LAYOUT MANAGERS

The Layout Managers are used to arrange components of frame in a particular manner. A layout manager arranges the child components of a container. It positions and sets the size of components within the container's display area according to a particular layout scheme. The layout manager's job is to fit the components into the available area, while maintaining the proper spatial relationships between the components. AWT comes with a few standard layout managers that will collectively handle most situations.

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. Java has several predefined **LayoutManager** classes. You can use the layout manager that best fits your application.

FlowLayout

FlowLayout implements a simple layout style, which is similar to how words flow in a text editor. The direction of the layout is governed by the container's component orientation property, which, by default, is left to right, top to bottom. Therefore, by default, components are laid out line-by-line beginning at the center . 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

These values specify left, center, right, leading edge, and trailing edge alignment, respectively.

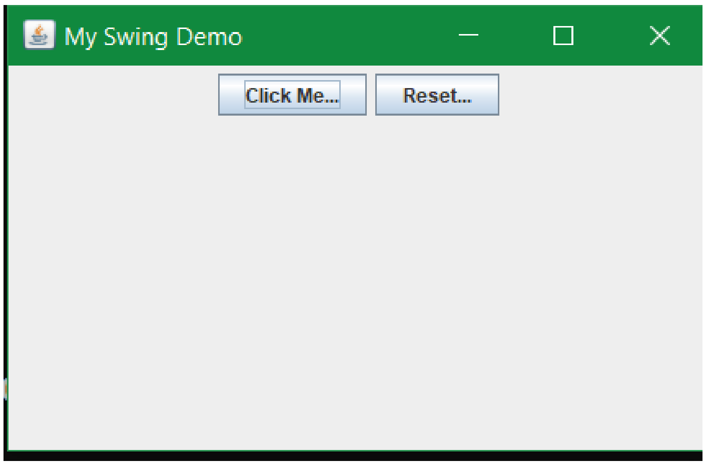
The third constructor allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively**.**

Example:FlowLayout

```
import javax.swing.*;
import java.awt.*;
class SwingDemo extends JFrame
{
        JButton b1,b2;
        SwingDemo()
        {
                super("My Swing Demo");
                setLayout(new FlowLayout());
                b1=new JButton("Click Me...");
                b2=new JButton("Reset...");
                add(b1);
                add(b2);
                setSize(500,600);
                setVisible(true);
        }
        public static void main(String args[])
        {
```

```
                    new SwingDemo();
          }
}
```

Output



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**:

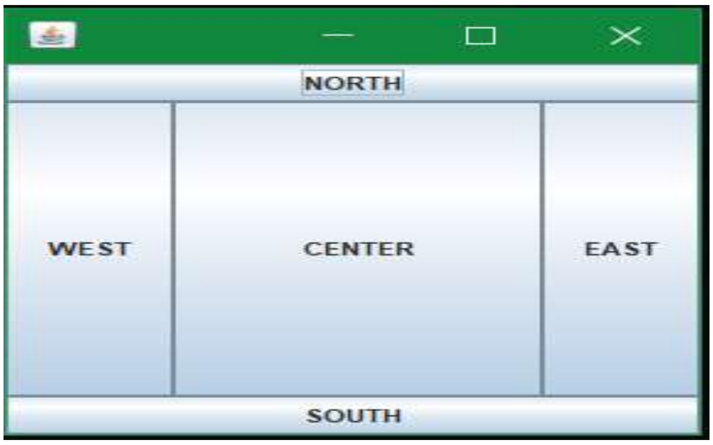
void add(Component *compObj*, Object *region*)

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

Example:BorderLayout

```
import java.awt.*;
import javax.swing.*;
public class Border1
{
	JFrame f;
	Border1()
	{
		f=new JFrame();
		JButton b1=new JButton("NORTH");;
		JButton b2=new JButton("SOUTH");;
		JButton b3=new JButton("EAST");;
		JButton b4=new JButton("WEST");;
		JButton b5=new JButton("CENTER");;
		f.add(b1,BorderLayout.NORTH);
		f.add(b2,BorderLayout.SOUTH);
		f.add(b3,BorderLayout.EAST);
		f.add(b4,BorderLayout.WEST);
		f.add(b5,BorderLayout.CENTER);
		f.setSize(300,300);
		f.setVisible(true);
	}
	public static void main(String[] args)
	{
		new Border1();
	}
}
```

<u>Output</u>



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*)

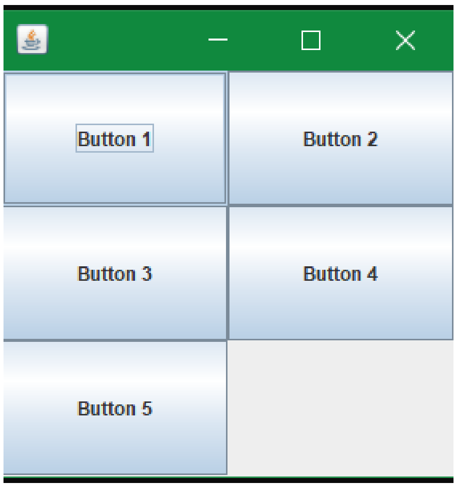
The first form creates a single-column grid layout. The second form creates a grid layout with the specified number of rows and columns. The third form allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively. Either *numRows* or *numColumns*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

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

Example:GridLayout

```
import java.awt.*;
import javax.swing.*;
public class Grid
{
        JFrame f;
        JButton b1,b2,b3,b4,b5;
        Grid()
        {
                f=new JFrame();
                f.setLayout(new GridLayout(3,2));
                b1=new JButton("Button 1");
                b2=new JButton("Button 2");
                b3=new JButton("Button 3");
                b4=new JButton("Button 4");
                b5=new JButton("Button 5");
                f.add(b1);
                f.add(b2);
                f.add(b3);
                f.add(b4);
                f.add(b5);
                f.setSize(300,300);
                f.setVisible(true);
        }
        public static void main(String[] args)
        {
                new Grid();
        }
}
```

Output



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 window. Once these steps are complete, you must provide some way for the user to select between cards. One common approach is to include one push button for each card in the deck.

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

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

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

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*)

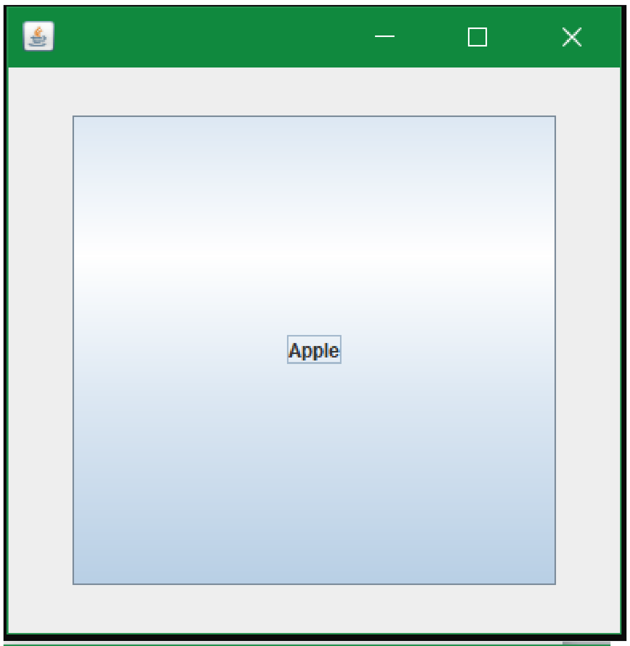
Here, *deck* is a reference to the container (usually a panel) that holds the cards, and *cardName* is the name of a card. Calling **first( )** causes the first card in the deck to be shown. To show the last card, call **last( )**. To show the next card, call **next( )**. To show the previous card, call **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*.

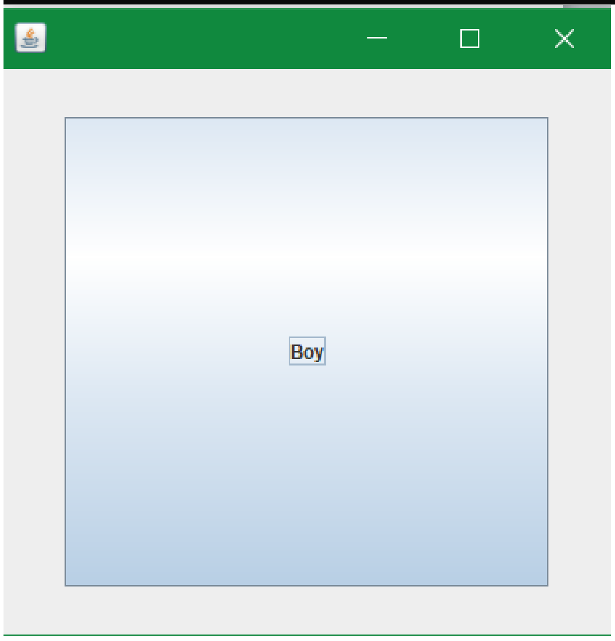
Example: CardLayout

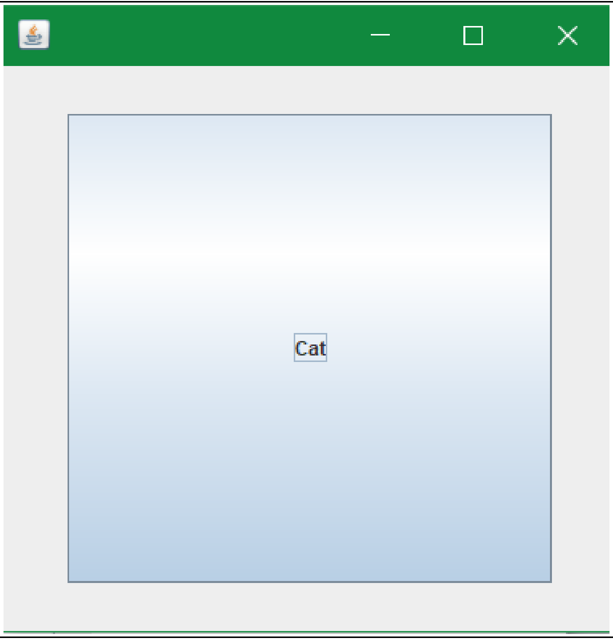
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

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

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

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

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

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

Output







Box Layout

The **BoxLayout** class is used to arrange the components either vertically (along Y-axis) or horizontally (along X-axis). In BoxLayout class, the components are put either in a single row or a single column. The components will not wrap so, for example, a horizontal arrangement of components will stay horizontally arranged when the frame is resized. For this purpose, BoxLayout provides four constants. They are as follows:

public static final int X_AXIS

public static final int Y_AXIS

public static final int LINE_AXIS

public static final int PAGE_AXIS

Constructor of the **BoxLayout** class:

BoxLayout(Container c, int axis) - This creates a BoxLayout class that arranges the components with the X-axis or Y-axis.

Commonly Used Methods:

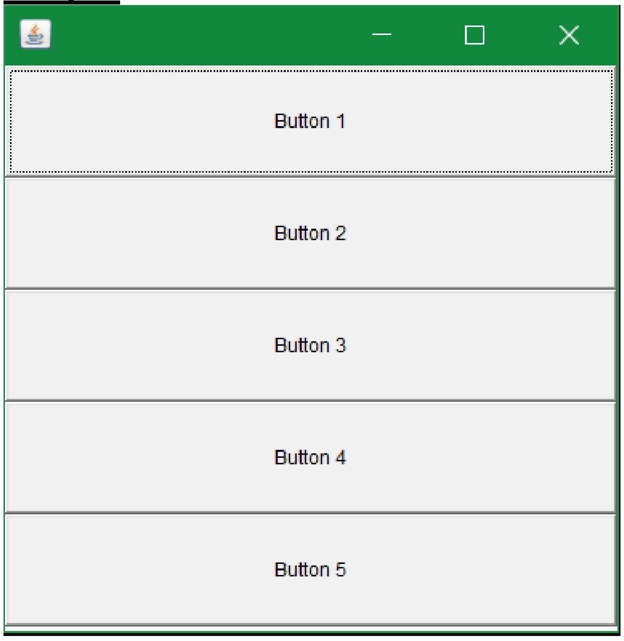
Method	**Description**
getLayoutAlignment	Returns the alignment along the X axis for the

X (Container con)	container.
getLayoutAlignment Y (Container con)	Returns the alignment along the Y axis for the container
maximumLayoutSize (Container con)	Returns the maximum dimensions the target container can use to lay out the components it contains.
minimumLayoutSize (Container con)	Returns the minimum dimensions needed to lay out the components contained in the specified target container.
layoutContainer (Container tar)	Called by the AWT when the specified container needs to be laid out.

Example: BoxLayout – Y AXIS

```
import java.awt.*;
import javax.swing.*;

public class BoxLayoutExample1 extends Frame
{
        Button buttons[];
        public BoxLayoutExample1 ()
        {
                buttons = new Button [5];
                for (int i = 0;i<5;i++)
                {
                        buttons[i] = new Button ("Button " + (i + 1));
                        add (buttons[i]);
                }
                setLayout (new BoxLayout (this, BoxLayout.Y_AXIS));
                setSize(400,400);
                setVisible(true);
        }
        public static void main(String args[])
        {
                BoxLayoutExample1 b=new BoxLayoutExample1();
        }
}
```

Output



Null Layout

The layout managers are used to automatically decide the position and size of the added components. In the absence of a layout manager, the position and size of the components have to be set manually. The **setBounds()** method is used in such a situation to set the position and size. To specify the position and size of the components manually, the layout manager of the frame can be null. Null layout is not a real layout manager. It means that no layout manager is assigned and the components can be put at specific x,y coordinates.

setBounds()

The setBounds() method needs four arguments. The first two arguments are x and y coordinates of the top-left corner of the component, the third argument is the width of the component and the fourth argument is the height of the component.

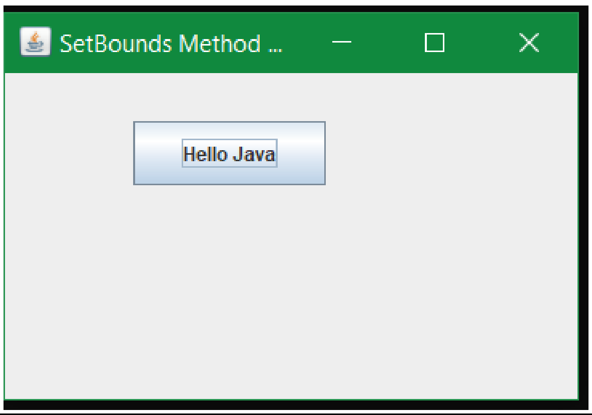
Syntax

```
setBounds(int x-coordinate, int y-coordinate, int width, int height)
```

Example : Null Layout

```
import javax.swing.*;
import java.awt.*;
public class SetBoundsTest
{
        public static void main(String arg[])
        {
                JFrame frame = new JFrame("SetBounds Method Test");
                frame.setSize(375, 250);
                frame.setLayout(null);    // Setting layout as null
                JButton button = new JButton("Hello Java");

                button.setBounds(80,30,120,40);       // Setting position and size of
a button
                frame.add(button);
                frame.setVisible(true);
  }
}
```

<u>Output</u>


SetBounds Method ...
Hello Java