**vIntroduction to Abstract Windowing Toolkit(AWT) & Swings**

**Introduction to AWT:**

The Abstract Window Toolkit (AWT) contains numerous classes and methods that allow you to create and manage windows. AWT is used to create Applet as well as Stand-alone windows(Frame).

**AWT Classes:**

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

Following table lists some of the 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-independentfashion.

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.

Event Encapsulates events.

EventQueue Queues events.

FileDialog Creates a window from which a file can be selected.

FlowLayout The flow layout manager. Flow layout positionscomponents left to right, top to bottom.

Font Encapsulates a type font.

FontMetrics Encapsulates various information related to a font. Thisinformation 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 GridBagLayoutclass.

GridBagLayout The grid bag layout manager. Grid bag layout displays components subject to the constraints specified byGridBagConstraints.

GridLayout The grid layout manager. Grid layout displays componentsin 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 andy.

Polygon Encapsulates a polygon.

PopupMenu Encapsulates a pop-up menu.

PrintJob A n abstract class that represents a print job.

Rectangle Encapsulates a rectangle.

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, scrollbars, 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.

**Window Fundamentals**

The AWT defines windows according to a class hierarchy that adds functionality and

specificity with each level. The two most common windows are those derived from **Panel**,

which is used by applets, and those derived from **Frame**, which creates a standard

window. Much of the functionality of these windows is derived from their parent classes.

****

**Figure :** The class hierarchy for Panel and Frame

**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. All user interface

elements that are displayed on the screen and that interact with the user are subclasses

of **Component**. It defines over a hundred public methods that are responsible for

managing events, such as mouse and keyboard input, positioning and sizing the window,

and repainting. 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. Other **Container** objects can be stored

inside of a **Container** (since they are themselves instances of **Component**). This makes

for a multileveled containment system. 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**. It doesn't add any new methods; it

simply implements **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.

Other components can be added to a **Panel** object by its **add( )** method (inherited from

**Container**). Once these components have been added, you can position and resize them

manually using the **setLocation( )**, **setSize( )**, or **setBounds( )** methods defined by

**Component**.

**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**, described

next.

**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. If you create a

**Frame** object from within an applet, it will contain a warning message, such as "Warning:

Applet Window," to the user that an applet window has been created. This message

warns users that the window they see was started by an applet and not by software

running on their computer. (An applet that could masquerade as a host-based application

could be used to obtain passwords and other sensitive information without the user's

knowledge.) When a **Frame** window is created by a program rather than an applet, a

normal window is created.

**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**. **Canvas** encapsulates a blank window upon

which you can draw.

**Working with Frame Windows**

After the applet, the type of window we will most often create is derived from **Frame**.

We will use it to create child windows within applets, and top-level or child windows for

applications.

**Frame** supports these two constructors:

Frame( )

Frame(String title)

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.*

There are several methods we will use when working with **Frame** windows. They are given as bellow

**Setting the Window's Dimensions**

The **setSize( )** method is used to set the dimensions of the window. Its signature is

shown here:

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. Its signature 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.

**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.

**Closing a Frame Window**

When using a frame window, your program must remove that window from the screen

when it is closed, by calling **setVisible(false)**. To intercept a window-close event, you must

implement the **windowClosing( )** method of the **WindowListener** interface. Inside

**windowClosing( )**, you must remove the window from the screen.

**Creating a Frame Window in an Applet**

For creating a Frame window within an Applet ,First, create a

subclass of **Frame**. Next, override any of the standard window methods, such as **init( )**,

**start( )**, **stop( )**, and **paint( )**. Finally, implement the **windowClosing( )** method of the

**WindowListener** interface, calling **setVisible(false)** when the window is closed.

Once you have defined a **Frame** subclass, you can create an object of that class. This

causes a frame window to come into existence, but it will not be initially visible. You make

it visible by calling **setVisible( )**. When created, the window is given a default height and

width. You can set the size of the window explicitly by calling the **setSize( )** method.

Example:

import java.awt.\*;

import java.applet.\*;

import java.awt.event.\*;

/\*<applet code=”frinapp” width=400 height=400></applet>\*/

class fr extends Frame

{

fr(String title)

{

super(title);

setSize(200,200);

setVisible(true);

addWindowListener(new WindowAdapter()

{

public void windowClosing(WindowEvent e)

{

setVisible(false);

}

});

}

public void paint(Graphics g)

{

g.drawString(“Sample frame window called from applet”,100,100);

}

}

public class frinapp extends Applet

{

Public void init()

{

fr f=new fr(“Demo frame”);

}

}

**Creating a Windowed Program**

Although creating applets is the most common use for Java's AWT, it is possible to create

stand-alone AWT-based applications, too. To do this, simply create an instance of the

window or windows you need inside **main( )**.

Example:-

import java.awt.\*;

import java.awt.event.\*;

class standaloneframe extends Frame

{

standaloneframe(String title)

{

super(title);

setSize(400,400);

setVisible(true);

addWindowListener(new WindowAdapter()

{

public void windowClosing(WindowEvent e)

{

System.exit(0);//When frame window closed program execution also terminates

}

});

}

public void paint(Graphics g)

{

g.drawOval(100,100,50,50);

}

public static void main(String ar[])

{

standaloneframe f=new standaloneframe("Top level window");

}

}

**AWT controls & 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

**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**.

**Adding and Removing Controls**

To include a control in a window, you must add it to the window. To do this, you must first

create an instance of the desired control and then add it to a window by calling **add( )**,

which is defined by **Container**. The **add( )** method has several forms. The following form

is the one that is used for the first part of this chapter:

Component add(Component compObj)

Here, *compObj* is an instance of the control that you want to add. A reference to *compObj*

is returned. Once a control has been added, it will automatically be visible whenever its

parent window is displayed.

Sometimes you will want to remove a control from a window when the control is no longer

needed. To do this, call **remove( )**. This method is also defined by **Container**. It has this

general form:

void remove(Component obj)

Here, *obj* is a reference to the control you want to remove. You can remove all controls

by calling **removeAll( )**.

**Responding to Controls**

Except for labels, which are passive controls, all controls generate events when they are

accessed by the user. For example, when the user clicks on a push button, an event is

sent that identifies the push button. In general, your program simply implements the

appropriate interface and then registers an event listener for each control that you need to

monitor.

once a listener has been installed, events areautomatically sent to it. In the sections that follow, the appropriate interface for each controlis specified.

**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( )

Label(String str)

Label(String str, int how)

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( )

Example:

import java.awt.\*;

import java.applet.\*;

/\*<applet code="labeldemo" width=400 height=400>

</applet>\*/

public class labeldemo extends Applet

{

Label l1,l2,l3;

public void init()

{

l1=new Label();

l1.setBackground(Color.BLUE);

l2=new Label("it is the label2");

l2.setBackground(Color.RED);

l3=new Label("it is the label3",Label.CENTER);

l3.setBackground(Color.GREEN);

add(l1);

add(l2);

add(l3);

}

}

**Buttons**

The most widely used control is the push button. A *push button* is a component that

contains a label and that generates an event when it is pressed. Push buttons are objects

of type **Button**. **Button** defines these two constructors:

Button( )

Button(String str)

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.

Buttons generate ActionEvent when it is pressed. So to handle ActionEvent we need to register Buttons with the ActionListener. Then we need to provide actionPerformed() method definition to handle ActionEvent.

Example:

import java.awt.\*;

import java.applet.\*;

/\*<applet code="buttondemo" width=400 height=400>

</applet>\*/

public class buttondemo extends Applet

{

Button b1,b2;

public void init()

{

b1=new Button();

b1.setBackground(Color.BLUE);

b2=new Button("Ok");

b2.setBackground(Color.RED);

add(b1);

add(b2);

}

}

**CheckBoxes:**

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.

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( )

Checkbox(String str)

Checkbox(String str, boolean on)

Checkbox(String str, boolean on, CheckboxGroup cbGroup)

Checkbox(String str, CheckboxGroup cbGroup, boolean on)

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.

Checkboxes generates ItemEvent when selected or deselected. So to handle ItemEvent we need to register checkboxes with ItemListener and we need to provide definition for itemStateChanged() method.

Example:

import java.awt.\*;

import java.applet.\*;

/\*<applet code="checkboxdemo" width=400 height=400>

</applet>\*/

public class checkboxdemo extends Applet

{

Checkbox c1,c2,c3;

public void init()

{

c1=new Checkbox();

c2=new Checkbox("Reading hobby");

c3=new Checkbox("Singing hobby",true);

add(c1);

add(c2);

add(c3);

}

}

**CheckboxGroup:**

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.

Example:

import java.awt.\*;

import java.applet.\*;

/\*<applet code="checkboxgrdemo" width=400 height=400>

</applet>\*/

public class checkboxgrdemo extends Applet

{

Label l=new Label("Gender");

Checkbox c1,c2;

CheckboxGroup cb=new CheckboxGroup();

public void init()

{

c1=new Checkbox("Male",cb,true);

c2=new Checkbox("Female",cb,false);

add(l);

add(c1);

add(c2);

}

}

**Choice Control**

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 **addItem( )** or **add( )**. They have these general forms:

void addItem(String name)

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( )** or **addItem( )** occur.

To determine which item is currently selected, you may call either **getSelectedItem( )** or

**getSelectedIndex( )**. These methods are shown here:

String getSelectedItem( )

int getSelectedIndex( )

The **getSelectedItem( )** method returns a string containing the name of the item.

**getSelectedIndex( )** returns the index of the item. The first item is at index 0. By default,

the first item added to the list is selected.

To obtain the number of items in the list, call **getItemCount( )**. You can set the currently

selected item using the **select( )** method with either a zero-based integer index or a string

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

int getItemCount( )

void select(int index)

void select(String name)

Given an index, you can obtain the name associated with the item at that index by calling

**getItem( )**, which has this general form:

String getItem(int index)

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

Each time a choice is selected, an item event is generated. This is sent to any listeners

that previously registered an interest in receiving item event notifications from that

component. Each listener implements the **ItemListener** interface. That interface defines

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

this method.

**Example:**

import java.awt.\*;

import java.awt.event.\*;

class choicedemo extends Frame

{

Choice ch=new Choice();

choicedemo(String title)

{

super(title);

setSize(400,400);

setVisible(true);

addWindowListener(new WindowAdapter()

{

public void windowClosing(WindowEvent e)

{

System.exit(0);

}

});

ch.add("AJP");

ch.add("MAN");

ch.add("OMD");

ch.add("STG");

add(ch);

}

public static void main(String ar[])

{

choicedemo cd=new choicedemo("Choice control demo");

}

}

**Using Lists**

The **List** class provides a compact, multiple-choice, scrolling selection list. Unlike the

**Choice** object, which shows only the single selected item in the menu, a **List** object can

be constructed to show any number of choices in the visible window. It can also be

created to allow multiple selections. **List** provides these constructors:

List( )

List(int numRows)

List(int numRows, boolean multipleSelect)

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.

To process list events, you will need to implement the **ActionListener** interface. Each

time a **List** item is double-clicked, an **ActionEvent** object is generated. Its

**getActionCommand( )** method can be used to retrieve the name of the newly selected

item. Also, each time an item is selected or deselected with a single click, an **ItemEvent**

object is generated. Its **getStateChange( )** method can be used to determine whether a

selection or deselection triggered this event. **getItemSelectable( )** returns a reference to

the object that triggered this event.

**Example:**

import java.awt.\*;

import java.awt.event.\*;

class listdemo extends Frame

{

List l=new List(3,true);

listdemo(String title)

{

super(title);

setSize(400,400);

setVisible(true);

addWindowListener(new WindowAdapter()

{

public void windowClosing(WindowEvent e)

{

System.exit(0);

}

});

l.add("AJP");

l.add("MAN");

l.add("OMD");

l.add("STG");

add(l);

}

public static void main(String ar[])

{

listdemo cd=new listdemo("List control demo");

}

}

**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( )

Scrollbar(int style)

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

The first form creates a vertical scroll bar. The second and third forms allow you to

specify the orientation of the scroll bar. If *style* is **Scrollbar.VERTICAL**, a vertical scroll

bar is created. If *style* is **Scrollbar.HORIZONTAL**, the scroll bar is horizontal. In the third

form of the constructor, the initial value of the scroll bar is passed in *initialValue.* The

number of units represented by the height of the thumb is passed in *thumbSize.* The

minimum and maximum values for the scroll bar are specified by *min* and *max.*

If you construct a scroll bar by using one of the first two constructors, then you need to

set its parameters by using **setValues( )**, shown here, before it can be used:

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

The parameters have the same meaning as they have in the third constructor just described.

To obtain the current value of the scroll bar, call **getValue( )**. It returns the current setting.

To set the current value, call **setValue( )**. These methods are as follows:

int getValue( )

void setValue(int newValue)

Here, *newValue* specifies the new value for the scroll bar. When you set a value, the

slider box inside the scroll bar will be positioned to reflect the new value.

You can also retrieve the minimum and maximum values via **getMinimum( )** and

**getMaximum( )**, shown here:

int getMinimum( )

int getMaximum( )

They return the requested quantity.

By default, 1 is the increment added to or subtracted from the scroll bar each time it is

scrolled up or down one line. You can change this increment by calling

**setUnitIncrement( )**. By default, page-up and page-down increments are 10. You can

change this value by calling **setBlockIncrement( )**. These methods are shown here:

void setUnitIncrement(int newIncr)

void setBlockIncrement(int newIncr)

To process scroll bar events, you need to implement the **AdjustmentListener** interface.

Each time a user interacts with a scroll bar, an **AdjustmentEvent** object is generated. Its

**getAdjustmentType( )** method can be used to determine the type of the adjustment. The

types of adjustment events are as follows:

BLOCK\_DECREMENT A page-down event has been generated.

BLOCK\_INCREMENT A page-up event has been generated.

TRACK An absolute tracking event has been generated.

UNIT\_DECREMENT The line-down button in a scroll bar has been pressed.

UNIT\_INCREMENT The line-up button in a scroll bar has been pressed

**Example:**

import java.awt.\*;

import java.applet.\*;

/\*<applet code=scrolldemo width=400 height=400>

</applet>\*/

public class scrolldemo extends Applet

{

Scrollbar h=new Scrollbar(Scrollbar.HORIZONTAL,10,10,0,200);

Scrollbar v=new Scrollbar(Scrollbar.VERTICAL,10,30,0,200);

public void init()

{

add(h);

add(v);

}

}

**Using a 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( )

TextField(int numChars)

TextField(String str)

TextField(String str, int numChars)

The first version creates a default text field. The second form creates a text field that is

*numChars* characters wide. The third form initializes the text field with the string

contained in *str.* The fourth form initializes a text field and sets its width.

**TextField** (and its superclass **TextComponent**) provides several methods that allow you

to utilize a text field. To obtain the string currently contained in the text field, call **getText(**

**)**. To set the text, call **setText( )**. These methods are as follows:

String getText( )

void setText(String str)

Here, *str* is the new string.

The user can select a portion of the text in a text field. Also, you can select a portion of

text under program control by using **select( )**. Your program can obtain the currently

selected text by calling **getSelectedText( )**. These methods are shown here:

String getSelectedText( )

void select(int startIndex, int endIndex)

**getSelectedText( )** returns the selected text. The **select( )** method selects the characters

beginning at *startIndex* and ending at *endIndex*–1*.*

You can control whether the contents of a text field may be modified by the user by

calling **setEditable( )**. You can determine editability by calling **isEditable( )**. These

methods are shown here:

boolean isEditable( )

void setEditable(boolean canEdit)

**isEditable( )** returns **true** if the text may be changed and **false** if not. In **setEditable( )**, if

*canEdit* is **true**, the text may be changed. If it is **false**, the text cannot be altered.

There may be times when you will want the user to enter text that is not displayed, such

as a password. You can disable the echoing of the characters as they are typed by

calling **setEchoChar( )**. This method specifies a single character that the **TextField** will

display when characters are entered (thus, the actual characters typed will not be

shown). You can check a text field to see if it is in this mode with the **echoCharIsSet( )**

method. You can retrieve the echo character by calling the **getEchoChar( )** method.

These methods are as follows:

void setEchoChar(char ch)

boolean echoCharIsSet( )

char getEchoChar( )

Here, *ch* specifies the character to be echoed.

TextField generates ActionEvent when we hit enter key in textfield and generates TextEvent whenever we change text inside the textfield.

import java.awt.\*;

import java.applet.\*;

/\*<applet code=textdemo width=200 height=100>

</applet>\*/

public class textdemo extends Applet

{

Label uname=new Label("User Name");

Label pass=new Label("Password");

TextField u=new TextField(20);

TextField p=new TextField(20);

public void init()

{

setLayout(new GridLayout(2,2));

p.setEchoChar('\*');

add(uname);

add(u);

add(pass);

add(p);

}

}

**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( )

TextArea(int numLines, int numChars)

TextArea(String str)

TextArea(String str, int numLines, int numChars)

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

Here, *numLines* specifies the height, in lines, of the text area, and *numChars* specifies its

width, in characters. Initial text can be specified by *str.* In the fifth form you can specify

the scroll bars that you want the control to have. *sBars* must be one of these values:

SCROLLBARS\_BOTH

SCROLLBARS\_NONE

SCROLLBARS\_HORIZONTAL\_ONLY

SCROLLBARS\_VERTICAL\_ONLY

**TextArea** is a subclass of **TextComponent**. Therefore, it supports the **getText( )**,

**setText( )**, **getSelectedText( )**, **select( )**, **isEditable( )**, and **setEditable( )** methods

described in the preceding section.

**TextArea** adds the following methods:

void append(String str)

void insert(String str, int index)

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

The **append( )** method appends the string specified by *str* to the end of the current text.

**insert( )** 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.*

Text areas only generate got-focus and lost-focus events.

import java.awt.\*;

import java.applet.\*;

/\*

<applet code="TextAreaDemo" width=300 height=250>

</applet>

\*/

public class TextAreaDemo extends Applet {

public void init() {

String val = "There are two ways of constructing " +"a software design.\\n" +"One way is to make it so simple\\n" +

"that there are obviously no deficiencies.\\n" +"And the other way is to make it so complicated\\n" +

"that there are no obvious deficiencies.\\n\\n" +" -C.A.R. Hoare\\n\\n" +"There's an old story about the person who wished\\n" +

"his computer were as easy to use as his telephone.\\n" +"That wish has come true,\\n" +"since I no longer know how to use my telephone.\\n\\n" +" -Bjarne Stroustrup, AT&T, (inventor of C++)";

TextArea text = new TextArea(val, 10, 30);

add(text);

}

}

**Understanding Layout Managers**

A layout manager automatically arranges your controls within a window by using some type of algorithm.

it is possible to lay out Java controls by hand, too, you generally won't want to, for two main reasons. First, it is very

tedious to manually lay out a large number of components. Second, sometimes the width

and height information is not yet available when you need to arrange some control,

because the native toolkit components haven't been realized.

Each **Container** object has a layout manager associated with it. A layout manager is an

instance of any class that implements the **LayoutManager** interface. The layout manager

is set by the **setLayout( )** method. If no call to **setLayout( )** is made, then the default

layout manager is used. Whenever a container is resized (or sized for the first time), the

layout manager is used to position each of the components within it.

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

**void** setLayout(LayoutManager layoutObj)

Here, *layoutObj* is a reference to the desired layout manager. If you wish to disable the

layout manager and position components manually, pass **null** for *layoutObj.* If you do

this, you will need to determine the shape and position of each component manually,

using the **setBounds( )** method defined by **Component**. Normally, you will want to use a

layout manager.

Each layout manager keeps track of a list of components that are stored by their names.

The layout manager is notified each time you add a component to a container. Whenever

the container needs to be resized, the layout manager is consulted via its

**minimumLayoutSize( )** and **preferredLayoutSize( )** methods. Each component that is

being managed by a layout manager contains the **getPreferredSize( )** and

**getMinimumSize( )** methods. These return the preferred and minimum size required to

display each component. The layout manager will honor these requests if at all possible,

while maintaining the integrity of the layout policy. You may override these methods for

controls that you subclass. Default values are provided otherwise.

Java has several predefined **LayoutManager** classes.

**FlowLayout**

**FlowLayout** is the default layout manager. This is the layout manager that the preceding

examples have used. **FlowLayout** implements a simple layout style, which is similar to

how words flow in a text editor. Components are laid out from the upper-left corner, left to

right and top to bottom. When no more components fit on a line, the next one appears on

the next line. A small space is left between each component, above and below, as well as

left and right. 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

These values specify left, center, and right alignment, respectively. The third form allows

you to specify the horizontal and vertical space left between components in *horz* and *vert,*

respectively.

**Example:**

import java.awt.\*;

import java.applet.\*;

/\*<applet code=flowdemo width=400 height=400>

</applet>\*/

public class flowdemo extends Applet

{

Button ok=new Button(“OK”);

Button cancel=new Button(“Cancel”);

public void init()

{

setLayout(new FlowLayout(FlowLayout.LEFT,10,10));

add(ok);

add(cancel);

}

}

**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

B orderLayout.SOUTH

BorderLayout.EAST

B orderLayout.WEST

BorderLayout.NORTH

When adding components, you will use these constants with the following form of **add( )**,

which is defined by **Container**:

void add(Component compObj, Object region);

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

will be added.

**Example:**

import java.awt.\*;

import java.applet.\*;

/\*<applet code=borderdemo width=400 height=400>

</applet>\*/

public class borderdemo extends Applet

{

Button e=new Button("East");

Button w=new Button("West");

Button s=new Button("South");

Button n=new Button("North");

Label l=new Label("Center");

public void init()

{

setLayout(new BorderLayout());

add(e,BorderLayout.EAST);

add(w,BorderLayout.WEST);

add(s,BorderLayout.SOUTH);

add(n,BorderLayout.NORTH);

add(l,BorderLayout.CENTER);

}

}

**Using Insets**

Sometimes you will want to leave a small amount of space between the container that

holds your components and the window that contains it. To do this, override the

**getInsets( )** method that is defined by **Container**. This function returns an **Insets** object

that contains the top, bottom, left, and right inset to be used when the container is

displayed. These values are used by the layout manager to inset the components when it

lays out the window. The constructor for **Insets** is shown here:

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

The values passed in *top*, *left*, *bottom*, and *right* specify the amount of space between the

container and its enclosing window.

The **getInsets( )** method has this general form:

Insets getInsets( )

When overriding one of these methods, you must return a new **Insets** object that

contains the inset spacing you desire.

**Example:**

import java.awt.\*;

import java.applet.\*;

/\*<applet code=borderdemo width=400 height=400>

</applet>\*/

public class borderdemo extends Applet

{

Button e=new Button("East");

Button w=new Button("West");

Button s=new Button("South");

Button n=new Button("North");

Label l=new Label("Center");

public void init()

{

setLayout(new BorderLayout());

add(e,BorderLayout.EAST);

add(w,BorderLayout.WEST);

add(s,BorderLayout.SOUTH);

add(n,BorderLayout.NORTH);

add(l,BorderLayout.CENTER);

}

public Insets getInsets() {

return new Insets(10, 10, 10, 10);

}

}

**GridLayout**

**GridLayout** lays out components in a two-dimensional grid. When you instantiate a

**GridLayout**, you define the number of rows and columns. The constructors supported by

**GridLayout** are shown here:

GridLayout( )

GridLayout(int numRows, int numColumns )

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

The first form creates a single-column grid layout. The second form creates a grid layout

with the specified number of rows and columns. The third form allows you to specify the

horizontal and vertical space left between components in *horz* and *vert*, respectively.

Either *numRows* or *numColumns* can be zero. Specifying *numRows* as zero allows for

unlimited-length columns. Specifying *numColumns* as zero allows for unlimited-length

rows.

**Example:**

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. This can be

useful for user interfaces with optional components that can be dynamically enabled and

disabled upon user input. You can prepare the other layouts and have them hidden,

ready to be activated when needed.

**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.

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 main applet panel. 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*.

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. Calling **first( )** causes the first card in the deck to be

shown. To show the last card, call **last( )**. To show the next card, call **next( )**. To show

the previous card, call **previous( )**. Both **next( )** and **previous( )** automatically cycle back

to the top or bottom of the deck, respectively. The **show( )** method displays the card

whose name is passed in *cardName.*

**Example:**

import java.applet.\*;

import java.awt.\*;

import java.awt.event.\*;

/\*<applet code=carddemo width=400 height=400>

</applet>\*/

public class carddemo extends Applet implements ActionListener

{

Button card1=new Button("Show card1");

Button card2=new Button("Show card2");

Panel main,p1,p2;

Label l1,l2;

CardLayout c;

public void init()

{c=new CardLayout();

main=new Panel();

p1=new Panel();

p2=new Panel();

main.setLayout(c);

l1=new Label(" ");

l2=new Label(" ");

l1.setBackground(Color.RED);

l2.setBackground(Color.GREEN);

p1.add(l1);

p2.add(l2);

main.add(p1,"card1");

main.add(p2,"card2");

add(main);

add(card1);add(card2);

card1.addActionListener(this);

card2.addActionListener(this);

}

public void actionPerformed(ActionEvent e)

{

if(e.getSource()==card1)

c.show(main,"card1");

else

c.show(main,"card2");

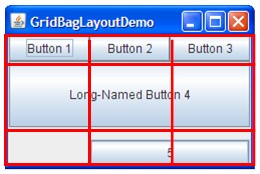
}

}

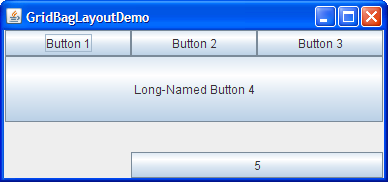
**GridBagLayout**

GridBagLayout is one of the most flexible — and complex — layout managers the Java platform provides. AGridBagLayout places components in a grid of rows and columns, allowing specified components to span multiple rows or columns. Not all rows necessarily have the same height. Similarly, not all columns necessarily have the same width. Essentially, GridBagLayout places components in rectangles (cells) in a grid, and then uses the components' preferred sizes to determine how big the cells should be.

The following figure shows the grid for the preceding applet. As you can see, the grid has three rows and three columns. The button in the second row spans all the columns; the button in the third row spans the two right columns.



If you enlarge the window as shown in the following figure, you will notice that the bottom row, which contains Button 5, gets all the new vertical space. The new horizontal space is split evenly among all the columns. This resizing behavior is based on weights the program assigns to individual components in the GridBagLayout. You will also notice that each component takes up all the available horizontal space — but not (as you can see with button 5) all the available vertical space. This behavior is also specified by the program.



The way the program specifies the size and position characteristics of its components is by specifying *constraints* for each component. The preferred approach to set constraints on a component is to use the Container.add variant, passing it aGridBagConstraints object, as demonstrated in the next sections.

The following sections explain the constraints you can set and provide examples.

## Specifying Constraints

The following code is typical of what goes in a container that uses a [GridBagLayout](https://docs.oracle.com/javase/8/docs/api/java/awt/GridBagLayout.html" \t "_blank). You will see a more detailed example in the next section.

JPanel pane = new JPanel(new GridBagLayout());

GridBagConstraints c = new GridBagConstraints();

//For each component to be added to this container:

//...Create the component...

//...Set instance variables in the GridBagConstraints instance...

pane.add(theComponent, c);

As you might have guessed from the above example, it is possible to reuse the same GridBagConstraints instance for multiple components, even if the components have different constraints. However, it is recommended that you do not reuseGridBagConstraints, as this can very easily lead to you introducing subtle bugs if you forget to reset the fields for each new instance.

**Note:** The following discussion assumes that the GridBagLayout controls a container that has a left-to-right component orientation.

You can set the following [GridBagConstraints](https://docs.oracle.com/javase/8/docs/api/java/awt/GridBagConstraints.html" \t "_blank) instance variables:

gridx, gridy

Specify the row and column at the upper left of the component. The leftmost column has address gridx=0 and the top row has address gridy=0. Use GridBagConstraints.RELATIVE (the default value) to specify that the component be placed just to the right of (for gridx) or just below (for gridy) the component that was added to the container just before this component was added. We recommend specifying the gridx and gridy values for each component rather than just using GridBagConstraints.RELATIVE; this tends to result in more predictable layouts.

gridwidth, gridheight

Specify the number of columns (for gridwidth) or rows (for gridheight) in the component's display area. These constraints specify the number of cells the component uses, not the number of pixels it uses. The default value is 1. Use GridBagConstraints.REMAINDER to specify that the component be the last one in its row (for gridwidth) or column (for gridheight). Use GridBagConstraints.RELATIVE to specify that the component be the next to last one in its row (for gridwidth) or column (for gridheight). We recommend specifying the gridwidth andgridheight values for each component rather than just using GridBagConstraints.RELATIVE andGridBagConstraints.REMAINDER; this tends to result in more predictable layouts.

**Note:** GridBagLayout does not allow components to span multiple rows unless the component is in the leftmost column or you have specified positive gridx and gridy values for the component.

fill

Used when the component's display area is larger than the component's requested size to determine whether and how to resize the component. Valid values (defined as GridBagConstraints constants) include NONE (the default),HORIZONTAL (make the component wide enough to fill its display area horizontally, but do not change its height),VERTICAL (make the component tall enough to fill its display area vertically, but do not change its width), and BOTH(make the component fill its display area entirely).

ipadx, ipady

Specifies the internal padding: how much to add to the size of the component. The default value is zero. The width of the component will be at least its minimum width plus ipadx\*2 pixels, since the padding applies to both sides of the component. Similarly, the height of the component will be at least its minimum height plus ipady\*2 pixels.

insets

Specifies the external padding of the component -- the minimum amount of space between the component and the edges of its display area. The value is specified as an [Insets](https://docs.oracle.com/javase/8/docs/api/java/awt/Insets.html) object. By default, each component has no external padding.

anchor

Used when the component is smaller than its display area to determine where (within the area) to place the component. Valid values (defined as GridBagConstraints constants) are CENTER (the default), PAGE\_START, PAGE\_END,LINE\_START, LINE\_END, FIRST\_LINE\_START, FIRST\_LINE\_END, LAST\_LINE\_END, and LAST\_LINE\_START.

Here is a picture of how these values are interpreted in a container that has the default, left-to-right component orientation.

|  |  |  |
| --- | --- | --- |
| FIRST\_LINE\_START | PAGE\_START | FIRST\_LINE\_END |
| LINE\_START | CENTER | LINE\_END |
| LAST\_LINE\_START | PAGE\_END | LAST\_LINE\_END |

**Version note:** The PAGE\_\* and \*LINE\_\* constants were introduced in 1.4. Previous releases require values named after points of the compass. For example, NORTHEAST indicates the top-right part of the display area. We recommend that you use the new constants, instead, since they enable easier localization.

weightx, weighty

Specifying weights is an art that can have a significant impact on the appearance of the components aGridBagLayout controls. Weights are used to determine how to distribute space among columns (weightx) and among rows (weighty); this is important for specifying resizing behavior.

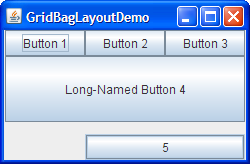
Unless you specify at least one non-zero value for weightx or weighty, all the components clump together in the center of their container. This is because when the weight is 0.0 (the default), the GridBagLayout puts any extra space between its grid of cells and the edges of the container.

Generally weights are specified with 0.0 and 1.0 as the extremes: the numbers in between are used as necessary. Larger numbers indicate that the component's row or column should get more space. For each column, the weight is related to the highest weightx specified for a component within that column, with each multicolumn component's weight being split somehow between the columns the component is in. Similarly, each row's weight is related to the highest weighty specified for a component within that row. Extra space tends to go toward the rightmost column and bottom row.

The next section discusses constraints in depth, in the context of explaining how the example program works.

## The Example Explained

Here, again, is a picture of the GridBagLayoutDemo application.



Click the Launch button to run GridBagLayoutDemo using [Java™ Web Start](http://www.oracle.com/technetwork/java/javase/javawebstart/index.html) ([download JDK 7 or later](http://www.oracle.com/technetwork/java/javase/downloads/index.html)). Alternatively, to compile and run the example yourself, consult the [example index](https://docs.oracle.com/javase/tutorial/uiswing/examples/layout/index.html#GridBagLayoutDemo).

[Launches the GridBagLayoutDemo example](https://docs.oracle.com/javase/tutorialJWS/samples/uiswing/GridBagLayoutDemoProject/GridBagLayoutDemo.jnlp)

The following code creates the GridBagLayout and the components it manages. You can find the entire source file in[GridBagLayoutDemo.java](https://docs.oracle.com/javase/tutorial/uiswing/examples/layout/GridBagLayoutDemoProject/src/layout/GridBagLayoutDemo.java).

JButton button;

pane.setLayout(new GridBagLayout());

GridBagConstraints c = new GridBagConstraints();

if (shouldFill) {

//natural height, maximum width

c.fill = GridBagConstraints.HORIZONTAL;

}

button = new JButton("Button 1");

if (shouldWeightX) {

c.weightx = 0.5;

}

c.fill = GridBagConstraints.HORIZONTAL;

c.gridx = 0;

c.gridy = 0;

pane.add(button, c);

button = new JButton("Button 2");

c.fill = GridBagConstraints.HORIZONTAL;

c.weightx = 0.5;

c.gridx = 1;

c.gridy = 0;

pane.add(button, c);

button = new JButton("Button 3");

c.fill = GridBagConstraints.HORIZONTAL;

c.weightx = 0.5;

c.gridx = 2;

c.gridy = 0;

pane.add(button, c);

button = new JButton("Long-Named Button 4");

c.fill = GridBagConstraints.HORIZONTAL;

c.ipady = 40; //make this component tall

c.weightx = 0.0;

c.gridwidth = 3;

c.gridx = 0;

c.gridy = 1;

pane.add(button, c);

button = new JButton("5");

c.fill = GridBagConstraints.HORIZONTAL;

c.ipady = 0; //reset to default

c.weighty = 1.0; //request any extra vertical space

c.anchor = GridBagConstraints.PAGE\_END; //bottom of space

c.insets = new Insets(10,0,0,0); //top padding

c.gridx = 1; //aligned with button 2

c.gridwidth = 2; //2 columns wide

c.gridy = 2; //third row

pane.add(button, c);

This example uses one GridBagConstraints instance for all the components the GridBagLayout manages, however in real-life situations it is recommended that you do not reuse GridBagConstraints, as this can very easily lead to you introducing subtle bugs if you forget to reset the fields for each new instance. Just before each component is added to the container, the code sets (or resets to default values) the appropriate instance variables in the GridBagConstraints object. It then adds the component to its container, specifying the GridBagConstraints object as the second argument to theadd method.

For example, to make button 4 be extra tall, the example has this code:

c.ipady = 40;

And before setting the constraints of the next component, the code resets the value of ipady to the default:

c.ipady = 0;

If a component's display area is larger than the component itself, then you can specify whereabouts in the display area the component will be displayed by using the GridBagConstraints.anchor constraint. The anchor constraint's values can be absolute (north, south, east, west, and so on), or orientation-relative (at start of page, at end of line, at the start of the first line, and so on), or relative to the component's baseline. For a full list of the possible values of the anchor constraint, including baseline-relative values,see the API documentation for [GridBagConstraints.anchor](https://docs.oracle.com/javase/8/docs/api/java/awt/GridBagConstraints.html" \l "anchor" \t "_blank). You can see in the code extract above that Button 5 specifies that it should be displayed at the end of the display area by setting an anchor atGridBagConstraints.PAGE\_END.

**Note:** The Tutorial's examples used to specify the constraints object a different way, which you might see in other programs as well. Rather than specifying the constraints with the add method, our examples used to invoke thesetConstraints method on the GridBagLayout object. For example:

GridBagLayout gridbag = new GridBagLayout();

pane.setLayout(gridbag);

...

gridbag.setConstraints(button, c);

pane.add(button);

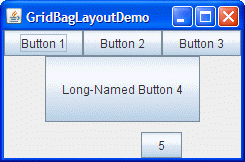
However, we recommend you use the Container.add method since it makes for cleaner code than if you were to use setConstraints.

Here is a table that shows all the constraints for each component in GridBagLayoutDemo's content pane. Values that are not the default are marked in **boldface**. Values that are different from those in the previous table entry are marked in *italics*.

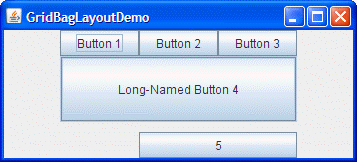
|  |  |
| --- | --- |
| **Component** | **Constraints** |
| All components | ipadx = 0  **fill = GridBagConstraints.HORIZONTAL** |
| Button 1 | ipady = 0 **weightx = 0.5** weighty = 0.0 gridwidth = 1 anchor = GridBagConstraints.CENTER insets = new Insets(0,0,0,0) **gridx = 0gridy = 0** |
| Button 2 | **weightx = 0.5**  ***gridx = 1***  **gridy = 0** |
| Button 3 | **weightx = 0.5**  ***gridx = 2***  **gridy = 0** |
| Button 4 | ***ipady = 40***  *weightx = 0.0*  ***gridwidth = 3***  ***gridx = 0***  ***gridy = 1*** |
| Button 5 | *ipady = 0*  weightx = 0.0  ***weighty = 1.0***  ***anchor = GridBagConstraints.PAGE\_END***  ***insets = new Insets(10,0,0,0)***  ***gridwidth = 2***  ***gridx = 1***  ***gridy = 2*** |

GridBagLayoutDemo has two components that span multiple columns (buttons 4 and 5). To make button 4 tall, we added internal padding (ipady) to it. To put space between buttons 4 and 5, we used insets to add a minimum of 10 pixels above button 5, and we made button 5 hug the bottom edge of its cell.

All the components in the pane container are as wide as possible, given the cells that they occupy. The program accomplishes this by setting the GridBagConstraints fill instance variable to GridBagConstraints.HORIZONTAL, leaving it at that setting for all the components. If the program did not specify the fill, the buttons would be at their natural width, like this:



When you enlarge GridBagLayoutDemo's window, the columns grow proportionately. This is because each component in the first row, where each component is one column wide, has weightx = 0.5. The actual value of these components' weightxis unimportant. What matters is that all the components, and consequently, all the columns, have an equal weight that is greater than 0. If no component managed by the GridBagLayout had weightx set, then when the components' container was made wider, the components would stay clumped together in the center of the container, like this:



If the container is given a size that is smaller or bigger than the prefered size, then any space is distributed according to theGridBagContainer weights.

Note that if you enlarge the window, the last row is the only one that gets taller. This is because only button 5 has weightygreater than zero.

**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 Java 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. 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 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

pop-up 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 isdisabled.

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( )

CheckboxMenuItem(String itemName)

CheckboxMenuItem(String itemName, boolean on)

Here, *itemName* is the name shown in the menu. Checkable items operate as toggles.

Each time one is selected, its state changes. In the first two forms, the checkable entry is

unchecked. In the third form, if *on* is **true**, the checkable entry is initially checked.

Otherwise, it is cleared.

You can obtain the status of a checkable item by calling **getState( )**. You can set it to a

known state by using **setState( )**. These methods are shown here:

boolean getState( )

void setState(boolean checked)

If the item is checked, **getState( )** returns **true**. Otherwise, it returns **false**. To check an

item, pass **true** to **setState( )**. To clear an item, pass **false**.

Once you have created a menu item, you must add the item to a **Menu** object by using

**add( )**, which has the following general form:

MenuItem add(MenuItem item)

Here, *item* is the item being added. Items are added to a menu in the order in which the

calls to **add( )** take place. The *item* is returned.

Once you have added all items to a **Menu** object, you can add that object to the menu

bar by using this version of **add( )** defined by **MenuBar**:

Menu add(Menu menu)

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

Menus only generate events when an item of type **MenuItem** or **CheckboxMenuItem** is

selected. They do not generate events when a menu bar is accessed to display a dropdown

menu, for example. Each time a menu item is selected, an **ActionEvent** object is

generated. Each time a check box menu item is checked or unchecked, an **ItemEvent**

object is generated. Thus, you must implement the **ActionListener** and **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( ).

**Example:**

import java.awt.\*;

import java.awt.event.\*;

public class menudemo extends Frame

{

MenuBar mbr=new MenuBar();

Menu f,e,sa;

MenuItem n,o,sapdf,sawd,cu,pas;

CheckboxMenuItem m=new CheckboxMenuItem("Demo");

menudemo(String str)

{

super(str);

setMenuBar(mbr);

f=new Menu("File");

e=new Menu("Edit");

sa=new Menu("Save as");

n=new MenuItem("New");

o=new MenuItem("Open");

sapdf=new MenuItem("Save as pdf");

sawd=new MenuItem("Save as word");

cu=new MenuItem("Cut");

pas=new MenuItem("Paste");

mbr.add(f);

mbr.add(e);

f.add(n);

f.add(o);

f.addSeparator();

f.add(sa);

sa.add(sapdf);

sa.add(sawd);

e.add(cu);

e.add(pas);

e.add(m);

setSize(400,400);

setVisible(true);

addWindowListener(new WindowAdapter()

{

public void windowClosing(WindowEvent e)

{

System.exit(0);

}

});

}

public static void main(String ar[])

{

menudemo d=new menudemo("Menu program demo");

}

}

**Dialog Boxes**

Often, you will want to use a *dialog box* to hold a set of related controls. Dialog boxes are

primarily used to obtain user input. They are similar to frame windows, except that dialog

boxes are always child windows of a top-level window. Also, dialog boxes don't have

menu bars. In other respects, dialog boxes function like frame windows. (You can add

controls to them, for example, in the same way that you add controls to a frame window.)

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, other parts of your

program remain active and accessible. Dialog boxes are of type **Dialog**. The most

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.*

Generally, you will subclass **Dialog**, adding the functionality required by your application.

**Example:**

import java.awt.\*;

import java.awt.event.\*;

class dialog extends Dialog implements ActionListener

{

Button b;

dialog(Frame parent,String title,boolean mode)

{

super(parent,title,mode);

setLayout(new FlowLayout());

setSize(200,200);

setVisible(true);

b=new Button("Ok");

add(b);

b.addActionListener(this);

}

public void actionPerformed(ActionEvent e)

{

dispose();

}

}

public class menudemo extends Frame

{

MenuBar mbr=new MenuBar();

Menu f,e,sa;

MenuItem n,o,sapdf,sawd,cu,pas;

CheckboxMenuItem m=new CheckboxMenuItem("Demo");

menudemo(String str)

{

super(str);

setMenuBar(mbr);

f=new Menu("File");

e=new Menu("Edit");

sa=new Menu("Save as");

n=new MenuItem("New");

o=new MenuItem("Open");

sapdf=new MenuItem("Save as pdf");

sawd=new MenuItem("Save as word");

cu=new MenuItem("Cut");

pas=new MenuItem("Paste");

mbr.add(f);

mbr.add(e);

f.add(n);

f.add(o);

f.addSeparator();

f.add(sa);

sa.add(sapdf);

sa.add(sawd);

e.add(cu);

e.add(pas);

e.add(m);

setSize(400,400);

setVisible(true);

addWindowListener(new WindowAdapter()

{

public void windowClosing(WindowEvent e)

{

System.exit(0);

}

});

}

public static void main(String ar[])

{

menudemo d=new menudemo("Menu program demo");

dialog g=new dialog(d,"Dialog window",false);

}

}

**FileDialog**

Java provides a built-in dialog box that lets the user specify a file. 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.

**FileDialog** provides these constructors:

FileDialog(Frame parent, String boxName)

FileDialog(Frame parent, String boxName, int how)

FileDialog(Frame parent)

Here, *parent* is the owner of the dialog box, and *boxName* is 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. The third constructor creates a dialog box for

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.

**Example:**

import java.awt.\*;

import java.awt.event.\*;

public class filediademo extends Frame

{

String str=” “;

public static void main(String ar[])

{

filediademo d=new filediademo();

d.setTitle("File dialog opening frame");

d.setSize(400,400);

d.setVisible(true);

d.addWindowListener(new WindowAdapter()

{

public void windowClosing(WindowEvent e)

{

System.exit(0);

}

});

FileDialog g=new FileDialog(d,"Title",FileDialog.LOAD);

g.setVisible(true);

}

}