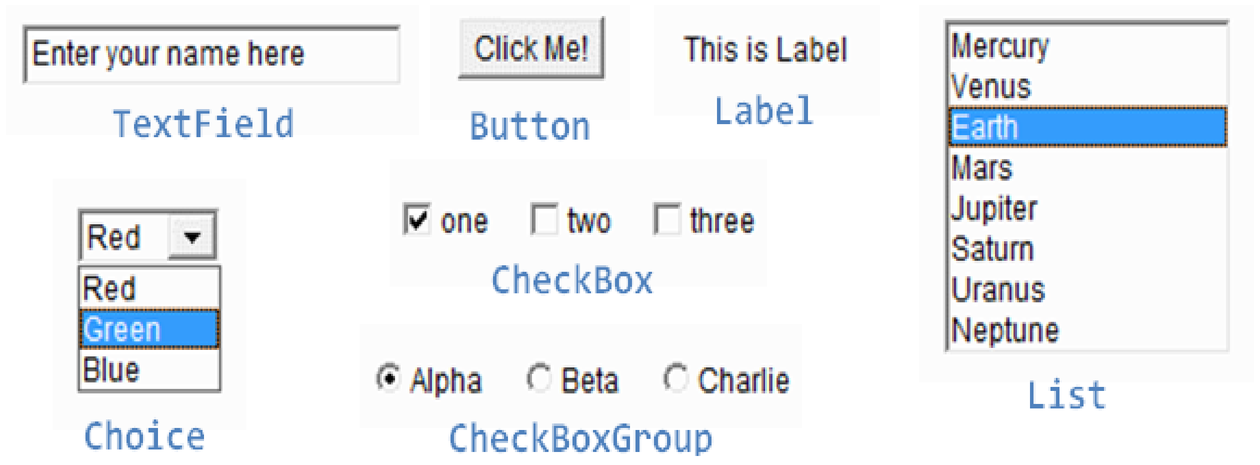
Secondary Containers: Panel and ScrollPane

Secondary containers are placed inside a top-level container or another secondary container. AWT also provide these secondary containers:

- Panel: a rectangular box under a higher-level container, used to *layout* a set of related GUI components in pattern such as grid or flow.
- ScrollPane: provides automatic horizontal and/or vertical scrolling for a single child component.

AWT Component Classes

AWT provides many ready-made and reusable GUI components. The frequently-used are: Button, TextField, Label, Checkbox, CheckboxGroup (radio buttons), List, and Choice, as illustrated below.

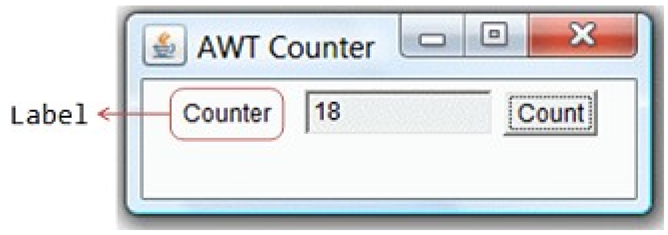


AWT UI Elements:

Following is the list of commonly used controls while designed GUI using AWT.

Sr. No.	Control & Description
1	Label A Label object is a component for placing text in a container.
2	Button This class creates a labeled button.
3	Check Box A check box is a graphical component that can be in either an **on** (true) or **off** (false) state.
4	Check Box Group The CheckboxGroup class is used to group the set of checkbox.
5	List The List component presents the user with a scrolling list of text items.
6	Text Field A TextField object is a text component that allows for the editing of a single

	line of text.
7	Text Area A TextArea object is a text component that allows for the editing of a multiple lines of text.
8	Choice A Choice control is used to show pop up menu of choices. Selected choice is shown on the top of the menu.
9	Canvas A Canvas control represents a rectangular area where application can draw something or can receive inputs created by user.
10	Image An Image control is superclass for all image classes representing graphical images.
11	Scroll Bar A Scrollbar control represents a scroll bar component in order to enable user to select from range of values.
12	Dialog A Dialog control represents a top-level window with a title and a border used to take some form of input from the user.
13	File Dialog A FileDialog control represents a dialog window from which the user can select a file.



java.awt.Label

A java.awt.Label provides a text description message. Take note that System.out.println() prints to the system console, not to the graphics screen. You could use a Label to label another component (such as text field) or provide a text description.

Check the JDK API specification for java.awt.Label.

Constructors

public Label(String *strLabel*, int *alignment*);

The first constructor constructs a Label object with the given text string in the given alignment. Note that three static constants Label.LEFT, Label.RIGHT, and Label.CENTER are defined in the class for you to specify the alignment (rather than asking you to memorize arbitrary integer values).

public Label(String *strLabel*);

The second constructor constructs a Label object with the given text string in default of left-aligned.

public Label();

The third constructor constructs a Label object with an initially empty string. You could set the label text via the setText() method later.

Constants

public static final LEFT;   // Label.LEFT
public static final RIGHT;   // Label.RIGHT
public static final CENTER; // Label.CENTER

These three constants are defined for specifying the alignment of the Label's text.

Public Methods

```
// Examples
public String getText();
public void setText(String strLabel);
public int getAlignment();
public void setAlignment(int alignment);
```

The getText() and setText() methods can be used to read and modify the Label's text. Similarly, the getAlignment() and setAlignment() methods can be used to retrieve and modify the alignment of the text.

Constructing a Component and Adding the Component into a Container

Three steps are necessary to create and place a GUI component:

1. Declare the component with an *identifier* (*name*);
2. Construct the component by invoking an appropriate constructor via the new operator;
3. Identify the container (such as Frame or Panel) designed to hold this component. The container can then add this component onto itself via *aContainer*.add(*aComponent*) method. Every container has a add(Component) method. Take note that it is the container that actively and explicitly adds a component onto itself, instead of the other way.

Example

```
// Declare an Label instance called
Label lblInput;
// Construct by invoking a constructor via the new operator
 lblInput = new Label("Enter ID");

// add(lblInput)- "this" is typically a subclass of Frame
add(lblInput);

// Modify the Label's text string
lblInput.setText("Enter password");

// Retrieve the Label's text string
```

lblInput.getText();

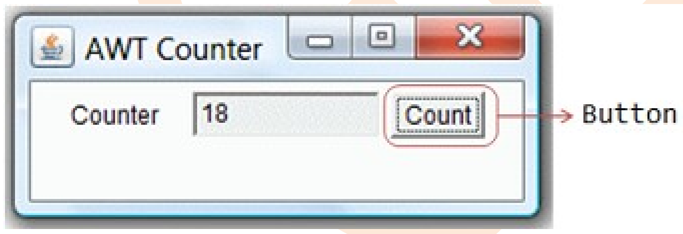
An Anonymous Instance

You can create a Label without specifying an identifier, called *anonymous instance*. In the case, the Java compiler will assign an *anonymous identifier* for the allocated object. You will not be able to reference an anonymous instance in your program after it is created. This is usually alright for a Label instance as there is often no need to reference a Label after it is constructed.

Example

```
// Allocate an anonymous Label instance. "this" container adds the instance into
itself.
//You CANNOT reference an anonymous instance to carry out further operations.
add(new Label("Enter Name: ", Label.RIGHT));

// Same as
Label lblXxx = new Label("Enter Name: ", Label.RIGHT));
// lblXxx assigned by compiler add(lblXxx);
```

java.awt.Button



A java.awt.Button is a GUI component that triggers a certain programmed *action* upon clicking.

Constructors

```
public Button(String buttonLabel);
  // Construct a Button with the given label
public Button();
  // Construct a Button with empty label
```

The Button class has two constructors. The first constructor creates a Button object with the given label painted over the button. The second constructor creates a Button object with no label.

Public Methods

```
public String getLabel();
// Get the label of this Button instance
public void setLabel(String buttonLabel);
// Set the label of this Button instance
public void setEnabled(boolean enable);
// Enable or disable this Button. Disabled Button cannot be clicked.
```

The getLabel() and setLabel() methods can be used to read the current label and modify the label of a button, respectively.

Note: The latest Swing's JButton replaces getLabel()/setLabel() with getText()/setText() to be consistent with all the components. We will describe Swing later.

Event

Clicking a button fires a so-called *ActionEvent* and triggers a certain programmed action. I will explain event-handling later.

Example

```
Button btnColor = new Button("Red"); // Declare and allocate a
                                       Button instance called btnColor
add(btnColor);                       // "this" Container adds the Button
...
Button btnColor = new Button();
btnColor.setLabel("green");        // Change the button's label
String x = btnColor.getLabel();              // Read the button's label
...
add(new Button("Blue"));// Create an anonymous Button. It CANNOT be
referenced later
```

Useful Methods of Component class

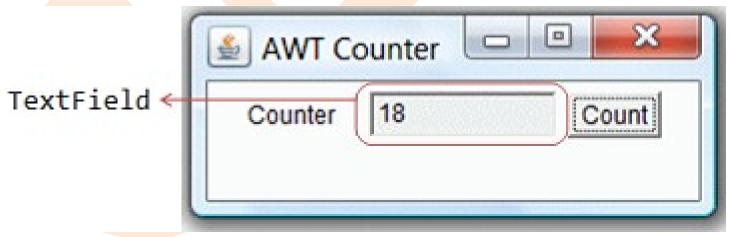
Method	Description

public void add(Component c)	inserts a component on this component.
public void setSize(int width,int height)	sets the size (width and height) of the component.
public void setLayout(LayoutManager m)	defines the layout manager for the component.
public void setVisible(boolean status)	changes the visibility of the component, by default false.

In a GUI program, a component must be kept in a container. You need to identify a container to hold the components. Every container has a method called add(Component c). A container (say aContainer) can invoke aContainer.add(aComponent) to add aComponent into itself. For example,

```
Panel panel = new Panel();        // Panel is a Container
Button btn = new Button("Press"); // Button is a Component
panel.add(btn);                   // The Panel Container adds a Button Component
```

java.awt.TextField



A java.awt.TextField is single-line text box for users to enter texts. (There is a multiple-line text box called TextArea.) Hitting the "ENTER" key on a TextField object triggers an action-event.

Constructors

public TextField(String *strInitialText*, int *columns*);

// Construct a TextField instance with the given initial text string with the number of columns.
public TextField(String *strInitialText*);
// Construct a TextField instance with the given initial text string.
public TextField(int *columns*);
// Construct a TextField instance with the number of columns.

Public Methods

public String getText();
// Get the current text on this TextField instance
public void setText(String *strText*);
// Set the display text on this TextField instance
public void setEditable(boolean *editable*);
// Set this TextField to editable (read/write) or non-editable (read-only)

Event

Hitting the "ENTER" key on a TextField fires a *ActionEvent*, and triggers a certain programmed action.

Example

```
TextField tfInput = new TextField(30); // Declare and allocate an TextField
instance called
                                                        tfInput
add(tfInput);                          // "this" Container adds the TextField
TextField tfResult = new TextField();  // Declare and allocate an TextField
instance called
                                                      tfResult
tfResult.setEditable(false) ;                  // Set to read-only
add(tfResult);                       // "this" Container adds the TextField
......
// Read an int from TextField "tfInput", square it, and display on "tfResult".
// getText() returns a String, need to convert to int

int number = Integer.parseInt(tfInput.getText());
number *= number;

// setText() requires a String, need to convert the int number to String.
```

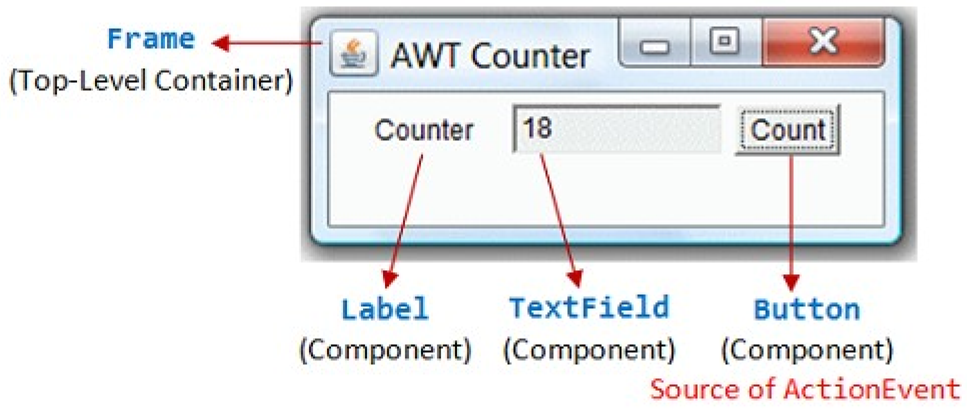
tfResult.setText(number + "");

Take note that getText()/SetText() operates on String. You can convert a String to a primitive, such as int or double via static method Integer.parseInt() or Double.parseDouble(). To convert a primitive to a String, simply concatenate the primitive with an empty String.

Example: AWTCounter

Let's assemble some components together into a simple GUI counter program, as illustrated. It has a top-level container Frame, which contains three components - a Label "Counter", a non-editable TextField to display the current count, and a "Count" Button. The TextField displays "0" initially.

Each time you click the button, the counter's value increases by 1.



```
import java.awt.*;  // Using AWT container and component classes
import java.awt.event.*;// Using AWT event classes and listener interfaces

// An AWT program inherits from the top-level container java.awt.Frame
public class AWTCounter extends Frame implements ActionListener {
  private Label lblCount;    // Declare component Label
  private TextField tfCount; // Declare component TextField
  private Button btnCount;   // Declare component Button
  private int count = 0;     // Counter's value

  /** Constructor to setup GUI components and event handling */
  public AWTCounter () {
    FlowLayout fl = new FlowLayoutt()
    setLayout(fl);
      // "super" Frame sets its layout to FlowLayout, which arranges the
components
      //  from left-to-right, and flow to next row from top-to-bottom.

    lblCount = new Label("Counter");  // construct Label
    add(lblCount);                    // "super" Frame adds Label

    tfCount = new TextField("0", 10); // construct TextField
    tfCount.setEditable(false);       // set to read-only
    add(tfCount);                     // "super" Frame adds tfCount

    btnCount = new Button("Count");   // construct Button
    add(btnCount);                    // "super" Frame adds Button

    btnCount.addActionListener(this);
      // Clicking Button source fires ActionEvent
      // btnCount registers this instance as ActionEvent listener


    setTitle("AWT Counter");  // "super" Frame sets title
    setSize(250, 100);        // "super" Frame sets initial window size


    setVisible(true);         // "super" Frame shows

  }

  /** The entry main() method */
  public static void main(String[] args) {
    // Invoke the constructor to setup the GUI, by allocating an instance
    AWTCounter app = new AWTCounter();
  }

  /** ActionEvent handler - Called back upon button-click. */
  @Override
  public void actionPerformed(ActionEvent evt)
  {
    ++count; // increase the counter value
    // Display the counter value on the TextField tfCount
    tfCount.setText(count + ""); // convert int to String
  }
}
```

CheckBox Class

Checkbox control is used to turn an option on(true) or off(false). There is label for each checkbox representing what the checkbox does.The state of a checkbox can be changed by clicking on it.

Class constructors

S.N.	Constructor & Description
1	Checkbox() Creates a check box with an empty string for its label.
2	Checkbox(String label) Creates a check box with the specified label.
3	Checkbox(String label, boolean state) Creates a check box with the specified label and sets the specified state.
4	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.
5	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.

Class methods

S.N.	Method & Description
1	void addItemListener(ItemListener l) Adds the specified item listener to receive item events from this check box.

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

Example of CheckBox

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);
}
}
```

CheckboxGroup [Radio Button]

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:

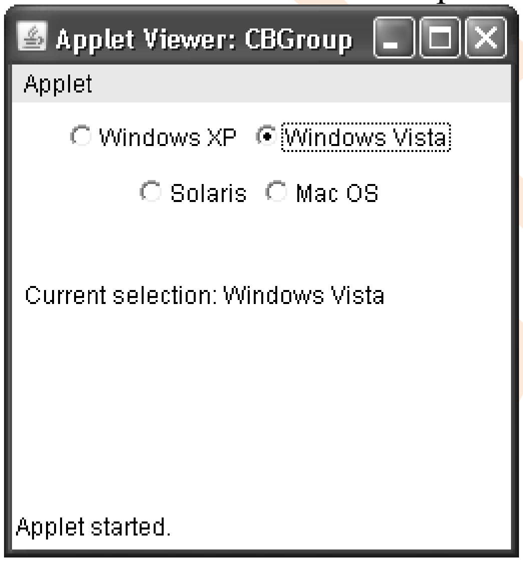
Checkbox getSelectedCheckbox( )
void setSelectedCheckbox(Checkbox *which*)

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);
}

}
```

Output generated by the **CBGroup** applet is shown in Figure 24-3. Notice that the check
boxes are now circular in shape.



```
Current selection: Windows Vista
Applet started.
```

AWT Choice Class

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 stringClass

Constructors

S.N.	Constructor & Description
1	**Choice()** Creates a new choice menu.

Class methods

S.N.	Method & Description
1	**void add(String item)** Adds an item to this Choice menu.
2	**void addItem(String item)** Obsolete as of Java 2 platform v1.1.
3	**void addItemListener(ItemListener l)** Adds the specified item listener to receive item events from this Choice menu.
4	**int countItems()** Deprecated. As of JDK version 1.1, replaced by getItemCount().
5	**String getItem(int index)** Gets the string at the specified index in this Choice menu.
6	**int getItemCount()** Returns the number of items in this Choice menu.
7	**ItemListener[] getItemListeners()** Returns an array of all the item listeners registered on this choice.
8	**int getSelectedIndex()** Returns the index of the currently selected item.
9	**String getSelectedItem()** Gets a representation of the current choice as a string.
10	**Object[] getSelectedObjects()** Returns an array (length 1) containing the currently selected item.
11	**void insert(String item, int index)** Inserts the item into this choice at the specified position.
12	**void remove(int position)**

	Removes an item from the choice menu at the specified position.
13	**void remove(String item)** Removes the first occurrence of item from the Choice menu.
14	**void removeAll()** Removes all items from the choice menu.
15	**void select(int pos)** Sets the selected item in this Choice menu to be the item at the specified position.
16	**void select(String str)** Sets the selected item in this Choice menu to be the item whose name is equal to the specified string.

Choice Example

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

public class AwtControlDemo {

  private Frame mainFrame;
  private Label headerLabel;
  private Label statusLabel;
  private Panel controlPanel;

  public AwtControlDemo(){
    prepareGUI();
  }

  public static void main(String[] args){
    AwtControlDemo  awtControlDemo = new AwtControlDemo();
    awtControlDemo.showChoiceDemo();
  }

  private void prepareGUI(){
```

```
  mainFrame = new Frame("Java AWT Examples");
  mainFrame.setSize(400,400);
  mainFrame.setLayout(new GridLayout(3, 1));
  mainFrame.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent windowEvent){
      System.exit(0);
    }
  });
  headerLabel = new Label();
  headerLabel.setAlignment(Label.CENTER);
  statusLabel = new Label();
  statusLabel.setAlignment(Label.CENTER);
  statusLabel.setSize(350,100);

  controlPanel = new Panel();
  controlPanel.setLayout(new FlowLayout());

  mainFrame.add(headerLabel);
  mainFrame.add(controlPanel);
  mainFrame.add(statusLabel);
  mainFrame.setVisible(true);
}

private void showChoiceDemo(){

  headerLabel.setText("Control in action: Choice");
  final Choice fruitChoice = new Choice();

  fruitChoice.add("Apple");
  fruitChoice.add("Grapes");
  fruitChoice.add("Mango");
  fruitChoice.add("Peer");

  Button showButton = new Button("Show");

  showButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      String data = "Fruit Selected: "
      + fruitChoice.getItem(fruitChoice.getSelectedIndex());
      statusLabel.setText(data);
```

```
         }
      });

      controlPanel.add(fruitChoice);
      controlPanel.add(showButton);

      mainFrame.setVisible(true);
   }
}
```

List Class

The List represents a list of text items. The list can be configured to that user can choose either one item or multiple items

Class constructors

S.N.	Constructor & Description
1	**List()** Creates a new scrolling list.
2	**List(int rows)** Creates a new scrolling list initialized with the specified number of visible lines.
3	**List(int rows, boolean multipleMode)** Creates a new scrolling list initialized to display the specified number of rows.

Class methods

<T extends EventListener> T[] getListeners(Class<T> listenerType)
Returns an array of all the objects currently registered as FooListeners upon this List.

S.N.	Method & Description
1	**void add(String item)**

	Adds the specified item to the end of scrolling list.
2	**void add(String item, int index)** Adds the specified item to the the scrolling list at the position indicated by the index.
3	**void addActionListener(ActionListener l)** Adds the specified action listener to receive action events from this list.
4	**void addItemListener(ItemListener l)** Adds the specified item listener to receive item events from this list.
5	**void deselect(int index)** Deselects the item at the specified index.
6	**String getItem(int index)** Gets the item associated with the specified index.
7	**int getItemCount()** Gets the number of items in the list.
8	**String[] getItems()** Gets the items in the list.
10	**int getRows()** Gets the number of visible lines in this list.
11	**int getSelectedIndex()** Gets the index of the selected item on the list,
12	**int[] getSelectedIndexes()** Gets the selected indexes on the list.
13	**String getSelectedItem()** Gets the selected item on this scrolling list.
14	**String[] getSelectedItems()**

	Gets the selected items on this scrolling list.
15	**boolean isIndexSelected(int index)** Determines if the specified item in this scrolling list is selected.
16	**boolean isMultipleMode()** Determines whether this list allows multiple selections.
17	**boolean isSelected(int index)** Deprecated. As of JDK version 1.1, replaced by isIndexSelected(int).
18	**void makeVisible(int index)** Makes the item at the specified index visible.
19	**void remove(int position)** Removes the item at the specified position from this scrolling list.
20	**void remove(String item)** Removes the first occurrence of an item from the list.
21	**void removeActionListener(ActionListener l)** Removes the specified action listener so that it no longer receives action events from this list.
22	**void removeAll()** Removes all items from this list.
23	**void replaceItem(String newValue, int index)** Replaces the item at the specified index in the scrolling list with the new string.
24	**void select(int index)** Selects the item at the specified index in the scrolling list.
25	**void setMultipleMode(boolean b)** Sets the flag that determines whether this list allows multiple selections.

Using a 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 adds the following methods:

void append(String *str*)
void insert(String *str*, int *index*)

void replaceRange(String str, int startIndex, int endIndex)

```
// 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);
}}
```

Layout Managers

A container has a so-called *layout manager* to arrange its components. The layout managers provide a level of abstraction to map your user interface on all windowing systems, so that the layout can be *platform-independent*.

The java.awt package provides five layout manager classes

Name	Description
BorderLayout	Arranges components along the sides of the container and in the middle.
CardLayout	Arrange components in "cards" Only one card is visible at a time.
FlowLayout	Arranges components in variable-length rows.
GridBagLayout	Aligns components horizontally and vertically; components can be of different sizes.
GridLayout	Arranges components in fixed-length rows and columns.

Container's setLayout()

A container has a setLayout() method to set its layout manager:

```
// java.awt.Container
```

public void setLayout(LayoutManager mgr)

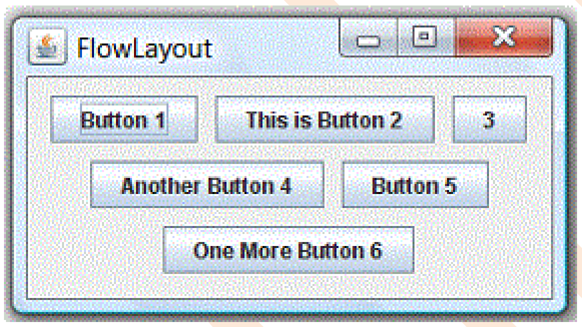
You can get the current layout via Container's getLayout().

To set up the layout of a Container (such as Frame, JFrame, Panel, or JPanel), you have to:

1. Construct an instance of the chosen layout object, via new and constructor, e.g., new FlowLayout())
2. Invoke the setLayout() method of the Container, with the layout object created as the argument;
3. Place the GUI components into the Container using the add() method in the correct order; or into the correct zones.

FlowLayout

In the java.awt.FlowLayout, components are arranged from left-to-right inside the container in the order that they are added (via method aContainer.add(aComponent)). When one row is filled, a new row will be started. The actual appearance depends on the width of the display window.



Fields of FlowLayout class:

1. **public static final int LEFT**
2. **public static final int RIGHT**
3. **public static final int CENTER**
4. **public static final int LEADING**
5. **public static final int TRAILING**

Constructors of FlowLayout class:

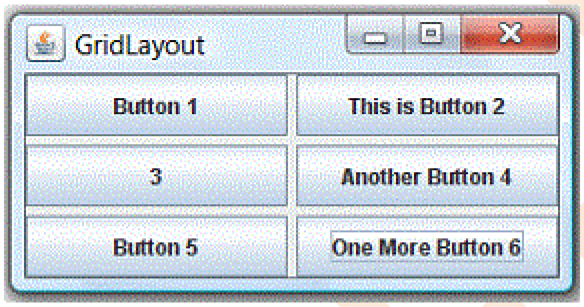
1. **FlowLayout():** creates a flow layout with centered alignment and a default 5 unit horizontal and vertical gap.
2. **FlowLayout(int align):** creates a flow layout with the given alignment and a default 5 unit horizontal and vertical gap.
3. **FlowLayout(int align, int hgap, int vgap):** creates a flow layout with the given alignment and the given horizontal and vertical gap.

Example

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

// An AWT GUI program inherits the top-level container java.awt.Frame
public class AWTFlowLayoutDemo extends Frame {
  private Button btn1, btn2, btn3, btn4, btn5, btn6;

  /** Constructor to setup GUI components */
  public AWTFlowLayoutDemo () {
    setLayout(new FlowLayout());
      // "this" Frame sets layout to FlowLayout, which arranges the components
      // from left-to-right, and flow from top-to-bottom.

    btn1 = new Button("Button 1");
    add(btn1);
    btn2 = new Button("This is Button 2");
    add(btn2);
    btn3 = new Button("3");
    add(btn3);
    btn4 = new Button("Another Button 4");
    add(btn4);
    btn5 = new Button("Button 5");
    add(btn5);
    btn6 = new Button("One More Button 6");
    add(btn6);

    setTitle("FlowLayout Demo"); // "this" Frame sets title
    setSize(280, 150);      // "this" Frame sets initial size
    setVisible(true);       // "this" Frame shows
  }

  /** The entry main() method */
  public static void main(String[] args) {
    new AWTFlowLayoutDemo();  // Let the constructor do the job
  }
}
```

GridLayout

In java.awt.GridLayout, components are arranged in a grid (matrix) of rows and columns inside the Container. Components are added in a left-to-right, top-to-bottom manner in the order they are added (via method *aContainer*.add(*aComponent*)).



Constructors of GridLayout Class

Name	Description
GridLayout()	This constructor creates new Grid Layout.
GridLayout(int rows, int cols)	This constructor creates new Grid Layout with specific number of row and columns.
GridLayout(int rows, int cols, int hgap, int vgap)	This constructor creates new Grid Layout with specific number of row and columns with horizontal and vertical gap between components.

Methods of GridLayout Class

Method	Description
addLayoutComponent(Component comp, Object constraints)	This method adds the specific component.
getColumns()	This method returns the columns.
getHgap()	This method returns the Horizontal gap between Components.
getRows()	This method returns the rows.
getVgap()	This method returns the Vertical gap between Components.
setColumns(int cols)	This method sets the specific columns.
setHgap(int hgap)	This method sets the specific Horizontal gap between Components.
setRows(int rows)	This method sets the specific rows.
setVgap(int vgap)	This method sets the specific Vertical gap between Components.

Example

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

// An AWT GUI program inherits the top-level container java.awt.Frame

public class AWTGridLayoutDemo extends Frame {
  private Button btn1, btn2, btn3, btn4, btn5, btn6;
  /** Constructor to setup GUI components */
  public AWTGridLayoutDemo () {
    setLayout(new GridLayout(3, 2, 3, 3));
      // "this" Frame sets layout to 3x2 GridLayout, horizontal and verical gaps of
3 pixels

    // The components are added from left-to-right, top-to-bottom
    btn1 = new Button("Button 1");
    add(btn1);
    btn2 = new Button("This is Button 2");
    add(btn2);
    btn3 = new Button("3");
    add(btn3);
    btn4 = new Button("Another Button 4");
    add(btn4);
    btn5 = new Button("Button 5");
    add(btn5);
    btn6 = new Button("One More Button 6");
    add(btn6);

    setTitle("GridLayout Demo"); // "this" Frame sets title
    setSize(280, 150);      // "this" Frame sets initial size
    setVisible(true);       // "this" Frame shows
  }

  /** The entry main() method */
  public static void main(String[] args) {
    new AWTGridLayoutDemo();  // Let the constructor do the job
  }
}
```

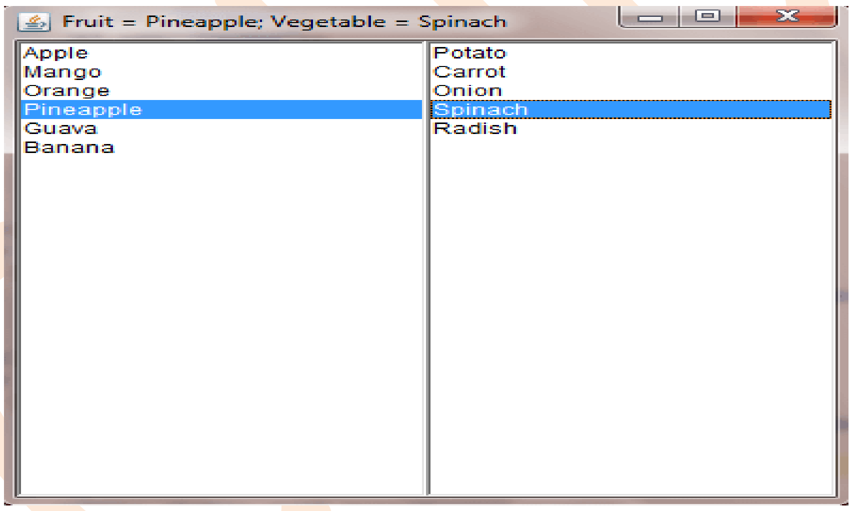
If rows or cols is 0, but not both, then any number of components can be placed in that column or row. If both the rows and cols are specified, the cols value is ignored. The actual cols is determined by the actual number of components and rows.

Example 2

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

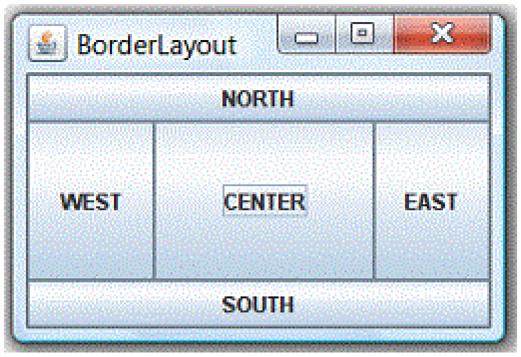
```
class GridLayoutDemo extends Frame
{
List l1,l2;
public GridLayoutDemo()
{
// Set the frame properties
setTitle("GridLayout Demo");
setSize(400,400);
setLayout(new GridLayout());
setLocationRelativeTo(null);
setVisible(true);
// Create lists
l1=new List();
l2=new List();
// Add items to lists
l1.add("Apple");
l1.add("Mango");
l1.add("Orange");
l1.add("Pineapple");
l1.add("Guava");
l1.add("Banana");
l2.add("Potato");
l2.add("Carrot");
l2.add("Onion");
l2.add("Spinach");
l2.add("Radish");
// Add lists
add(l1);
add(l2);
// Add item listeners
l1.addItemListener(new ItemListener()
{
    public void itemStateChanged(ItemEvent ie)
    {
            setTitle("Fruit =   "+ l1.getSelectedItem()+";
            Vegetable = "+l2.getSelectedItem());
    }
 });
l2.addItemListener(new ItemListener()
{
    public void itemStateChanged(ItemEvent ie)
    {
    setTitle("Fruit =   "+l1.getSelectedItem()+";
    Vegetable = "+l2.getSelectedItem());
     }
});
}

public static void main(String args[])
{
new GridLayoutDemo();
}
}
```

Output :



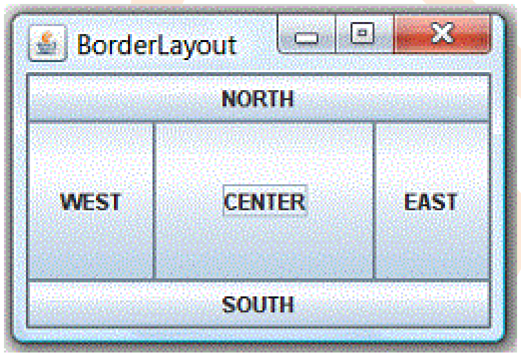
```
Fruit = Pineapple; Vegetable = Spinach

Apple          Potato
Mango          Carrot
Orange         Onion
Pineapple      Spinach
Guava          Radish
Banana
```

BorderLayout



In **java.awt.BorderLayout**, the container is divided into 5 zones: EAST, WEST, SOUTH, NORTH, and CENTER. Components are added using method *aContainer*.add(*acomponent*, *aZone*), where *azone* is either BorderLayout.NORTH (or PAGE_START), BorderLayout.SOUTH (or PAGE_END), BorderLayout.WEST (or LINE_START), BorderLayout.EAST (or LINE_END), or BorderLayout.CENTER. The method *aContainer*.add(*aComponent*) without specifying the zone adds the component to the CENTER.

You need not add components to all the 5 zones. The NORTH and SOUTH components may be stretched horizontally; the EAST and WEST components may be stretched vertically; the CENTER component may stretch both horizontally and vertically to fill any space left over.



Constructors of BorderLayout Class

Name	Description
BorderLayout()	This constructor creates new BorderLayout without any space between Components.

Name	Description
BorderLayout(int hgap, int vgap)	This constructor creates new BorderLayout with horizontal and Vertical space between Components.

Methods of BorderLayout Class

Method	Description
addLayoutComponent(Component comp, Object constraints)	This method adds the specific component.
getHgap()	This method gets the Horizontal gap between components.
getVgap()	This method gets the Vertical gap between components.
setHgap(int hgap)	This method sets the Horizontal gap between components.
setVgap()	

```
import java.awt.*;
import java.awt.event.*;
public class AWTBorderLayoutDemo extends Frame {
  private Button btnNorth, btnSouth, btnCenter, btnEast, btnWest;

  /** Constructor to setup GUI components */
  public AWTBorderLayoutDemo () {
    setLayout(new BorderLayout(3, 3));
      // "this" Frame sets layout to BorderLayout,
      // horizontal and vertical gaps of 3 pixels

    // The components are added to the specified zone
    btnNorth = new Button("NORTH");
    add(btnNorth, BorderLayout.NORTH);
    btnSouth = new Button("SOUTH");
    add(btnSouth, BorderLayout.SOUTH);
    btnCenter = new Button("CENTER");
    add(btnCenter, BorderLayout.CENTER);
    btnEast = new Button("EAST");
    add(btnEast, BorderLayout.EAST);
    btnWest = new Button("WEST");
    add(btnWest, BorderLayout.WEST);

    setTitle("BorderLayout Demo"); // "this" Frame sets title
    setSize(280, 150);       // "this" Frame sets initial size
    setVisible(true);        // "this" Frame shows
  }

  /** The entry main() method */
  public static void main(String[] args) {
    new AWTBorderLayoutDemo();  // Let the constructor do the job
  }
}
```

CardLayout class

The CardLayout class manages the components in such a manner that only one component is visible at a time. It treats each component as a card that is why it is known as CardLayout.

Constructors of CardLayout class:

1. **CardLayout():** creates a card layout with zero horizontal and vertical gap.
2. **CardLayout(int hgap, int vgap):** creates a card layout with the given horizontal and vertical gap.

Commonly used methods of CardLayout class:

- **public void next(Container parent):** is used to flip to the next card of the given container.

- **public void previous(Container parent):** is used to flip to the previous card of the given container.
- **public void first(Container parent):** is used to flip to the first card of the given container.
- **public void last(Container parent):** is used to flip to the last card of the given container.
- **public void show(Container parent, String name):** is used to flip to the specified card with the given name

Use of a card layout requires a bit more work than the other layouts. The cards are typically held in an object of type **Panel**. This panel must have **CardLayout** selected as its layout manager.

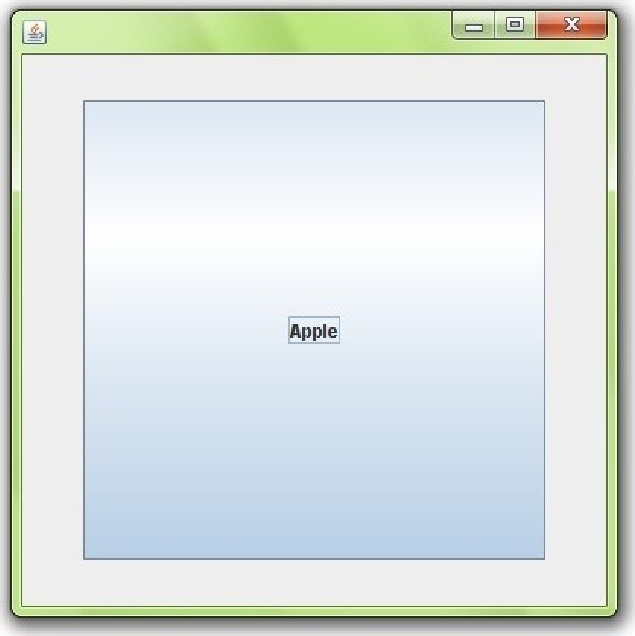
The cards that form the deck are also typically objects of type **Panel**. Thus, you must create a panel that contains the deck and a panel for each card in the deck. Next, you add to the appropriate panel the components that form each card. You then add these panels to the panel for which **CardLayout** is the layout manager. Finally, you add this panel to the window. Once these steps are complete, you must provide some way for the user to select between cards. One common approach is to include one push button for each card in the deck.

When card panels are added to a panel, they are usually given a name. Thus, most of the time, you will use this form of **add( )** when adding cards to a panel:

void add(Component *panelObj*, Object *name*)

Here, *name* is a string that specifies the name of the card whose panel is specified by *panelObj*.

Example of CardLayout class:



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

import javax.swing.*;

public class CardLayoutExample extends JFrame implements ActionListener{
CardLayout card;
JButton b1,b2,b3;
Container c;
   CardLayoutExample()
   {
        c=getContentPane();
       card=new CardLayout(40,30);
//create CardLayout object with 40 hor space and 30 ver space
      c.setLayout(card);

      b1=new JButton("Apple");
      b2=new JButton("Boy");
      b3=new JButton("Cat");
      b1.addActionListener(this);
      b2.addActionListener(this);
      b3.addActionListener(this);
```

```
        c.add("a",b1);c.add("b",b2);c.add("c",b3);

    }
    public void actionPerformed(ActionEvent e) {
    card.next(c);
    }

    public static void main(String[] args) {
        CardLayoutExample cl=new CardLayoutExample();
        cl.setSize(400,400);
        cl.setVisible(true);
        cl.setDefaultCloseOperation(EXIT_ON_CLOSE);
    }
}
```

Example 2: CardLayout

The following example creates a two-level card deck that allows the user to select an operating system. Windows-based operating systems are displayed in one card. Macintosh and Solaris are displayed in the other card.

```
// Demonstrate CardLayout.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="CardLayoutDemo" width=300 height=100>
</applet>
*/
public class CardLayoutDemo extends Applet
implements ActionListener, MouseListener {
Checkbox winXP, winVista, solaris, mac;
Panel osCards;
CardLayout cardLO;
Button Win, Other;
public void init() {
Win = new Button("Windows");
Other = new Button("Other");
add(Win);
add(Other);
cardLO = new CardLayout();
osCards = new Panel();
osCards.setLayout(cardLO); // set panel layout to card layout
winXP = new Checkbox("Windows XP", null, true);
winVista = new Checkbox("Windows Vista");
solaris = new Checkbox("Solaris");
mac = new Checkbox("Mac OS");
// add Windows check boxes to a panel
Panel winPan = new Panel();
winPan.add(winXP);
winPan.add(winVista);
// add other OS check boxes to a panel
Panel otherPan = new Panel();
otherPan.add(solaris);
otherPan.add(mac);
// add panels to card deck panel
osCards.add(winPan, "Windows");
osCards.add(otherPan, "Other");
// add cards to main applet panel
add(osCards);
// register to receive action events
Win.addActionListener(this);
Other.addActionListener(this);
// register mouse events
addMouseListener(this);
public void mousePressed(MouseEvent me) {
cardLO.next(osCards);
}
// Provide empty implementations for the other MouseListener methods.
public void mouseClicked(MouseEvent me) {
}
public void mouseEntered(MouseEvent me) {
}
public void mouseExited(MouseEvent me) {
}
public void mouseReleased(MouseEvent me) {
}
public void actionPerformed(ActionEvent ae) {
if(ae.getSource() == Win) {
```

```
cardLO.show(osCards, "Windows");
}
else {
cardLO.show(osCards, "Other");
}
}
}
```

Example 3 (CardLayout)

```
import java.awt.*;
import java.awt.event.*;
public class TwinCities extends Frame implements ActionListener
{
        List tl, gl, ml, al; // 4 lists to add to a container that implements card layout
        Button fb, lb, nb, pb; // 4 buttons to navigate the cards of the container
        CardLayout cl;
        Panel sp, bp; // 2 panels to add lists and buttons
        public TwinCities()
        {
                super("Welcome"); // title to Frame
            tl = new List(4); // instantiate list objects
            gl = new List(4);
            ml = new List(4);
            al = new List(4);

            // populate theaters list
            tl.add("Sudersan complex");
            tl.add("Sandhya complex");
            tl.add("Odeon complex");
            tl.add("Ramakrishana Estate");
            tl.add("IMAX 70MM");
            tl.add("Surya 35MM");
            tl.add("Shanthi 70MM");
            // populate gardens list
            gl.add("Indira Park");
            gl.add("Lumbibi Park");
            gl.add("Sanjivayya Park");
            gl.add("Zoo Park");
            gl.add("Public Gardens"); // populate monuments list
```

```
ml.add("Chiran Palace");
ml.add("Falaknuma Palace");
ml.add("Charminar");
ml.add("QQ Tombs");
ml.add("Golconda Fort");
ml.add("Zuma Majid");
// populate special attractions list
al.add("Birla Mandir");
al.add("Planetorium");
al.add("Hi-Tech city");
al.add("Buddha Purnima");
al.add("Ramoji Filmcity");
al.add("Shilparamam");

cl = new CardLayout(); // create a card layout object
sp = new Panel();
// create a panel to add all the lists
sp.setLayout(cl); // set the layout to panel, sp
sp.add(tl, "t");
// add lists to panel
sp.add(gl, "g");
sp.add(ml, "m");
// the string parameter is for show() later
sp.add(al, "a");
// instantiate buttons
fb = new Button("First Button");
lb = new Button("Last Button");
nb = new Button("Next Button");
pb = new Button("Previous Button");
bp = new Panel() ;
// create a panel to add buttons
bp.setLayout(new GridLayout(1, 4)); // set layout to bp panel
bp.add(fb);
// add each button to panel
bp.add(nb);
bp.add(pb);
bp.add(lb);
fb.addActionListener(this);
// register the buttons with listener
nb.addActionListener(this);
```

```
            pb.addActionListener(this);
            lb.addActionListener(this);
            add(bp, "South");
            // add panels to frame
            add(sp, "Center");
            setSize(300, 300);
            setVisible(true) ;
        }
        public void actionPerformed(ActionEvent e)
        {
                String str = e.getActionCommand();
           if(str.equals("First Button"))
              cl.first(sp) ;
           else if(str.equals("Next Button"))
                  cl.next(sp);
           else if(str.equals("Previous Button"))
                 cl.previous(sp);
           else if(str.equals("Last Button"))
                cl.last(sp);

         }
        public static void main(String args[])
        {
                new TwinCities();
        }

}
```

ScrollBar Class

The **scrollbar** scrolls through a range of integer values. It is a very convenient component to take a value from the user within a range of values. Scrollbar have two orientations: **Vertical** or **Horizontal**. Scrollbar can be set with maximum and minimum values within which the scrollbar moves. The scroll bar may be given an initial value. At this value, the scroll box is positioned by default in the scroll bar.

Event handler for Scrollbar

Scrollbar generates **AdjustmentEvent** and is handled by **AdjustmentListener**. The **AdjustmentListener** includes only one abstract method **adjustmentValueChanged()** which must be overridden with event handling code.

Scrollbar is a well known window tool in Windows environment. Scrollbar comes with some properties mentioned herewith.

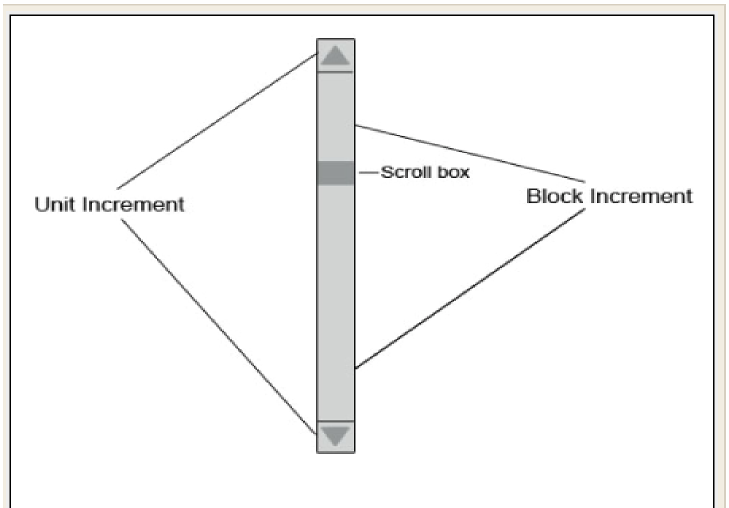


Figure: Scrollbar Properties

Property	Description
Alignment(orientation)	A scrollbar can be either **vertical** or **horizontal**. The default is vertical.
Unit increment	When the user clicks the arrow on either end of the scrollbar, the scrollbar value is changed by this amount for which it is set. By default is it is 1. The value can be set explicitly with the method **setUnitIncrement()**.
Block increment	When the user clicks anywhere in the area between **scroll box** (also known as **thumb**) and arrow, the value is changed by this amount. By default is it is 10. The value can be set explicitly with the method **setBlockIncrement()**.
Minimum and Maximum values	It is the range of values within which the scroll box moves. Beyond these values, the scroll box cannot move.
Initial value	It is the value where by default the scroll box is

	positioned. Or to say, it is the default value given by the scrollbar when displayed.
Scroll box size	This gives the size of the scroll box. It is actually the maximum value minus the visible amount. Visible amount can be set explicitly. For example, if the maximum value is 300 and the visible amount is 50, the size of the scroll box is 250 (300 – 50). But the scroll box can scroll to a maximum value of 300.

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*)

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*)

Example of ScrollBar

```
import java.awt.*;
import java.awt.event.*;
public class ScrollDemo extends Frame implements AdjustmentListener
{
   Scrollbar redScroll, greenScroll, blueScroll;
   Label redLabel, greenLabel, blueLabel;
   Panel p1;

   public ScrollDemo()
   {
       setBackground(Color.yellow);
       p1 = new Panel();
       p1.setLayout(new GridLayout(3, 2, 5, 5));

       redScroll = new Scrollbar(Scrollbar.HORIZONTAL, 0, 0,  0, 255);
       redScroll.setUnitIncrement(5);          //default is 1
       redScroll.setBlockIncrement(15);      //default is 10
       p1.add(redLabel = new Label("RED"));
       p1.add(redScroll);
                                               // simiarly set for green scroll bar
       greenScroll = new Scrollbar(Scrollbar.HORIZONTAL, 0, 0, 0, 255);
       greenScroll.setUnitIncrement(5);
       greenScroll.setBlockIncrement(15);
       p1.add(greenLabel = new Label("GREEN"));
       p1.add(greenScroll);
```

```
    blueScroll = new Scrollbar();
    blueScroll.setOrientation(Scrollbar.HORIZONTAL);
    blueScroll.setValue(0);
    blueScroll.setVisibleAmount(0);
    blueScroll.setUnitIncrement(5);
    blueScroll.setBlockIncrement(10);
    blueScroll.setMinimum(0);
    blueScroll.setMaximum(255);
    p1.add(blueLabel = new Label("BLUE"));
    p1.add(blueScroll);

    redScroll.addAdjustmentListener(this);
    greenScroll.addAdjustmentListener(this);
    blueScroll.addAdjustmentListener(this);

    add(p1,"South");

    setTitle("Playing With Colors");
    setSize(450,325);
    setVisible(true);
}

public Insets getInsets()
{
   Insets is1 = new Insets(5, 8, 10, 25);
   return is1;
}

public void adjustmentValueChanged(AdjustmentEvent e)
{
   int rv = redScroll.getValue();
   int gv = greenScroll.getValue();
   int bv = blueScroll.getValue();

  redLabel.setText("RED: "+ rv);
  greenLabel.setText("GREEN: "+ gv);
  blueLabel.setText("BLUE: "+ bv);

 Color clr1 = new Color(rv, gv, bv);
 setBackground(clr1);
```

```
    }

    public static void main(String args[])
    {
        new ScrollDemo();
    }
}
```

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**, and **MenuItem**. In general, a menu bar contains one or more **Menu** objects. Each **Menu** object contains a list of **MenuItem** objects. Each **MenuItem** object represents something that can be selected by the user. Since **Menu** is a subclass of **MenuItem**, a hierarchy of nested submenus can be created.

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

Following are the constructors for **Menu**:

Menu( )
Menu(String *optionName*)
Menu(String *optionName*, boolean *removable*)

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 type **MenuItem**. It defines these constructors:
MenuItem( )
MenuItem(String *itemName*)
MenuItem(String *itemName*, MenuShortcut *keyAccel*)

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.

Following is an example that adds a series of nested menus to a pop-up window. The item selected is displayed in the window. The state of the two check box menu items is also displayed.

```
// Illustrate menus.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
// Create a subclass of Frame.
class MenuFrame extends Frame {
String msg = "";
CheckboxMenuItem debug, test;
MenuFrame(String title)
{
super(title);
// create menu bar and add it to frame
```

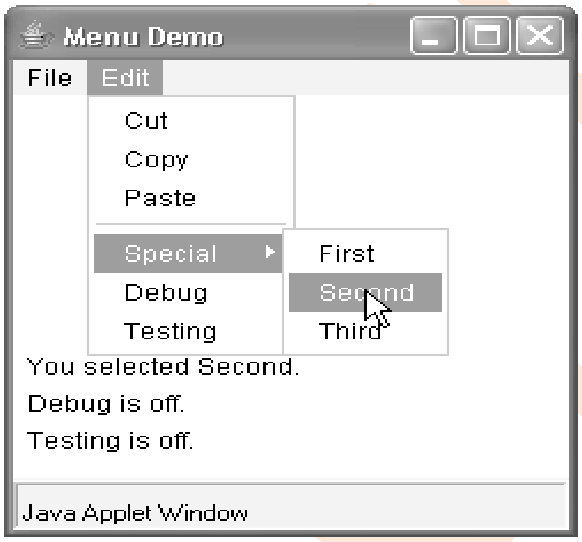
```
MenuBar mbar = new MenuBar();
setMenuBar(mbar);
// create the menu items
Menu file = new Menu("File");
MenuItem item1, item2, item3, item4, item5;
file.add(item1 = new MenuItem("New..."));
file.add(item2 = new MenuItem("Open..."));
file.add(item3 = new MenuItem("Close"));
file.add(item4 = new MenuItem("-"));
file.add(item5 = new MenuItem("Quit..."));
mbar.add(file);
Menu edit = new Menu("Edit");
MenuItem item6, item7, item8, item9;
edit.add(item6 = new MenuItem("Cut"));
edit.add(item7 = new MenuItem("Copy"));
edit.add(item8 = new MenuItem("Paste"));
edit.add(item9 = new MenuItem("-"));
Menu sub = new Menu("Special");
MenuItem item10, item11, item12;
sub.add(item10 = new MenuItem("First"));
sub.add(item11 = new MenuItem("Second"));
sub.add(item12 = new MenuItem("Third"));
edit.add(sub);
// these are checkable menu items
debug = new CheckboxMenuItem("Debug");
edit.add(debug);
test = new CheckboxMenuItem("Testing");
edit.add(test);
mbar.add(edit);
// create an object to handle action and item events
MyMenuHandler handler = new MyMenuHandler(this);
// register it to receive those events
item1.addActionListener(handler);
item2.addActionListener(handler);
item3.addActionListener(handler);
item4.addActionListener(handler);
item5.addActionListener(handler);
item6.addActionListener(handler);
item7.addActionListener(handler);
item8.addActionListener(handler);
```

```
item9.addActionListener(handler);
item10.addActionListener(handler);
item11.addActionListener(handler);
item12.addActionListener(handler);
debug.addItemListener(handler);
test.addItemListener(handler);
// create an object to handle window events
MyWindowAdapter adapter = new MyWindowAdapter(this);
// register it to receive those events
addWindowListener(adapter);
}
public void paint(Graphics g) {
g.drawString(msg, 10, 200);
if(debug.getState())
g.drawString("Debug is on.", 10, 220);
else
g.drawString("Debug is off.", 10, 220);
if(test.getState())
g.drawString("Testing is on.", 10, 240);
else
g.drawString("Testing is off.", 10, 240);
}
}
class MyWindowAdapter extends WindowAdapter {
MenuFrame menuFrame;
public MyWindowAdapter(MenuFrame menuFrame) {
this.menuFrame = menuFrame;
}
public void windowClosing(WindowEvent we) {
menuFrame.setVisible(false);
}
}
class MyMenuHandler implements ActionListener,
ItemListener {
MenuFrame menuFrame;
public MyMenuHandler(MenuFrame menuFrame) {
this.menuFrame = menuFrame;
}
// Handle action events.
public void actionPerformed(ActionEvent ae) {
```

```
String msg = "You selected ";
String arg = ae.getActionCommand();
if(arg.equals("New..."))
msg += "New.";
else if(arg.equals("Open..."))
msg += "Open.";
else if(arg.equals("Close"))
msg += "Close.";
else if(arg.equals("Quit..."))
msg += "Quit.";
else if(arg.equals("Edit"))
msg += "Edit.";
else if(arg.equals("Cut"))
msg += "Cut.";
else if(arg.equals("Copy"))
msg += "Copy.";
else if(arg.equals("Paste"))
msg += "Paste.";
else if(arg.equals("First"))
msg += "First.";
else if(arg.equals("Second"))
msg += "Second.";
else if(arg.equals("Third"))
msg += "Third.";
else if(arg.equals("Debug"))
msg += "Debug.";
else if(arg.equals("Testing"))
msg += "Testing.";
menuFrame.msg = msg;
menuFrame.repaint();
}
// Handle item events.
public void itemStateChanged(ItemEvent ie) {
menuFrame.repaint();
}
}
// Create frame window.
public class MenuDemo extends Applet {
Frame f;
public void init() {
```

```
f = new MenuFrame("Menu Demo");
int width = Integer.parseInt(getParameter("width"));
int height = Integer.parseInt(getParameter("height"));
setSize(new Dimension(width, height));
f.setSize(width, height);
f.setVisible(true);
}
public void start() {
f.setVisible(true);
}
public void stop() {
f.setVisible(false);
}
}
```

OUTPUT



```
You selected Second.
Debug is off.
Testing is off.
```