# Adv. Java Programming (22517): 1- Abstract Windowing Toolkit (AWT)

**Teaching Hours** - 8, **Total Marks** - 12 (R - 2, U - 4, A - 6)

The Abstract Window Toolkit (AWT) was Java's first GUI framework, and it has been part of Java since version 1.0. It contains numerous classes and methods that allow you to create windows and simple controls. A common use of the AWT is in applets, it is also used to create stand-alone windows that run in a GUI environment, such as Windows.

## AWT Classes

AWT classes are contained in the **java.awt** package. It is one of Java's largest packages. following Table lists AWT classes,

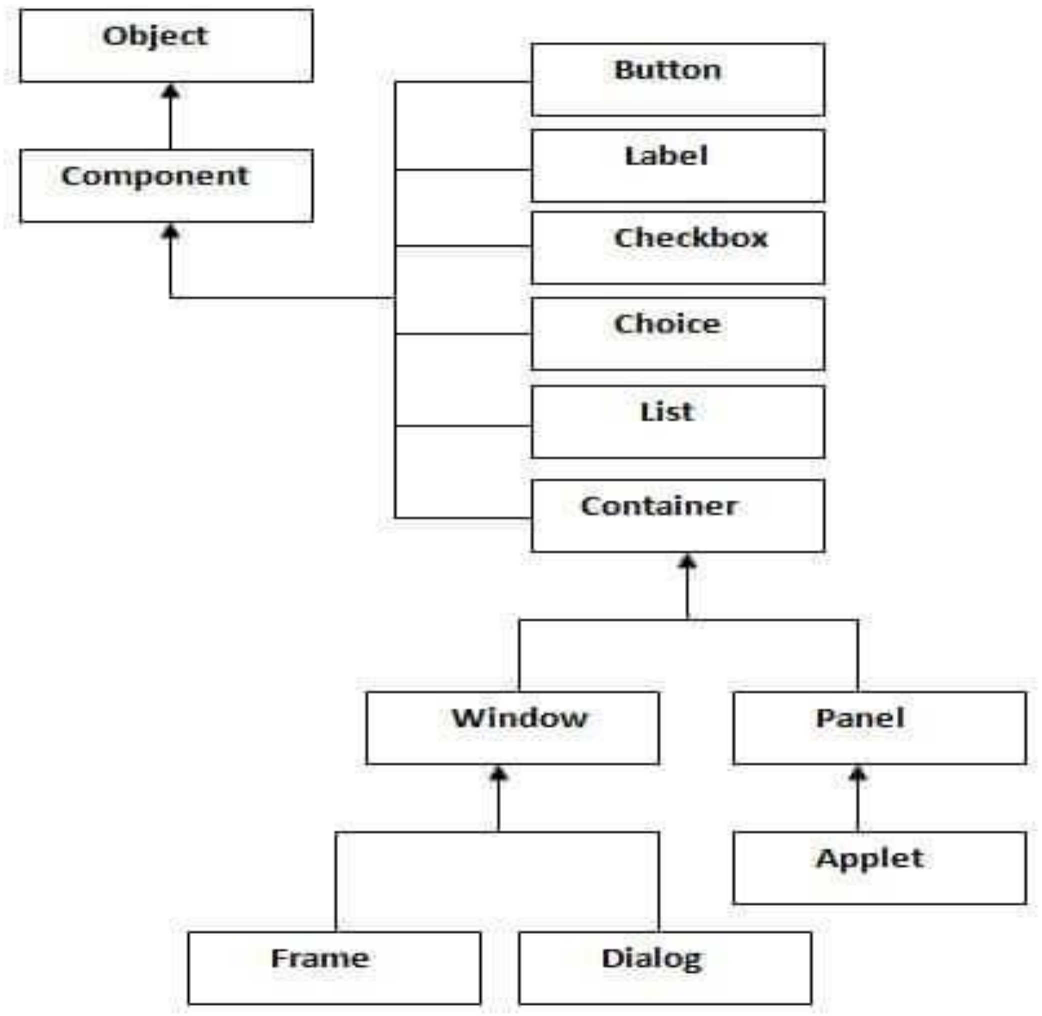
| Class | Description |
|---|---|
| AWTEvent | Encapsulates AWT events. |
| AWTEventMulticaster | Dispatches events to multiple listeners. |
| BorderLayout | The border layout manager. Border layouts use five components: North, South, East, West, and Center. |
| Button | Creates a push button control. |
| Canvas | A blank, semantics-free window. |
| CardLayout | The card layout manager. Card layouts emulate index cards. Only the one on top is showing. |
| Checkbox | Creates a check box control. |
| CheckboxGroup | Creates a group of check box controls. |
| CheckboxMenuItem | Creates an on/off menu item. |
| Choice | Creates a pop-up list. |
| Color | Manages colors in a portable, platform-independent fashion. |
| Component | An abstract superclass for various AWT components. |
| Container | A subclass of **Component** that can hold other components. |
| Cursor | Encapsulates a bitmapped cursor. |
| Dialog | Creates a dialog window. |
| Dimension | Specifies the dimensions of an object. The width is stored in **width**, and the height is stored in **height**. |
| EventQueue | Queues events. |
| FileDialog | Creates a window from which a file can be selected. |
| FlowLayout | The flow layout manager. Flow layout positions components left to right, top to bottom. |
| Font | Encapsulates a type font. |
| FontMetrics | Encapsulates various information related to a font. This information helps you display text in a window. |
| Frame | Creates a standard window that has a title bar, resize corners, and a menu bar. |
| Graphics | Encapsulates the graphics context. This context is used by the various output methods to display output in a window. |

| | |
|---|---|
| GraphicsDevice | Describes a graphics device such as a screen or printer. |
| GraphicsEnvironment | Describes the collection of available **Font** and **GraphicsDevice** objects. |
| GridBagConstraints | Defines various constraints relating to the **GridBagLayout** class. |
| GridBagLayout | The grid bag layout manager. Grid bag layout displays components subject to the constraints specified by **GridBagConstraints**. |
| GridLayout | The grid layout manager. Grid layout displays components in a two dimensional grid. |
| Image | Encapsulates graphical images. |
| Insets | Encapsulates the borders of a container. |
| Label | Creates a label that displays a string. |
| List | Creates a list from which the user can choose. Similar to the standard Windows list box. |
| MediaTracker | Manages media objects. |
| Menu | Creates a pull-down menu. |
| MenuBar | Creates a menu bar. |
| MenuComponent | An abstract class implemented by various menu classes. |
| MenuItem | Creates a menu item. |
| MenuShortcut | Encapsulates a keyboard shortcut for a menu item. |
| Panel | The simplest concrete subclass of **Container**. |
| Point | Encapsulates a Cartesian coordinate pair, stored in **x** and **y**. |
| Polygon | Encapsulates a polygon. |
| PopupMenu | Encapsulates a pop-up menu. |
| PrintJob | An abstract class that represents a print job. |
| Rectangle | Encapsulates a rectangle. |
| Robot | Supports automated testing of AWT-based applications. |
| Scrollbar | Creates a scroll bar control. |
| ScrollPane | A container that provides horizontal and/or vertical scroll bars for another component. |
| SystemColor | Contains the colors of GUI widgets such as windows, scroll bars, text, and others. |
| TextArea | Creates a multiline edit control. |
| TextComponent | A superclass for **TextArea** and **TextField**. |
| TextField | Creates a single-line edit control. |
| Toolkit | Abstract class implemented by the AWT. |
| Window | Creates a window with no frame, no menu bar, and no title. |

## 1.1 Component, Container, Window, Frame, Panel

### Window Fundamentals

The AWT defines windows according to a class hierarchy that adds functionality and specificity with each level.



### Component:

At the top of the AWT hierarchy is the **Component** class. **Component** is an abstract class that encapsulates all of the attributes of a visual component. Except for menus, all user interface elements that are displayed on the screen and that interact with the user are subclasses of **Component**. A **Component** object is responsible for remembering the current foreground and background colors and the currently selected text font.

**Container:**

The **Container** class is a subclass of **Component**. It has additional methods that allow other **Component** objects to be nested within it. A container is responsible for laying out (that is, positioning) any components that it contains. It does this through the use of various layout managers.

**Panel:**

The **Panel** class is a concrete subclass of **Container**. A **Panel** may be thought of as a recursively nestable, concrete screen component. **Panel** is the superclass for **Applet**. When screen output is directed to an applet, it is drawn on the surface of a **Panel** object.

In essence, a **Panel** is a window that does not contain a title bar, menu bar, or border. This is why you don't see these items when an applet is run inside a browser. When you run an applet using an applet viewer, the applet viewer provides the title and border.

**Window:**

The **Window** class creates a top-level window. A *top-level window* is not contained within any other object; it sits directly on the desktop. Generally, you won't create **Window** objects directly. Instead, you will use a subclass of **Window** called **Frame**

.

**Frame:**

**Frame** encapsulates what is commonly thought of as a "window." It is a subclass of **Window** and has a title bar, menu bar, borders, and resizing corners.

**Canvas:**

Although it is not part of the hierarchy for applet or frame windows, there is one other type of window that you will find valuable: **Canvas**. Derived from **Component**, **Canvas** encapsulates a blank window upon which you can draw.

## Working with Frame Windows

Type of AWT-based window created is derived from Frame. You will use it to create child windows within applets, and top-level or child windows for stand-alone applications. Here are two of Frame's constructors:

```
Frame( ) throws HeadlessException
Frame(String title) throws HeadlessException
```

The first form creates a standard window that does not contain a title. The second form creates a window with the title specified by title.

Notice that you cannot specify the dimensions of the window. Instead, you must set the size of the window after it has been created. A HeadlessException is thrown if an attempt is made to create a Frame instance in an environment that does not support user interaction.

There are several key methods you will use when working with Frame windows. They are examined here.

Setting the Window's Dimensions

void setSize(int newWidth, int newHeight)

void setSize(Dimension newSize)

The new size of the window is specified by newWidth and newHeight, or by the width and height fields of the Dimension object passed in newSize. The dimensions are specified in terms of pixels.

The getSize( ) method is used to obtain the current size of a window. One of its forms is shown here:

Dimension getSize( )

This method returns the current size of the window contained within the width and height fields of a Dimension object.

**The HeadlessException**

When an attempt is made to instantiate a GUI component in a noninteractive environment (such as one in which no display, mouse, or keyboard is present). You can use this exception to write code that can adapt to non-interactive environments.

## Hiding and Showing a Window

After a frame window has been created, it will not be visible until you call **setVisible( )**. Its signature is shown here:

void setVisible(boolean *visibleFlag*)

The component is visible if the argument to this method is **true**. Otherwise, it is hidden.

## Setting a Window's Title

You can change the title in a frame window using **setTitle( )**, which has this general form:

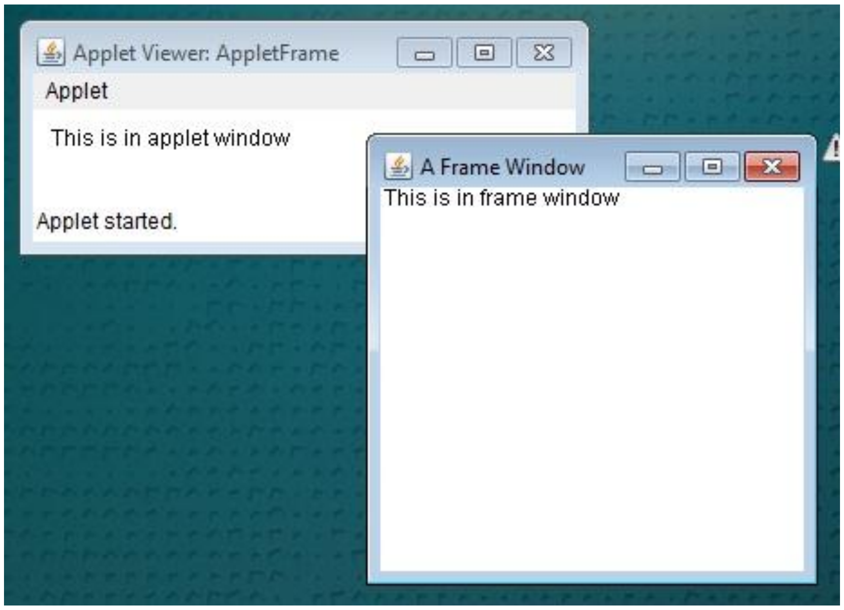
void setTitle(String *newTitle*)

Here, *newTitle* is the new title for the window.

## 1.2 Creating Windowed programs and applets

```
// Create a child frame window from within an applet.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/* <applet code="AppletFrame" width=300 height=50>  </applet>
*/
// Create a subclass of Frame.
class SampleFrame extends Frame   {
        SampleFrame(String title)  {
                super(title);
                // 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("This is in frame window", 10, 40);
        }
}
class MyWindowAdapter extends WindowAdapter {
        SampleFrame sampleFrame;
        public MyWindowAdapter(SampleFrame sampleFrame) {
                this.sampleFrame = sampleFrame;
        }
        public void windowClosing(WindowEvent we)  {
                sampleFrame.setVisible(false);
        }
}

// Create frame window.
public class AppletFrame extends Applet {
        Frame f;
        public void init()  {
                f = new SampleFrame("A Frame Window");
                f.setSize(250, 250);
                f.setVisible(true);
        }
        public void start()  {
                f.setVisible(true);
        }
        public void stop()  {
                f.setVisible(false);
        }
        public void paint(Graphics g)  {
                g.drawString("This is in applet window", 10, 20);
        }
}
```



```
This is in applet window
```

```
Applet started.
```

```
This is in frame window
```

## 1.3 AWT controls and Layout Managers :

*Controls* are components that allow a user to interact with your application in various ways—for example; a commonly used control is the push button.

A *layout manager* automatically positions components within a container. Thus, the appearance of a window is determined by a combination of the controls that it contains and the layout manager used to position them.

In addition to the controls, a frame window can also include a standard-style *menu bar*. Each entry in a menu bar activates a drop-down menu of options from which the user can choose. This constitutes the *main menu* of an application. As a general rule, a menu bar is positioned at the top of a window.

### AWT Control Fundamentals

The AWT supports the following types of controls:

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

These controls are subclasses of **Component**.

(Note that both Swing and JavaFX provide a substantially larger, more sophisticated set of controls.)

### Adding and Removing Controls

To include a control in a window, you must add it to the window. To do this, you must first create an instance of the desired control and then add it to a window by calling **add( )**, which is defined by **Container**.

The **add( )** method has several forms. The following form is the one that is used for the first part of this chapter:

Component add(Component *compRef*)

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

Sometimes you will want to remove a control from a window when the control is no longer needed. To do this, call **remove( )**. This method is also defined by **Container**. Here is one of its forms:

void remove(Component *compRef*)

Here, *compRef* is a reference to the control you want to remove. You can remove all controls by calling **removeAll( )**.

## Labels

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

**Label** defines the following constructors:

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

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

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

void setText(String *str*)
String getText( )

For **setText( )**, *str* specifies the new label. For **getText( )**, the current label is returned.

You can set the alignment of the string within the label by calling **setAlignment( )**. To obtain the current alignment, call **getAlignment( )**. The methods are as follows:

void setAlignment(int *how*)
int getAlignment( )

Here, *how* must be one of the alignment constants shown earlier.

The following example creates three labels and adds them to an applet window:

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

                // add labels to applet window
                add(one);
                add(two);
                add(three);
        }
}
```

## Button

A *push button* is a component that contains a label and that generates an event when it is pressed. Push buttons are objects of type **Button**. **Button** defines these two constructors:

Button( ) throws HeadlessException
Button(String *str*) throws HeadlessException
The first version creates an empty button.
The second creates a button that contains *str* as a label.

After a button has been created, you can set its label by calling **setLabel( )**. You can retrieve its label by calling **getLabel( )**. These methods are as follows:

void setLabel(String *str*)
String getLabel( )
Here, *str* becomes the new label for the button.

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

## Applying Check Boxes

A *check box* is a control that is used to turn an option on or off. It consists of a small box that can either contain a check mark or not. There is a label associated with each check box that describes what option the box represents.

You change the state of a check box by clicking on it. Check boxes can be used individually or as part of a group. Check boxes are objects of the **Checkbox** class.

**Checkbox** supports these constructors:

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

The first form creates a check box whose label is initially blank. The state of the check box is unchecked.

The second form creates a check box whose label is specified by *str*. The state of the check box is unchecked.

The third form allows you to set the initial state of the check box. If *on* is **true**, the check box is initially checked; otherwise, it is cleared.

The fourth and fifth forms create a check box whose label is specified by *str* and whose group is specified by *cbGroup*. If this check box is not part of a group, then *cbGroup* must be **null**. The value of *on* determines the initial state of the check box.

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

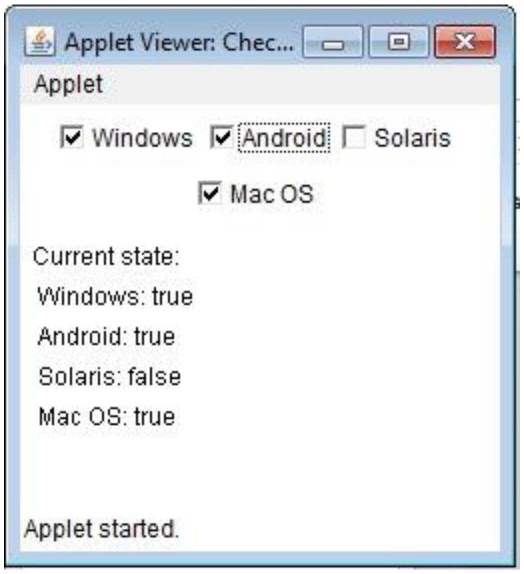
boolean getState( )
void setState(boolean *on*)
String getLabel( )
void setLabel(String *str*)

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

// Demonstrate check boxes.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="CheckboxDemo" width=240 height=200>
</applet>
*/
public class CheckboxDemo extends Applet implements ItemListener  {
        String msg = "";
        Checkbox windows, android, solaris, mac;
        public void init() {
                windows = new Checkbox("Windows", null, true);
                android = new Checkbox("Android");
                solaris = new Checkbox("Solaris");
                mac = new Checkbox("Mac OS");
                add(windows);
                add(android);
                add(solaris);
                add(mac);
                windows.addItemListener(this);
                android.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: " + windows.getState();
                g.drawString(msg, 6, 100);
                msg = " Android: " + android.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);
        }
```

}



```
☑ Windows  ☑ Android  ☐ Solaris
☑ Mac OS
Current state:
Windows: true
Android: true
Solaris: false
Mac OS: true
Applet started.
```

## CheckboxGroup

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.

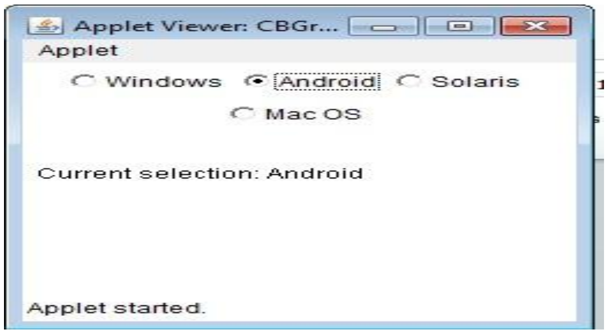
You can determine which check box in a group is currently selected by calling **getSelectedCheckbox( )**. You can set a check box by calling **setSelectedCheckbox( )**.

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.

```
// Demonstrate check box group.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*  <applet code="CBGroup" width=240 height=200> </applet> */
public class CBGroup extends Applet implements ItemListener {
	String msg = "";
	Checkbox windows, android, solaris, mac;
	CheckboxGroup cbg;
	public void init() {
		cbg = new CheckboxGroup();
		windows = new Checkbox("Windows", cbg, true);
		android = new Checkbox("Android", cbg, false);
		solaris = new Checkbox("Solaris", cbg, false);
		mac = new Checkbox("Mac OS", cbg, false);
		add(windows);
		add(android);
		add(solaris);
		add(mac);
		windows.addItemListener(this);
		android.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);
	}
}
```



Current selection: Android

Applet started.

## Choice

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

Each item in the list is a string that appears as a left-justified label in the order it is added to the **Choice** object.

**Choice** defines only 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 item. **getSelectedIndex( )** returns the index of the item. The first item is at index 0.

By default, the first item added to the list is selected. To obtain the number of items in the list, call **getItemCount( )**. You can set the currently selected item using the **select( )** method with either a zero-based integer index or

a string that will match a name in the list. These methods are shown here:

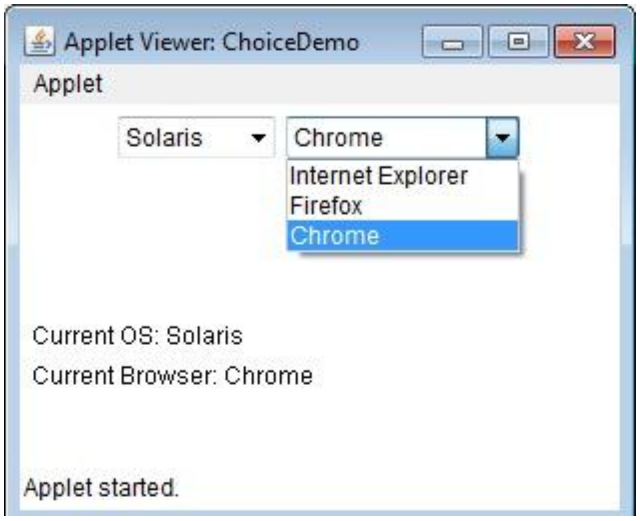
int getItemCount( )
void select(int *index*)
void select(String *name*)

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

String getItem(int *index*)

Here, *index* specifies the index of the desired item.

```
// Demonstrate Choice lists.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="ChoiceDemo" width=300 height=180> </applet>
*/
public class ChoiceDemo extends Applet implements ItemListener {
        Choice os, browser;
        String sg = "";
        public void init() {
                os = new Choice();
                browser = new Choice();
                // add items to os list
                os.add("Windows");
                os.add("Android");
                os.add("Solaris");
                os.add("Mac OS");
                // add items to browser list
                browser.add("Internet Explorer");
                browser.add("Firefox");
                browser.add("Chrome");
                // add choice lists to window
                add(os);
                add(browser);
                // register to receive item events
                os.addItemListener(this);
                browser.addItemListener(this);
        }
        public void itemStateChanged(ItemEvent ie) {
                repaint();
        }
        // Display current selections.
        public void paint(Graphics g) {
                msg = "Current OS: ";
                msg += os.getSelectedItem();
                g.drawString(msg, 6, 120);
                msg = "Current Browser: ";
                msg += browser.getSelectedItem();
                g.drawString(msg, 6, 140);
        }
}
```



```
Current OS: Solaris
Current Browser: Chrome
```

## List

The **List** class provides a compact, multiple-choice, scrolling selection list. **List** object can be constructed to show any number of choices in the visible window. It can also be created to allow multiple selections.

**List** provides these constructors:

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

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

In the second form, the value of *numRows* specifies the number of entries in the list that will always be visible (others can be scrolled into view as needed).

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

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

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

Here, *name* is the name of the item added to the list.

The first form adds items to the end of the list.

The second form adds the item at the index specified by *index*. Indexing begins at zero. You can specify –1 to add the item to the end of the list.

For lists that allow only single selection, you can determine which item is currently selected by calling either **getSelectedItem( )** or **getSelectedIndex( )**. These methods are shown here:

String getSelectedItem( )

int getSelectedIndex( )

The **getSelectedItem( )** method returns a string containing the name of the item. If more than one item is selected, or if no selection has yet been made, **null** is returned.

**getSelectedIndex( )** returns the index of the item. The first item is at index 0. If more than one item is selected, or if no selection has yet been made, –1 is returned.

For lists that allow multiple selection, you must use either **getSelectedItems( )** or **getSelectedIndexes( )**, shown here, to determine the current selections:

String[ ] getSelectedItems( )
int[ ] getSelectedIndexes( )

**getSelectedItems( )** returns an array containing the names of the currently selected items.
**getSelectedIndexes( )** returns an array containing the indexes of the currently selected items.

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

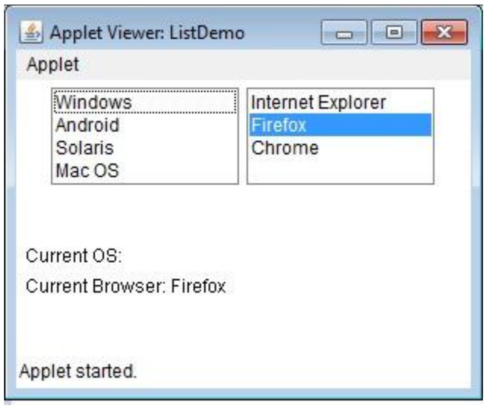
int getItemCount( )
void select(int *index*)

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

String getItem(int *index*)

Here, *index* specifies the index of the desired item.

```
// Demonstrate Lists.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="ListDemo" width=300 height=180>  </applet>
*/
public class ListDemo extends Applet implements ActionListener {
        List os, browser;
        String msg = "";
        public void init() {
                os = new List(4, true);
                browser = new List(4, false);
                // add items to os list
                os.add("Windows");
                os.add("Android");
                os.add("Solaris");
                os.add("Mac OS");
                // add items to browser list
                browser.add("Internet Explorer");
                browser.add("Firefox");
                browser.add("Chrome");
                browser.select(1);
                // add lists to window
                add(os);
                add(browser);
                // register to receive action events
                os.addActionListener(this);
                browser.addActionListener(this);
        }
        public void actionPerformed(ActionEvent ae) {
                repaint();
        }
        // Display current selections.
        public void paint(Graphics g) {
                int idx[];
                msg = "Current OS: ";
                idx = os.getSelectedIndexes();
                for(int i=0; i<idx.length; i++)
                msg += os.getItem(idx[i]) + " ";
                g.drawString(msg, 6, 120);
                msg = "Current Browser: ";
                msg += browser.getSelectedItem();
                g.drawString(msg, 6, 140);
        }
}
```



```
Current OS:
Current Browser: Firefox
```

Applet started.

## Managing Scroll Bars

*Scroll bars* are used to select continuous values between a specified minimum and maximum. Scroll bars may be oriented horizontally or vertically. A scroll bar is actually a composite of several individual parts. Each end has an arrow that you can click to move the current value of the scroll bar one unit in the direction of the arrow.

The current value of the scroll bar relative to its minimum and maximum values is indicated by the *slider box* (or *thumb*) for the scroll bar. The slider box can be dragged by the user to a new position. The scroll bar will then reflect this value. In the background space on either side of the thumb, the user can click to cause the thumb to jump in that direction by some increment larger than 1. Typically, this action translates into some form of page up and page down. Scroll bars are encapsulated by the **Scrollbar** class.

**Scrollbar** defines the following constructors:

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

The first form creates a vertical scroll bar.

The second and third forms allow you to specify the orientation of the scroll bar. If *style* is **Scrollbar.VERTICAL**, a vertical scroll bar is created. If *style* is **Scrollbar.HORIZONTAL**, the scroll bar is horizontal.

In the third form of the constructor, the initial value of the scroll bar is passed in *initialValue*. The number of units represented by the height of the thumb is passed in *thumbSize*. The minimum and maximum values for the scroll bar are specified by *min* and *max*.

If you construct a scroll bar by using one of the first two constructors, then you need to set its parameters by using **setValues( )**, shown here, before it can be used:

void setValues(int *initialValue*, int *thumbSize*, int *min*, int *max*)

To obtain the current value of the scroll bar, call **getValue( )**. It returns the current setting. To set the current value, call **setValue( )**. These methods are as follows:

int getValue( )
void setValue(int *newValue*)

Here, *newValue* specifies the new value for the scroll bar. When you set a value, the slider box inside the scroll bar will be positioned to reflect the new value.

You can also retrieve the minimum and maximum values via **getMinimum( )** and **getMaximum( )**, shown here:

int getMinimum( )
int getMaximum( )

They return the requested quantity.

By default, 1 is the increment added to or subtracted from the scroll bar each time it is scrolled up or down one line. You can change this increment by calling **setUnitIncrement( )**. By default, page-up and page-down increments are 10. You can change this value by calling **setBlockIncrement( )**. These methods are shown here:

void setUnitIncrement(int *newIncr*)
void setBlockIncrement(int *newIncr*)

## TextField

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

**TextField** is a subclass of **TextComponent**. **TextField** defines the following constructors:

TextField( ) throws HeadlessException
TextField(int *numChars*) throws HeadlessException
TextField(String *str*) throws HeadlessException
TextField(String *str*, int *numChars*) throws HeadlessException

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

To obtain the string currently contained in the text field, call **getText( )**. To set the text, call **setText( )**. These methods are as follows:

String getText( )
void setText(String *str*)

Here, *str* is the new string.

The user can select a portion of the text in a text field. Also, you can select a portion of text under program control by using **select( )**. Your program can obtain the currently selected text by calling **getSelectedText( )**. These methods are shown here:

String getSelectedText( )
void select(int *startIndex*, int *endIndex*)

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

You can control whether the contents of a text field may be modified by the user by calling **setEditable( )**. You can determine editability by calling **isEditable( )**. These methods are shown here:

```
boolean isEditable( )
void setEditable(boolean canEdit)
```

**isEditable( )** returns **true** if the text may be changed and **false** if not. In **setEditable( )**, if *canEdit* is **true**, the text may be changed. If it is **false**, the text cannot be altered.

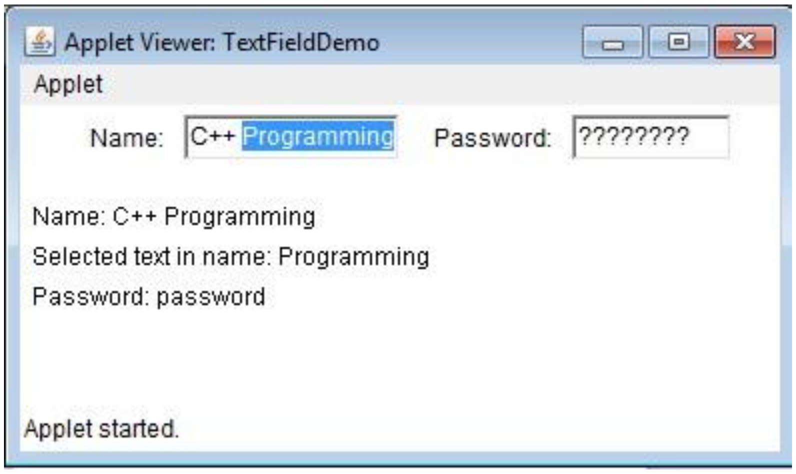
There may be times when you will want the user to enter text that is not displayed, such as a password. You can disable the echoing of the characters as they are typed by calling **setEchoChar( )**. This method specifies a single character that the **TextField** will display when characters are entered (thus, the actual characters typed will not be shown). You can check a text field to see if it is in this mode with the **echoCharIsSet( )** method. You can retrieve the echo character by calling the **getEchoChar( )** method. These methods are as follows:

```
void setEchoChar(char ch)
boolean echoCharIsSet( )
char getEchoChar( )
```

Here, *ch* specifies the character to be echoed. If *ch* is zero, then normal echoing is restored.

```
// Demonstrate text field.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="TextFieldDemo" width=380 height=150>
</applet>
*/
public class TextFieldDemo extends Applet implements ActionListener {
	TextField name, pass;
	public void init() {
		Label namep = new Label("Name: ", Label.RIGHT);
		Label passp = new Label("Password: ", Label.RIGHT);
		name = new TextField(12);
		pass = new TextField(8);
		pass.setEchoChar('?');
		add(namep);
		add(name);
		add(passp);
		add(pass);
		// register to receive action events
		name.addActionListener(this);
		pass.addActionListener(this);
	}
```

```
        // User pressed Enter.
        public void actionPerformed(ActionEvent ae) {
                repaint();
        }
        public void paint(Graphics g) {
                g.drawString("Name: " + name.getText(), 6, 60);
                g.drawString("Selected text in name: "+ name.getSelectedText(), 6, 80);
                g.drawString("Password: " + pass.getText(), 6, 100);
        }
}
```



```
Name: C++ Programming
Selected text in name: Programming
Password: password
```

## TextArea

AWT includes a simple multiline editor called **TextArea**. Following are the constructors for **TextArea**:

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

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

*sBars* must be one of these values:

| | |
|---|---|
| SCROLLBARS_BOTH | SCROLLBARS_NONE |
| SCROLLBARS_HORIZONTAL_ONLY | SCROLLBARS_VERTICAL_ONLY |

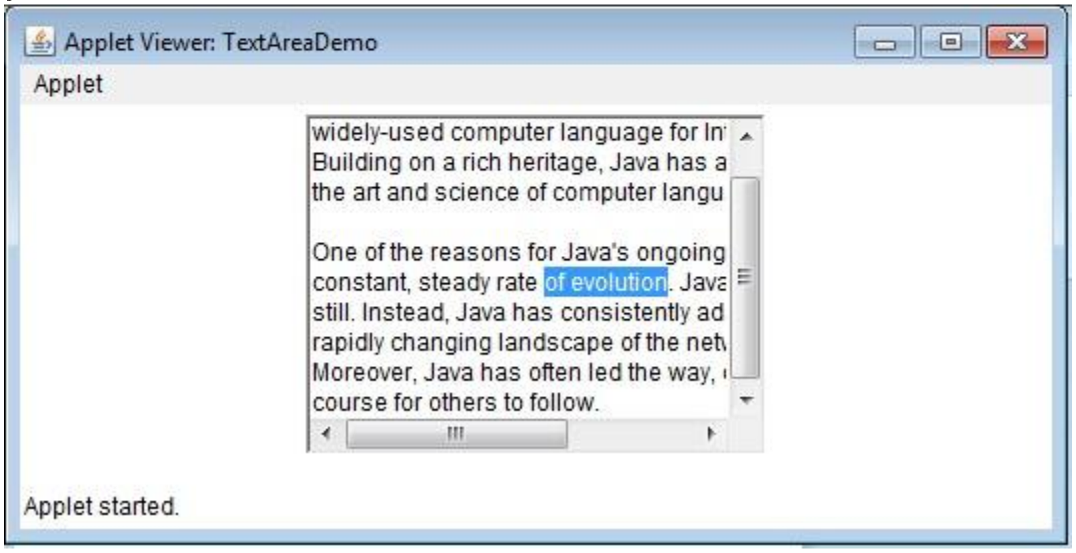
**TextArea** is a subclass of **TextComponent**. Therefore, it supports the **getText( )**, **setText( )**, **getSelectedText( )**, **select( )**, **isEditable( )**, and **setEditable( )** methods described in the preceding section. **TextArea** adds the following editing methods:

void append(String *str*)
void insert(String *str*, int *index*)
void replaceRange(String *str*, int *startIndex*, int *endIndex*)

The **append( )** method appends the string specified by *str* to the end of the current text. **insert( )** inserts the string passed in *str* at the specified index. To replace text, call **replaceRange( )**. It replaces the characters from *startIndex* to *endIndex*–1, with the replacement text passed in *str*.

```
// 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 8 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);
        }
}
```



```
widely-used computer language for In
Building on a rich heritage, Java has a
the art and science of computer langu

One of the reasons for Java's ongoing
constant, steady rate of evolution. Java
still. Instead, Java has consistently ad
rapidly changing landscape of the netw
Moreover, Java has often led the way, 
course for others to follow.
```

Applet started.

## 1.4 AWT controls and Layout Managers:

Every **Container** object has a layout manager associated with it. A layout manager is an instance of any class that implements the **LayoutManager** interface. The layout manager is set by **setLayout( )** method.

If no call to **setLayout( )** is made, then the default layout manager is used. Whenever a container is resized (or sized for the first time), the layout manager is used to position each of the components within it.

The **setLayout( )** method has the following general form:

void setLayout(LayoutManager *layoutObj*)

Here, *layoutObj* is a reference to the desired layout manager. If you wish to disable the layout manager and position components manually, pass **null** for *layoutObj*.

If you do this, you will need to determine the shape and position of each component manually, using the **setBounds( )** method defined by **Component**.

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

### FlowLayout

**FlowLayout** is the default layout manager. **FlowLayout** implements a simple layout style, which is similar to how words flow in a text editor. The direction of the layout, by default, is left to right, top to bottom.

Therefore, by default, components are laid out line-by-line beginning at the upper-left corner. In all cases, when a line is filled, layout advances to the next line. A small space is left between each component, above and below, as well as left and right. Here are the constructors for **FlowLayout**:

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

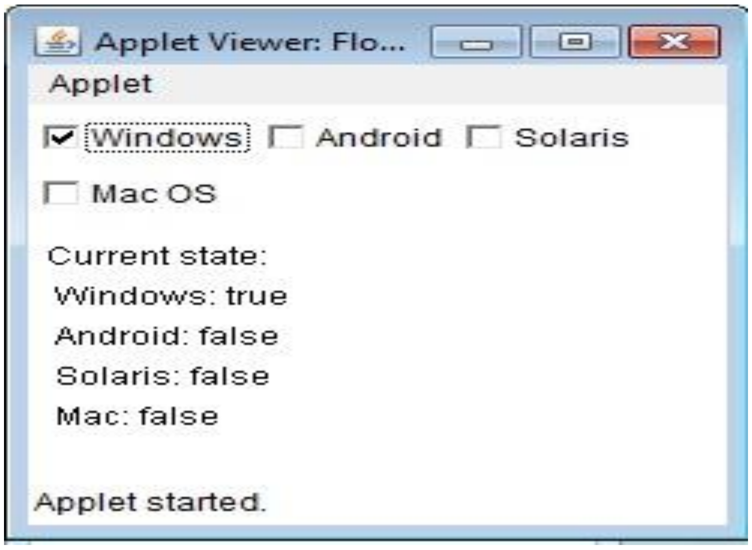
The first form creates the default layout, which centers components and leaves five pixels of space between each component.

The second form lets you specify how each line is aligned. Valid values for *how* are as follows:

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

The third constructor allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively.

```
// Use left-aligned flow layout.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="FlowLayoutDemo" width=240 height=200>   </applet>
*/
public class FlowLayoutDemo extends Applet implements ItemListener {
        String msg = "";
        Checkbox windows, android, solaris, mac;
        public void init() {
                // set left-aligned flow layout
                setLayout(new FlowLayout(FlowLayout.LEFT));
                windows = new Checkbox("Windows", null, true);
                android = new Checkbox("Android");
                solaris = new Checkbox("Solaris");
                mac = new Checkbox("Mac OS");
                add(windows);
                add(android);
                add(solaris);
                add(mac);
                // register to receive item events
                windows.addItemListener(this);
                android.addItemListener(this);
                solaris.addItemListener(this);
                mac.addItemListener(this);
        }
        // Repaint when status of a check box changes.
        public void itemStateChanged(ItemEvent ie) {
                repaint();
        }
        // Display current state of the check boxes.
        public void paint(Graphics g) {
                msg = "Current state: ";
                g.drawString(msg, 6, 80);
                msg = " Windows: " + windows.getState();
                g.drawString(msg, 6, 100);
                msg = " Android: " + android.getState();
                g.drawString(msg, 6, 120);
                msg = " Solaris: " + solaris.getState();
                g.drawString(msg, 6, 140);
                msg = " Mac: " + mac.getState();
                g.drawString(msg, 6, 160);
        }
}
```



```
Current state:
 Windows: true
 Android: false
 Solaris: false
 Mac: false
```

Applet started.

## BorderLayout

The **BorderLayout** class implements a common layout style for top-level windows. It has four narrow, fixed-width components at the edges and one large area in the center.

The four sides are referred to as north, south, east, and west. The middle area is called the center. Here are the constructors defined by **BorderLayout**:

BorderLayout( )
BorderLayout(int *horz*, int *vert*)

The first form creates a default border layout.

The second allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively.

**BorderLayout** defines the following constants that specify the regions:

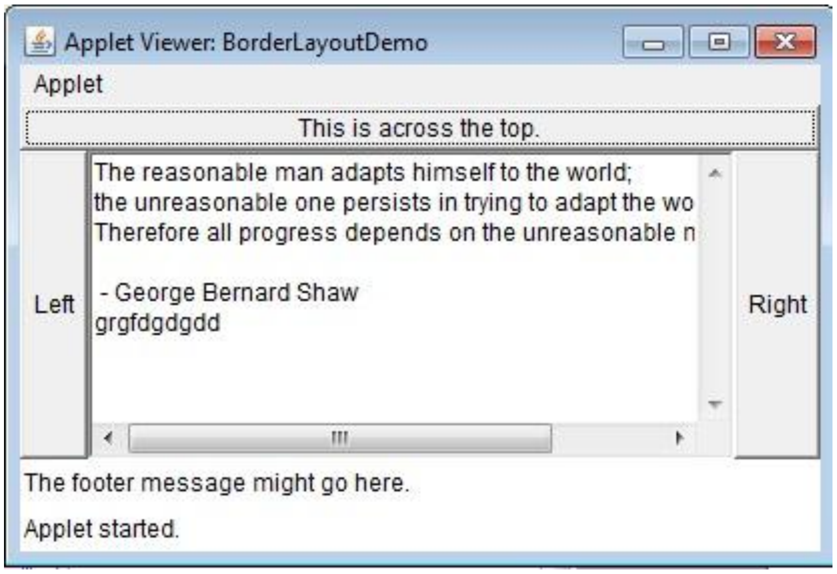
| | |
|---|---|
| BorderLayout.CENTER | BorderLayout.SOUTH |
| BorderLayout.EAST | BorderLayout.WEST |
| BorderLayout.NORTH | |

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

void add(Component *compRef*, Object *region*)

Here, *compRef* is a reference to the component to be added, and *region* specifies where the component will be added.

```
// Demonstrate BorderLayout.
import java.awt.*;
import java.applet.*;
import java.util.*;
/*
<applet code="BorderLayoutDemo" width=400 height=200> </applet>
*/
public class BorderLayoutDemo extends Applet {
    public void init() {
        setLayout(new BorderLayout());
        add(new Button("This is across the top."),
        BorderLayout.NORTH);
        add(new Label("The footer message might go here."),
        BorderLayout.SOUTH);
        add(new Button("Right"), BorderLayout.EAST);
        add(new Button("Left"), BorderLayout.WEST);
        String msg = "The reasonable man adapts " +"himself to the world;\n" +"the
unreasonable one persists in " +"trying to adapt the world to himself.\n" +"Therefore all progress
depends " +"on the unreasonable man.\n\n" +" - George Bernard Shaw\n\n";
        add(new TextArea(msg), BorderLayout.CENTER);
    }
}
```



```
The reasonable man adapts himself to the world;
the unreasonable one persists in trying to adapt the wo
Therefore all progress depends on the unreasonable n

 - George Bernard Shaw
grgfdgdgdd
```

The footer message might go here.

Applet started.

## GridLayout

**GridLayout** lays out components in a two-dimensional grid. When you instantiate a **GridLayout**, you define the number of rows and columns. The constructors supported by **GridLayout** are shown here:

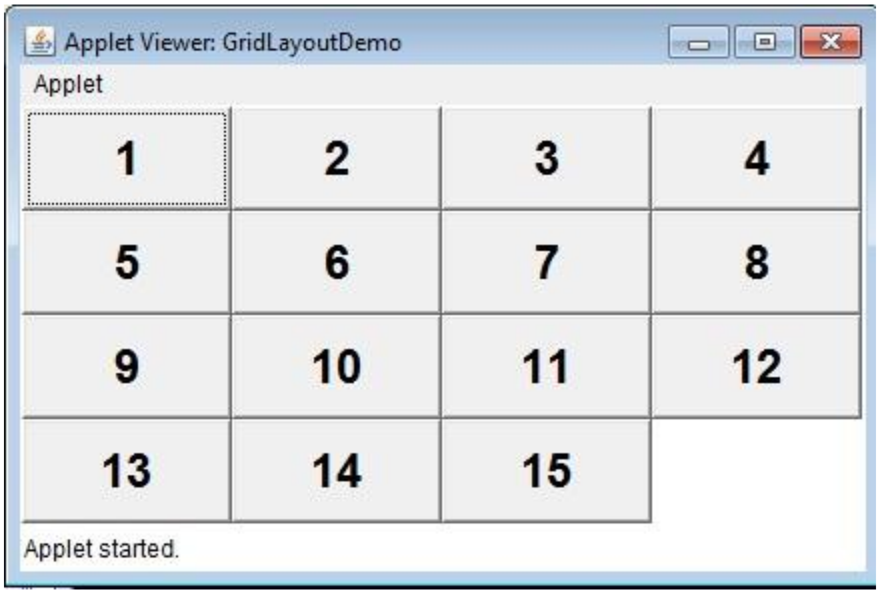
GridLayout( )
GridLayout(int *numRows*, int *numColumns*)
GridLayout(int *numRows*, int *numColumns*, int *horz*, int *vert*)

The first form creates a single-column grid layout.

The second form creates a grid layout with the specified number of rows and columns. The third form allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively. Either *numRows* or *numColumns* can be zero. Specifying *numRows* as zero allows for unlimitedlength columns. Specifying *numColumns* as zero allows for unlimited-length rows.

// Demonstrate GridLayout

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



**CardLayout**

The **CardLayout** class is unique among the other layout managers in that it stores several different layouts. Each layout can be thought of as being on a separate index card in a deck that can be shuffled so that any card is on top at a given time. **CardLayout** provides these two constructors:

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

The first form creates a default card layout.

The second form allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively.

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.

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 *panelRef*, Object *name*)

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

After you have created a deck, your program activates a card by calling one of the following methods defined by **CardLayout**:

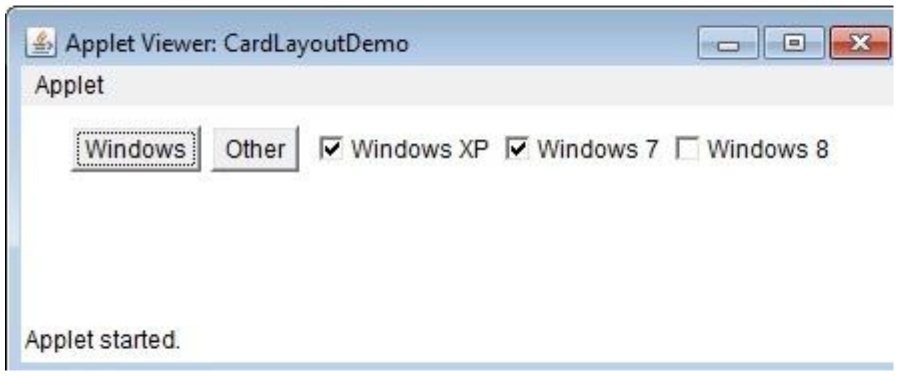
void first(Container *deck*)
void last(Container *deck*)
void next(Container *deck*)
void previous(Container *deck*)
void show(Container *deck*, String *cardName*)

Here, *deck* is a reference to the container (usually a panel) that holds the cards, and *cardName* is the name of a card. The **show( )** method displays the card whose name is passed in *cardName*.

```
// 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 windowsXP, windows7, windows8, android, 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
                windowsXP = new Checkbox("Windows XP", null, true);
                windows7 = new Checkbox("Windows 7", null, false);
                windows8 = new Checkbox("Windows 8", null, false);
```

```
		android = new Checkbox("Android");
		solaris = new Checkbox("Solaris");
		mac = new Checkbox("Mac OS");
		// add Windows check boxes to a panel
		Panel winPan = new Panel();
		winPan.add(windowsXP);
		winPan.add(windows7);
		winPan.add(windows8);
		// Add other OS check boxes to a panel
		Panel otherPan = new Panel();
		otherPan.add(a ndroid);
		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);
	}
	// Cycle through panels.
	public void mousePressed(MouseEvent me) {
		cardLO.next(osCards);
	}
	// Provide empty implementations for the other MouseListener methods.
	public void mouseClicked(MouseEvent me) {
	}
	public void mouseEntered(MouseEvent me) {
	}
	public void mouseExited(MouseEvent me) {
	}
	public void mouseReleased(MouseEvent me) {
	}
	public void actionPerformed(ActionEvent ae) {
		if(ae.getSource() == Win) {
			cardLO.show(osCards, "Windows");
		}
		else {
			cardLO.show(osCards, "Other");
		}
```

```
        }
}
```



## GridBagLayout

By using grid bag is that you can specify the relative placement of components by specifying their positions within cells inside a grid. The key to the grid bag is that each component can be a different size, and each row in the grid can have a different number of columns.

The location and size of each component in a grid bag are determined by a set of constraints linked to it. The constraints are contained in an object of type **GridBagConstraints**. Constraints include the height and width of a cell, and the placement of a component, its alignment, and its anchor point within the cell.

The general procedure for using a grid bag is to first create a new **GridBagLayout** object and to make it the current layout manager. Then, set the constraints that apply to each component that will be added to the grid bag. Finally, add the components to the layout manager.

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

GridBagLayout( )

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

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

The key to successfully using **GridBagLayout** is the proper setting of the constraints, which are stored in a **GridBagConstraints** object. **GridBagConstraints** defines several fields that you can set to govern the size, placement, and spacing of a component.

| **Field** | **Purpose** |
|---|---|
| int anchor | Specifies the location of a component within a cell. The default is **GridBagConstraints.CENTER**. |
| int fill | Specifies how a component is resized if the component is smaller than its cell. Valid values are **GridBagConstraints.NONE** (the default), **GridBagConstraints.HORIZONTAL**, **GridBagConstraints.VERTICAL**, **GridBagConstraints.BOTH**. |
| int gridheight | Specifies the height of component in terms of cells. The default is 1. |
| int | Specifies the width of component in terms of cells. The default is 1. |

| | |
|---|---|
| gridwidth | |
| int gridx | Specifies the X coordinate of the cell to which the component will be added. The default value is **GridBagConstraints.RELATIVE**. |
| int gridy | Specifies the Y coordinate of the cell to which the component will be added. The default value is **GridBagConstraints.RELATIVE**. |
| Insets insets | Specifies the insets. Default insets are all zero. |
| int ipadx | Specifies extra horizontal space that surrounds a component within a cell. The default is 0. |
| int ipady | Specifies extra vertical space that surrounds a component within a cell. The default is 0. |
| double weightx | Specifies a weight value that determines the horizontal spacing between cells and the edges of the container that holds them. The default value is 0.0. The greater the weight, the more space that is allocated. If all values are 0.0, extra space is distributed evenly between the edges of the window |
| double weighty | Specifies a weight value that determines the vertical spacing between cells and the edges of the container that holds them. The default value is 0.0. The greater the weight, the more space that is allocated. If all values are 0.0, extra space is distributed evenly between the edges of the window |

**GridBagConstraints** also defines several static fields that contain standard constraint values, such as **GridBagConstraints.CENTER** and **GridBagConstraints.VERTICAL**.

When a component is smaller than its cell, you can use the **anchor** field to specify where within the cell the component's top-left corner will be located. There are three types of values that you can give to **anchor**. The first are absolute:

| | |
|---|---|
| GridBagConstraints.CENTER | GridBagConstraints.SOUTH |
| GridBagConstraints.EAST | GridBagConstraints.SOUTHEAST |
| GridBagConstraints.NORTH | GridBagConstraints.SOUTHWEST |
| GridBagConstraints.NORTHEAST | GridBagConstraints.WEST |
| GridBagConstraints.NORTHWEST | |

The second type of values that can be given to **anchor** is relative, which means the values are relative to the container's orientation, which might differ for non-Western languages. The relative values are shown here:

| | |
|---|---|
| GridBagConstraints.FIRST_LINE_END | GridBagConstraints.LINE_END |
| GridBagConstraints.FIRST_LINE_START | GridBagConstraints.LINE_START |
| GridBagConstraints.LAST_LINE_END | GridBagConstraints.PAGE_END |
| GridBagConstraints.LAST_LINE_START | GridBagConstraints.PAGE_START |

The third type of values that can be given to **anchor** allows you to position components relative to the baseline of the row. These values are shown here:

| | |
|---|---|
| GridBagConstraints.BASELINE | GridBagConstraints.BASELINE_LEADING |
| GridBagConstraints.BASELINE_TRAILING | GridBagConstraints.ABOVE_BASELINE |
| GridBagConstraints.ABOVE_BASELINE_LEADING | GridBagConstraints.ABOVE_BASELINE_TRAILING |

| | |
|---|---|
| GridBagConstraints.BELOW_BASELINE | GridBagConstraints.BELOW_BASELINE_ LEADING |
| GridBagConstraints. BELOW_BASELINE_TRAILING | |

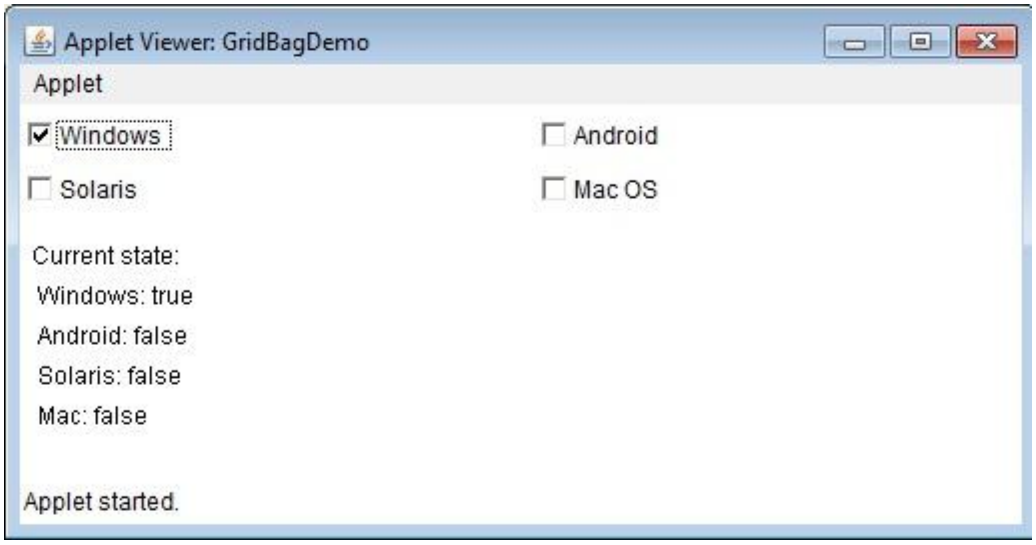
The horizontal position can be either centered, against the leading edge (LEADING), or against the trailing edge (TRAILING).

The weightx and weighty fields are both quite important and quite confusing at first glance. In general, their values determine how much of the extra space within a container is allocated to each row and column. By default, both these values are zero. When all values within a row or a column are zero, extra space is distributed evenly between the edges of the window. By increasing the weight, you increase that row or column's allocation of space proportional to the other rows or columns. The best way to understand how these values work is to experiment with them a bit.

The gridwidth variable lets you specify the width of a cell in terms of cell units. The default is 1. To specify that a component use the remaining space in a row, use GridBagConstraints.REMAINDER. To specify that a component use the next-to-last cell in a row, use GridBagConstraints.RELATIVE. The gridheight constraint works the same way, but in the vertical direction.

You can specify a padding value that will be used to increase the minimum size of a cell. To pad horizontally, assign a value to ipadx. To pad vertically, assign a value to ipady.

```
// Use GridBagLayout.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="GridBagDemo" width=250 height=200>  </applet>
*/
public class GridBagDemo extends Applet implements ItemListener {
        String msg = "";
        Checkbox windows, android, solaris, mac;
        public void init() {
                GridBagLayout gbag = new GridBagLayout();
                GridBagConstraints gbc = new GridBagConstraints();
                setLayout(gbag);
                // Define check boxes.
                windows = new Checkbox("Windows ", null, true);
                android = new Checkbox("Android");
                solaris = new Checkbox("Solaris");
                mac = new Checkbox("Mac OS");
                // Define the grid bag.
                // Use default row weight of 0 for first row.
                gbc.weightx = 1.0; // use a column weight of 1
                gbc.ipadx = 200; // pad by 200 units
                gbc.insets = new Insets(4, 4, 0, 0); // inset slightly from top left
                gbc.anchor = GridBagConstraints.NORTHEAST;
                gbc.gridwidth = GridBagConstraints.RELATIVE;
                gbag.setConstraints(windows, gbc);
                gbc.gridwidth = GridBagConstraints.REMAINDER;
                gbag.setConstraints(android, gbc);
                // Give second row a weight of 1.
                gbc.weighty = 1.0;
                gbc.gridwidth = GridBagConstraints.RELATIVE;
                gbag.setConstraints(solaris, gbc);
                gbc.gridwidth = GridBagConstraints.REMAINDER;
                gbag.setConstraints(mac, gbc);
                // Add the components.
                add(windows);
                add(android);
                add(solaris);
                add(mac);
                // Register to receive item events.
                windows.addItemListener(this);
                android.addItemListener(this);
                solaris.addItemListener(this);
                mac.addItemListener(this);
        }
        // Repaint when status of a check box changes.
        public void itemStateChanged(ItemEvent ie) {
                repaint();
        }
        // Display current state of the check boxes.
        public void paint(Graphics g) {
                msg = "Current state: ";
                g.drawString(msg, 6, 80);
                msg = " Windows: " + windows.getState();
                g.drawString(msg, 6, 100);
                msg = " Android: " + android.getState();
                g.drawString(msg, 6, 120);
                msg = " Solaris: " + solaris.getState();
                g.drawString(msg, 6, 140);
                msg = " Mac: " + mac.getState();
                g.drawString(msg, 6, 160);
        }
}
```



Current state:
Windows: true
Android: false
Solaris: false
Mac: false

Applet started.

## 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.

It is also possible to include checkable menu items. These are menu options of type **CheckboxMenuItem** and will have a check mark next to them when they are selected.

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

Following are the constructors for **Menu**:

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

Here, *optionName* specifies the name of the menu selection. If *removable* is **true**, the menu can be removed and allowed to float free. Otherwise, it will remain attached to the menu bar. (Removable menus are implementation-dependent.)

The first form creates an empty menu.Individual menu items are of type **MenuItem**. It defines these constructors:

MenuItem( ) throws HeadlessException
MenuItem(String *itemName*) throws HeadlessException
MenuItem(String *itemName*, MenuShortcut *keyAccel*) throws HeadlessException

Here, *itemName* is the name shown in the menu, and *keyAccel* is the menu shortcut for this item.

You can disable or enable a menu item by using the **setEnabled( )** method. Its form is shown here:

void setEnabled(boolean *enabledFlag*)

If the argument *enabledFlag* is **true**, the menu item is enabled. If **false**, the menu item is disabled.

You can determine an item's status by calling **isEnabled( )**. This method is shown here:

boolean isEnabled( )

**isEnabled( )** returns **true** if the menu item on which it is called is enabled. Otherwise, it returns **false**.

You can change the name of a menu item by calling **setLabel( )**. You can retrieve the current name by using **getLabel( )**. These methods are as follows:

void setLabel(String *newName*)
String getLabel( )

Here, *newName* becomes the new name of the invoking menu item. **getLabel( )** returns the current name.

You can create a checkable menu item by using a subclass of **MenuItem** called **CheckboxMenuItem**. It has these constructors:

CheckboxMenuItem( ) throws HeadlessException
CheckboxMenuItem(String *itemName*) throws HeadlessException
CheckboxMenuItem(String *itemName*, boolean *on*) throws HeadlessException

Here, *itemName* is the name shown in the menu. Checkable items operate as toggles. Each time one is selected, its state changes.

In the first two forms, the checkable entry is unchecked. In the third form, if *on* is **true**, the checkable entry is initially checked. Otherwise, it is cleared.

You can obtain the status of a checkable item by calling **getState( )**. You can set it to a known state by using **setState( )**. These methods are shown here:

boolean getState( )
void setState(boolean *checked*)

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

MenuItem add(MenuItem *item*)

Here, *item* is the item being added. Items are added to a menu in the order in which the calls to **add( )** take place. The *item* is returned.

Once you have added all items to a **Menu** object, you can add that object to the menu bar by using this version of **add( )** defined by **MenuBar**:

Menu add(Menu *menu*)

Here, *menu* is the menu being added. The *menu* is returned.

Menus generate events only when an item of type **MenuItem** or **CheckboxMenuItem** is selected. They do not generate events when a menu bar is accessed to display a drop-down menu, for example. Each time a menu item is selected, an **ActionEvent** object is generated.

By default, the action command string is the name of the menu item. However, you can specify a different action command string by calling **setActionCommand( )** on the menu item. Each time a check box menu item is checked or unchecked, an **ItemEvent** object is generated. Thus, you must implement the **ActionListener** and/or **ItemListener** interfaces in order to handle these menu events.

The **getItem( )** method of **ItemEvent** returns a reference to the item that generated this event. The general form of this method is shown here:

```
Object getItem( )
```

```
// Illustrate menus.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="MenuDemo" width=250 height=250> </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);
        }
}
```

## Dialog Boxes

Dialog boxes are primarily used to obtain user input and are often child windows of a top-level window. Dialog boxes don’t have menu bars, but in other respects, they function like frame windows.

Dialog boxes may be modal or modeless.

When a *modal* dialog box is active, all input is directed to it until it is closed. This means that you cannot access other parts of your program until you have closed the dialog box.

When a *modeless* dialog box is active, input focus can be directed to another window in your program. Thus, otherparts of your program remain active and accessible. Dialog boxes are of type **Dialog**. Two commonly used constructors are shown here:

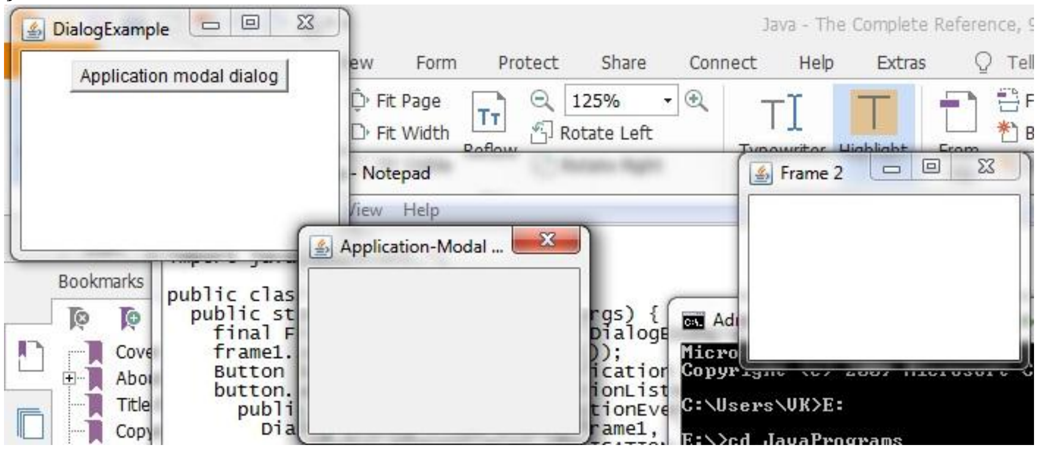
Dialog(Frame *parentWindow*, boolean *mode*)
Dialog(Frame *parentWindow*, String *title*, boolean *mode*)

Here, *parentWindow* is the owner of the dialog box. If *mode* is **true**, the dialog box is modal. Otherwise, it is modeless. The title of the dialog box can be passed in *title*.

Notice that when the dialog box is closed, **dispose( )** is called. This method is defined by **Window**, and it frees all system resources associated with the dialog box window.

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

public class DialogExample {
        public static void main(String[] args) {
                final Frame frame1 = new Frame("DialogExample");
                frame1.setLayout(new FlowLayout());
                Button button = new Button("Application modal dialog");
                button.addActionListener(new ActionListener() {
                    public void actionPerformed(ActionEvent e) {
                        Dialog dialog = new Dialog(frame1, "Application-Modal Dialog",
                        Dialog.ModalityType.APPLICATION_MODAL);
                        dialog.setBounds(200, 150, 200, 150);
                        dialog.setVisible(true);
                    }
                });
                frame1.add(button);
                frame1.setBounds(100, 100, 200, 150);
                frame1.setVisible(true);
                Frame frame2 = new Frame("Frame 2");
                frame2.setBounds(500, 100, 200, 150);
                frame2.setVisible(true);
                frame2.addWindowListener(new WindowAdapter() {
                    public void windowClosing(WindowEvent e) {
                        System.exit(0);
                    }
                });
        }
}
```



```
public clas
  public st
    final F
    frame1.
    Button
    button.
      publi
        Dia
```

```
rgs) {
Dialog
));
ication
ionList
tionEve
rame1,
```

```
Micro
Copyright (c) 2007 Microsoft C

C:\Users\VK>E:

E:\>cd JavaPrograms
```

## FileDialog

To create a file dialog box, instantiate an object of type **FileDialog**. This causes a file dialog box to be displayed.Usually, this is the standard file dialog box provided by the operating system. Here are three **FileDialog** constructors:

FileDialog(Frame *parent*)
FileDialog(Frame *parent*, String *boxName*)
FileDialog(Frame *parent*, String *boxName*, int *how*)

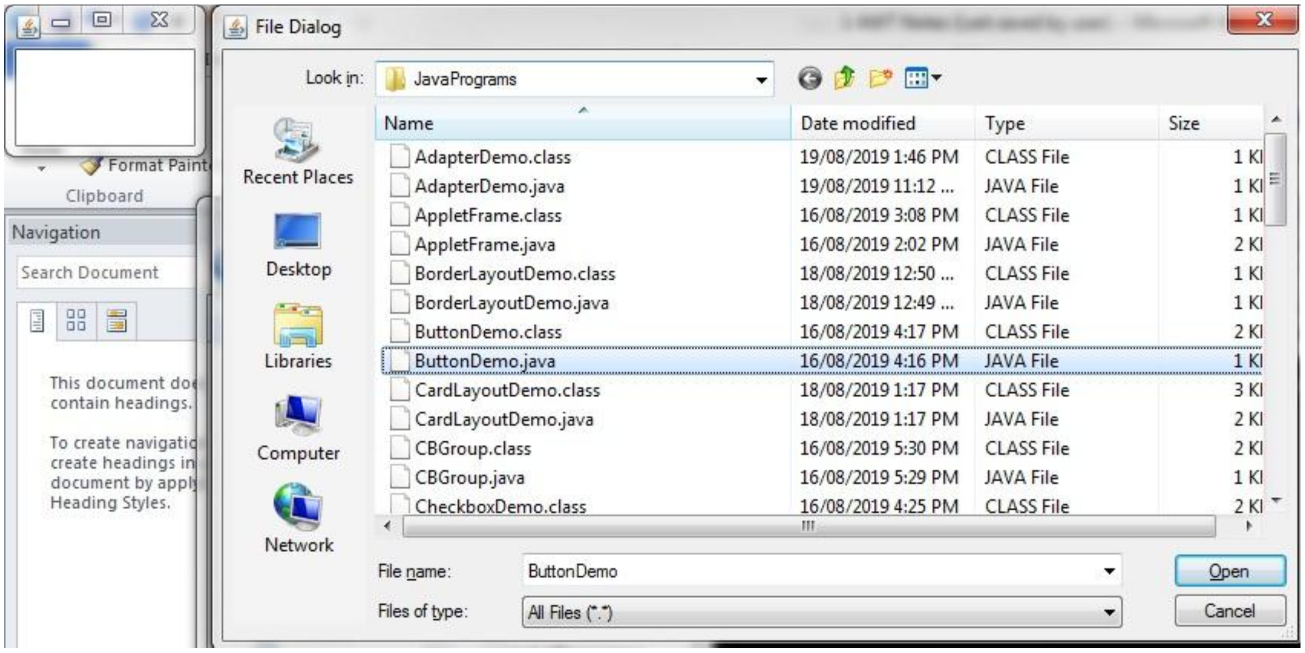
Here, *parent* is the owner of the dialog box. The *boxName* parameter specifies the name displayed in the box's title bar. If *boxName* is omitted, the title of the dialog box is empty. If *how* is **FileDialog.LOAD**, then the box is selecting a file for reading. If *how* is **FileDialog.SAVE**, the box is selecting a file for writing. If *how* is omitted, the box is selecting a file for reading.

**FileDialog** provides methods that allow you to determine the name of the file and its path as selected by the user. Here are two examples:

String getDirectory( )
String getFile( )

These methods return the directory and the filename, respectively.

```
// Demonstrate File Dialog box.This is an application, not an applet.
import java.awt.*;
import java.awt.event.*;
// Create a subclass of Frame.
class SampleFrame extends Frame {
        SampleFrame(String title) {
                super(title);
                // remove the window when closed
                addWindowListener(new WindowAdapter() {
                        public void windowClosing(WindowEvent we) {
                                System.exit(0);
                        }
                });
        }
}
// Demonstrate FileDialog.
class FileDialogDemo {
        public static void main(String args[]) {
                // create a frame that owns the dialog
                Frame f = new SampleFrame("File Dialog Demo");
                f.setVisible(true);
                f.setSize(100, 100);
                FileDialog fd = new FileDialog(f, "File Dialog");
                fd.setVisible(true);
        }
}
```



| Name | Date modified | Type | Size |
|---|---|---|---|
| AdapterDemo.class | 19/08/2019 1:46 PM | CLASS File | 1 KI |
| AdapterDemo.java | 19/08/2019 11:12 ... | JAVA File | 1 KI |
| AppletFrame.class | 16/08/2019 3:08 PM | CLASS File | 1 KI |
| AppletFrame.java | 16/08/2019 2:02 PM | JAVA File | 2 KI |
| BorderLayoutDemo.class | 18/08/2019 12:50 ... | CLASS File | 1 KI |
| BorderLayoutDemo.java | 18/08/2019 12:49 ... | JAVA File | 1 KI |
| ButtonDemo.class | 16/08/2019 4:17 PM | CLASS File | 2 KI |
| ButtonDemo.java | 16/08/2019 4:16 PM | JAVA File | 1 KI |
| CardLayoutDemo.class | 18/08/2019 1:17 PM | CLASS File | 3 KI |
| CardLayoutDemo.java | 18/08/2019 1:17 PM | JAVA File | 2 KI |
| CBGroup.class | 16/08/2019 5:30 PM | CLASS File | 2 KI |
| CBGroup.java | 16/08/2019 5:29 PM | JAVA File | 1 KI |
| CheckboxDemo.class | 16/08/2019 4:25 PM | CLASS File | 2 KI |

File name: ButtonDemo
Files of type: All Files (*.*)