Layout Managers

A layout manager automatically arranges the controls within a window by using some type of algorithm. In GUI environments, such as Windows, you can layout your controls by hand. It is possible to lay out Java controls by hand, too, but we do not do so 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 have not been realized. This is a chicken-and-egg situation; it is confusing to figure out when it is okay to use the size of a given component to position it relative to another.

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.

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. Normally, you will want to use a layout manager.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.. You may override these methods for controls that you subclass. Default values are provided otherwise.

Java has several predefined LayoutManager classes.

- FlowLayout
- BorderLayout
- Insets

- GridLayout
- CardLayout

FlowLayout

FlowLayout is the default layout manager. 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.
The constructors for FlowLayout are shown below:

1. FlowLayout( )
2. FlowLayout(int how)
3. 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.

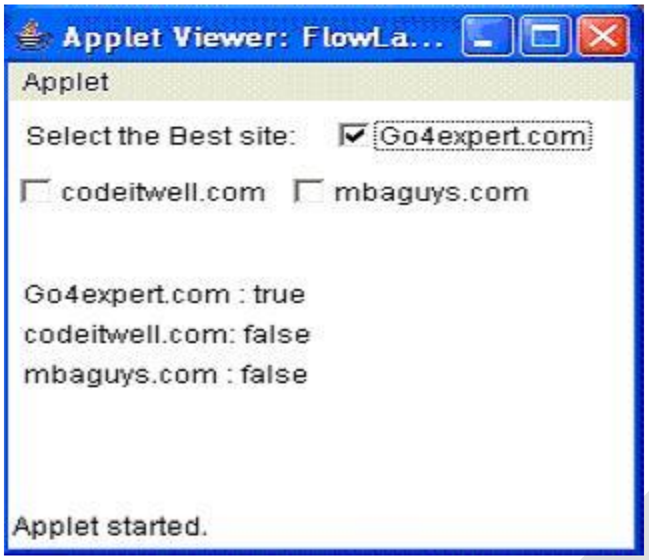
The second form lets you specify how each line is aligned. Valid values for how are as follows:

- FlowLayout.LEFT
- FlowLayout.CENTER
- FlowLayout.RIGHT

These values specify left, center, and right alignment, respectively. The third form allows specifying the horizontal and vertical space left between components in horz and vert, respectively.

Here is another type of the CheckboxDemo applet shown in the previous articles, such that it uses left-aligned flow layout.
Code:
```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="FlowLayoutTest" width=250 height=200>
</applet>
*/
public class FlowLayoutTest extends Applet implements ItemListener
{
    String str = "";
    Checkbox Go4expert, codeitwell,mbaguys;
    Label l1;
    public void init()
    {
      // set left-aligned flow layout
      setLayout(new FlowLayout(FlowLayout.LEFT));
      l1=new Label("Select the Best site:");
      Go4expert = new Checkbox("Go4expert.com", null, true);
      codeitwell = new Checkbox("codeitwell.com ");
      mbaguys = new Checkbox("mbaguys.net");
      add(l1);
      add(Go4expert);
      add(codeitwell);
      add(mbaguys);
      // register to receive item events

      Go4expert.addItemListener(this);
      codeitwell.addItemListener(this);
      mbaguys.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)
    {
      str = "Go4expert.com : " + Go4expert.getState();
      g.drawString(str, 6, 100);
      str = "codeitwell.com: " + codeitwell.getState();
      g.drawString(str, 6, 120);
      str = "mbaguys.net : " + mbaguys.getState();
      g.drawString(str, 6, 140);
   }
}
```

Output would be as shown below:-



Select the Best site: ☑ Go4expert.com
☐ codeitwell.com ☐ mbaguys.com

Go4expert.com : true
codeitwell.com: false
mbaguys.com : 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.

The constructors defined by BorderLayout are shown below:

1. BorderLayout( )
2. 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.

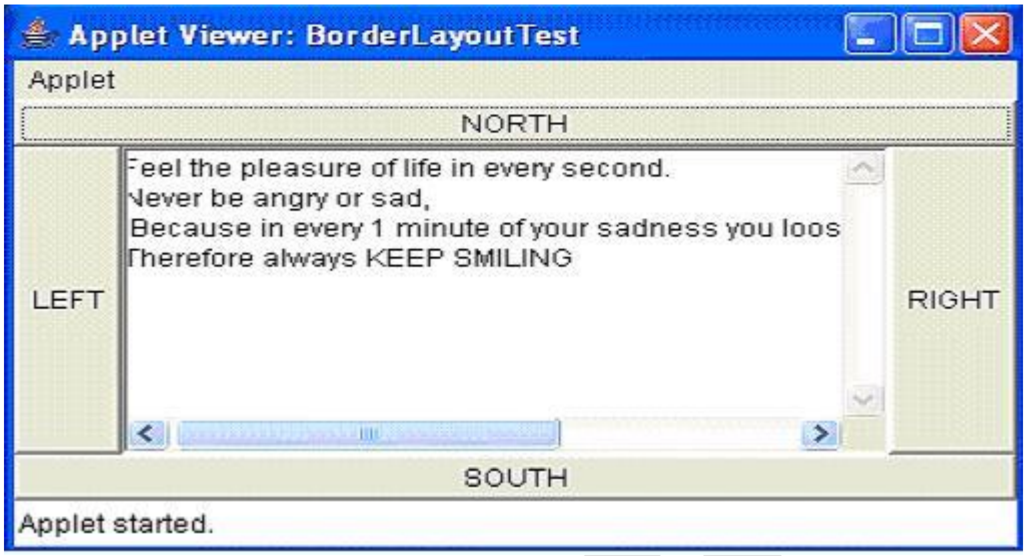
Example:

Here is an example of a BorderLayout with a component in each layout area:

Code:

```
import java.awt.*;
import java.applet.*;
import java.util.*;
/*
<applet code="BorderLayoutTest" width=400 height=200>
</applet>
*/
public class BorderLayoutTest extends Applet
{
    public void init()
    {
        setLayout(new BorderLayout());
        add(new Button("NORTH"), BorderLayout.NORTH);
        add(new Button("SOUTH"), BorderLayout.SOUTH);
        add(new Button("RIGHT"), BorderLayout.EAST);
        add(new Button("LEFT"), BorderLayout.WEST);
        String str = "Feel the pleasure of life in every second.\n" +
        "Never be angry or sad, \n " +
        "Because in every 1 minute of your sadness " +
        "you loose 60 seconds of your hapiness.\n" +
        "Therefore always KEEP SMILING \n\n";
        add(new TextArea(str), BorderLayout.CENTER);
    }
}
```

Output would be as shown below:



```
Feel the pleasure of life in every second.
Never be angry or sad,
Because in every 1 minute of your sadness you loos
Therefore always KEEP SMILING
```

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 below:

1. GridLayout( )
2. GridLayout(int numRows, int numColumns)
3. 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:

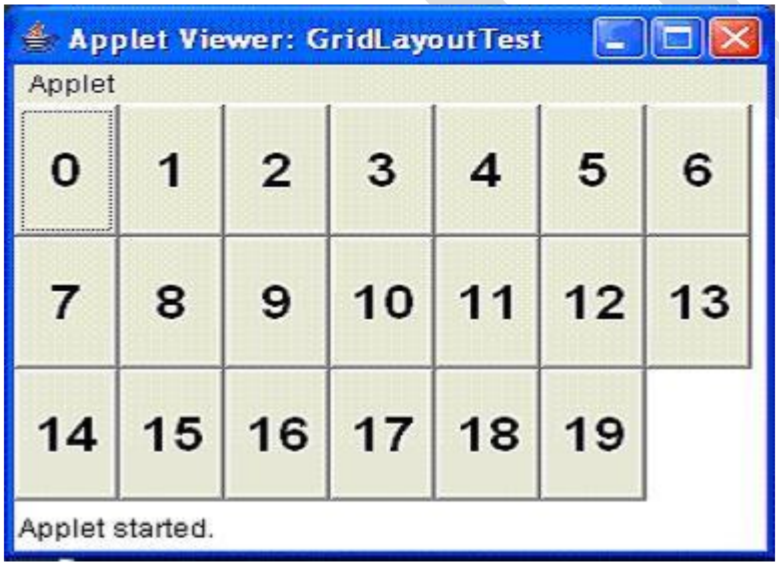
Here is a sample program that creates a 7×3 grid and fills it in with 20 buttons, each labeled with its index:

Code:

```
// Demonstrate GridLayout
```

```
import java.awt.*;
import java.applet.*;
/*
<applet code="GridLayoutTest" width=300 height=200>
</applet>
*/
public class GridLayoutTest extends Applet
{
  static final int row =3;
  static final int col =7;
  public void init()
  {
    setLayout(new GridLayout(row,col));
    setFont(new Font("SansSerif", Font.BOLD, 24));
    for(int i = 0; i < 20; i++)
    {
    add(new Button("" + i));
    }
  }
}
```

Output would be as shown below:



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. We can prepare the other layouts and have them hidden, ready to be activated when needed.

CardLayout provides the following two constructors:

1. CardLayout( )
2. 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. 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:

1. void first(Container deck)
2. void last(Container deck)
3. void next(Container deck)
4. void previous(Container deck)
5. 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:

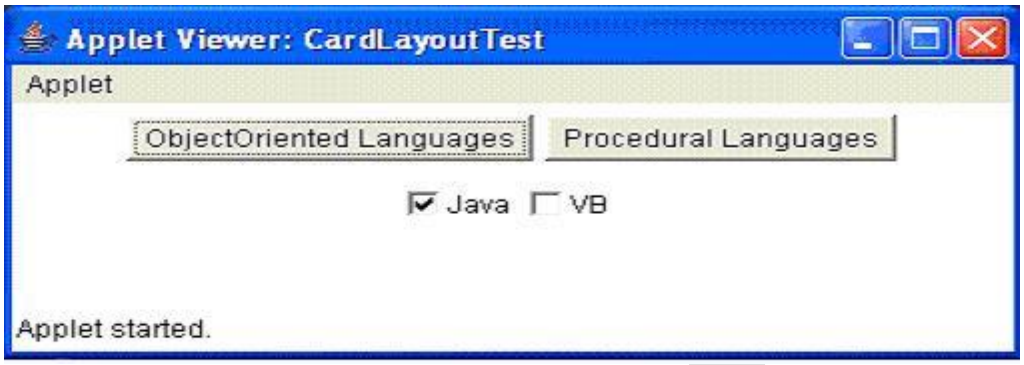
The following example creates a two-level card deck that allows the user to select a language. Procedural languages are displayed in one card. Object Oriented languages are displayed in the other card.

Code:

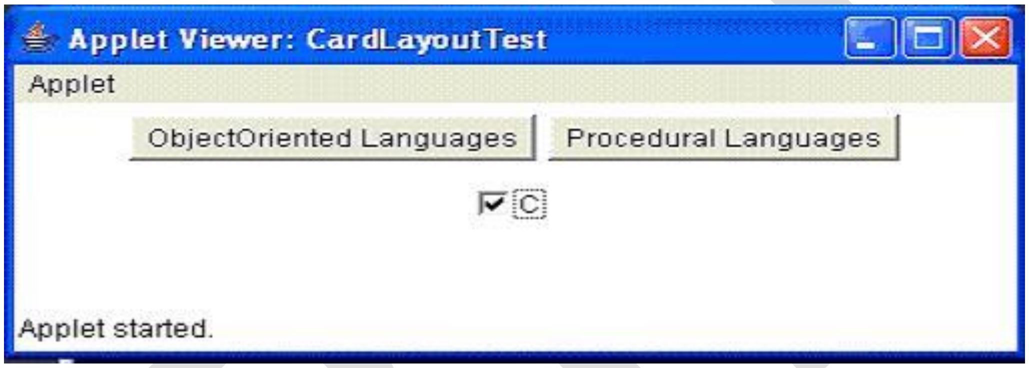
```
// Demonstrate CardLayout.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="CardLayoutTest" width=400 height=100>
</applet>
*/
public class CardLayoutTest extends Applet implements ActionListener,
MouseListener
{
  Checkbox Java, C, VB ;
  Panel langCards;
  CardLayout cardLO;
  Button OO, Other;
  public void init()
  {
    OO = new Button("ObjectOriented Languages");
    Other = new Button("Procedural Languages");
    add(OO);
    add(Other);
    cardLO = new CardLayout();
    langCards = new Panel();
    langCards.setLayout(cardLO); // set panel layout to card layout
    Java = new Checkbox("Java", null, true);
    C = new Checkbox("C");
    VB = new Checkbox("VB");

    // add OO languages check boxes to a panel
    Panel OOPan = new Panel();
```

```
      OOPan.add(Java);
      OOPan.add(VB);
      // Add other languages check boxes to a panel
      Panel otherPan = new Panel();
      otherPan.add(C);
      // add panels to card deck panel
      langCards.add(OOPan, "Object Oriented Languages");
      langCards.add(otherPan, "Procedural Languages");
      // add cards to main applet panel
      add(langCards);
      // register to receive action events
      OO.addActionListener(this);
      Other.addActionListener(this);
      // register mouse events
      addMouseListener(this);
    }
    // Cycle through panels.
    public void mousePressed(MouseEvent me)
    {
      cardLO.next(langCards);
    }
    // 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() == OO)
       {
           cardLO.show(langCards, "Object Oriented Languages");
       }
       else
       {
           cardLO.show(langCards, "Procedural Languages");
       }
    }
}
```

Output would be as shown below:



On clicking the Procedural languages Button the following output would be displayed:

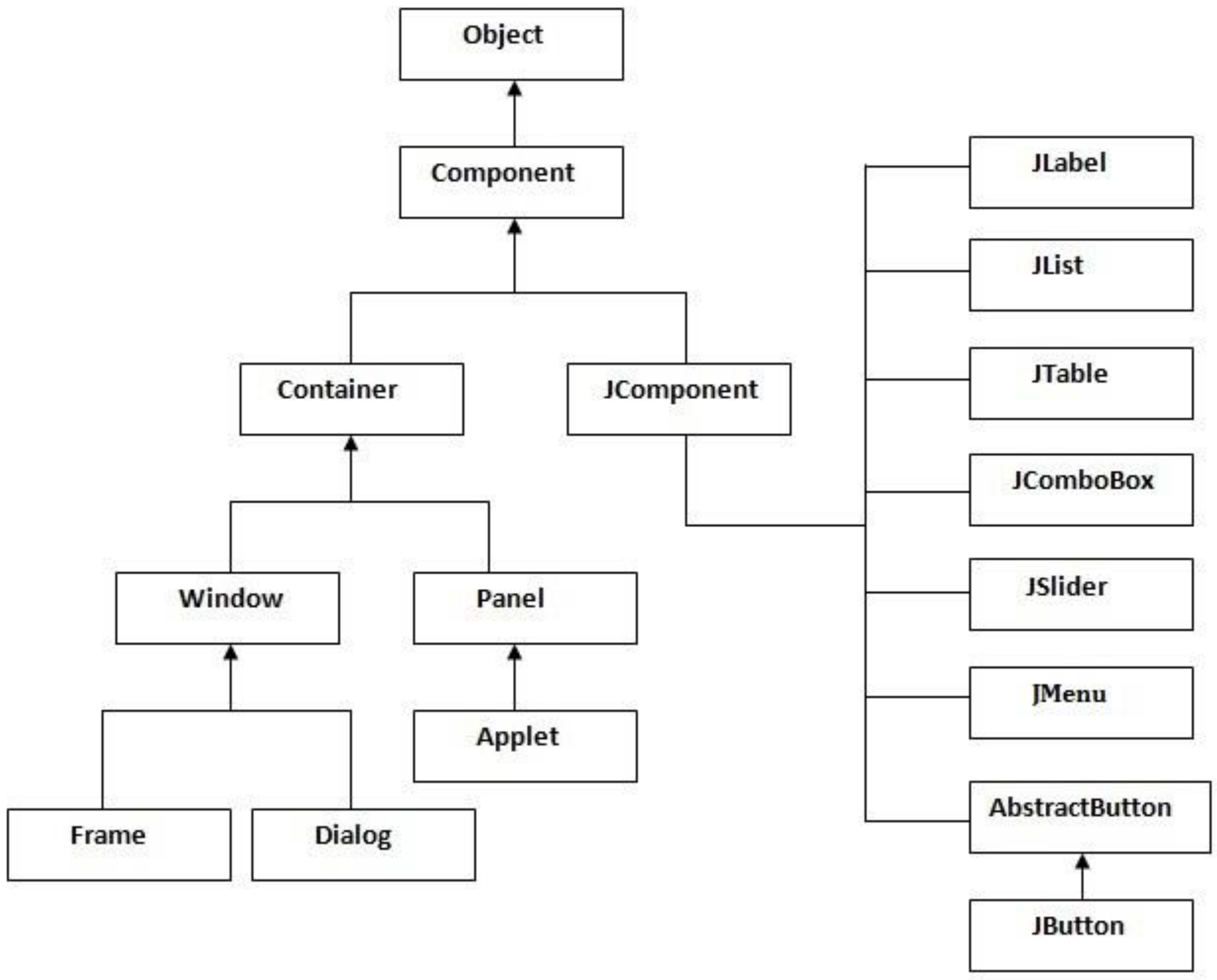


Difference between AWT and Swing

There are many differences between java awt and swing that are given below.

No.	Java AWT	Java Swing
1)	AWT components are **platform-dependent**.	Java swing components are **platform-independent**.
2)	AWT components are **heavyweight**.	Swing components are **lightweight**.
3)	AWT **doesn't support pluggable look and feel**.	Swing **supports pluggable look and feel**.
4)	AWT provides **less components** than Swing.	Swing provides **more powerful componentssuch** as tables, lists, scrollpanes, colorchooser, tabbedpane etc.
5)	AWT **doesn't follows MVC**(Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view.	Swing **follows MVC**.

he hierarchy of java swing API is given below.


Object
Component
Container
JComponent
Window
Panel
Applet
Frame
Dialog
JLabel
JList
JTable
JComboBox
JSlider
JMenu
AbstractButton
JButton