# CS205 Object Oriented Programming in Java

## Module 5 - **Graphical User Interface and Database support of Java**

## (Part 4)

Prepared by

**Renetha J.B.**

AP

Dept.of CSE,

Lourdes Matha College of Science and Technology

# Topics

- ☑ **Swings**
  - ☑ Swing Layout Managers

# Swing Layout Managers

- A layout manager **automatically arranges our controls within a window** by using some type of algorithm.
- 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.
- The **setLayout( )** method has the following general form:

```
void setLayout(LayoutManager layoutObj)
```

# **Swing Layout Managers**(contd.)

- The *layout manager is notified* each time we add a component to a container.
- Each layout manager keeps track of a list of components that are stored by their names.
- 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.

- Java has several predefined **LayoutManager classes,**
  - **FlowLayout**
  - **BorderLayout**
  - **GridLayout**
  - **CardLayout**
  - **GridBagLayout**

# FlowLayout

- The direction of the layout is governed by the container's component orientation property, which, by default, is **left to right, top to bottom.**
- In low Layout
  - components are laid out **line-by-line** **beginning at the upper-left corner.**
  - 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.
- The constructors for **FlowLayout:**

FlowLayout( )

FlowLayout(int *how)*

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

# **FlowLayout**(contd.)

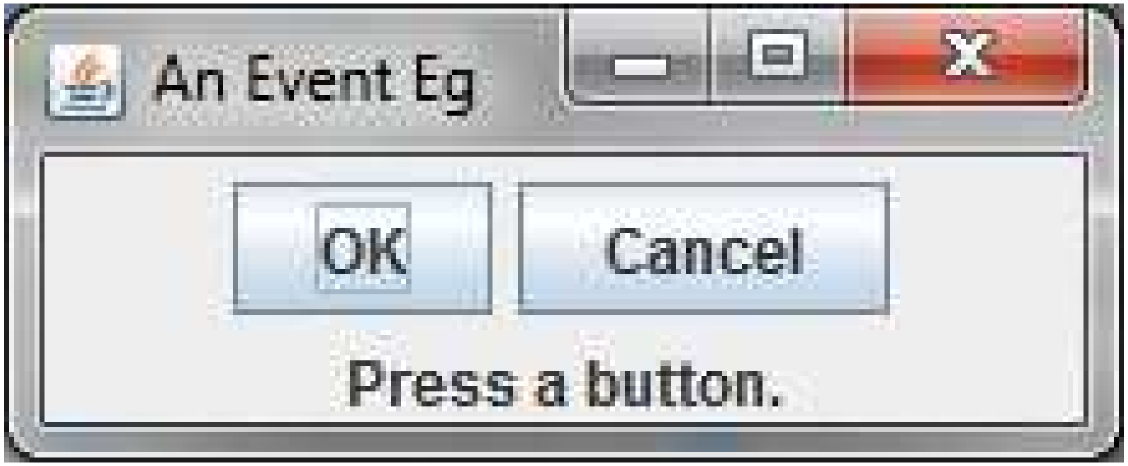
- FlowLayout( ) creates the default layout, which **centers components** and leaves five pixels of space between each component.
- FlowLayout(int *how)* lets us specify **how each line is aligned**.
  - Valid values for *how are as follows:*
    - FlowLayout.LEFT
    - FlowLayout.CENTER
    - FlowLayout.RIGHT
    - FlowLayout.LEADING
    - FlowLayout.TRAILING
      - These values specify left, center, right, leading edge, and trailing edge alignment, respectively.
- FlowLayout(int *how, int horz, int vert)* allows us to specify the **horizontal and vertical space left between components** in *horz and vert,* respectively.

```
import java.awt.*;      import java.awt.event.*;
import javax.swing.*;
class EventDemoSwing  extends JFrame implements
      ActionListener{
      JLabel jlab;
      JFrame jfrm;
      JButton jbtnOk;
      JButton jbtnCancel;
      EventDemoSwing()
      {
jfrm = new JFrame("An Event Eg");
jfrm.setLayout(new FlowLayout());
jfrm.setSize(220, 90);
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
jbtnOk = new JButton("OK");
JButton jbtnCancel = new JButton("Cancel");
jbtnOk.setToolTipText("click");
jbtnOk.addActionListener(this);
jbtnCancel.addActionListener(this);
jfrm.add(jbtnOk);
jfrm.add(jbtnCancel);
jlab = new JLabel("Press a button.");
jfrm.add(jlab);
jfrm.setVisible(true);          }
```

```
public void actionPerformed(ActionEvent ae)
   {
   String s = ae.getActionCommand();
   if(s.equalsIgnoreCase("ok"))
         jlab.setText("OK  pressed.");
   else if(s.equalsIgnoreCase("cancel"))
         jlab.setText("Cancel  pressed.");
   }
```

```
      public static void main(String args[])
 {SwingUtilities.invokeLater(new Runnable()
       {
      public void run()
       { new EventDemoSwing();
      }
      } );
  }   }
```

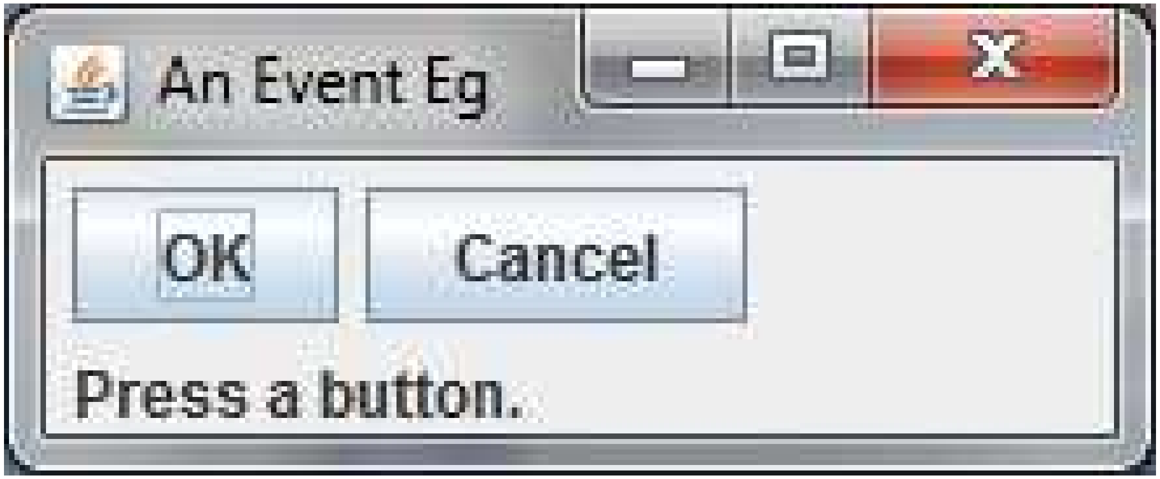
# **FlowLayout**(contd.)



Press a button.

If in code we modify the layout as

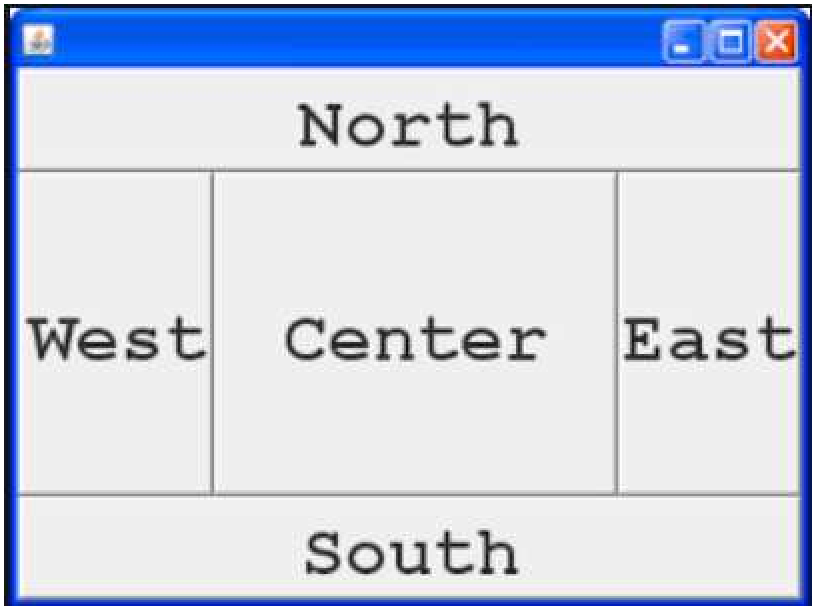
```
jfrm.setLayout(new FlowLayout(FlowLayout.LEFT));
```



Press a button.

# BorderLayout

- By default, the content pane associated with a **JFrame uses border layout.**
- 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
  - West.
- The middle area is called
  - Center



# **BorderLayout**(contd.)

- The constructors defined by **BorderLayout:**

BorderLayout( )

- BorderLayout( ) creates a default border layout.
- BorderLayout(int *horz, int vert)* specify the horizontal and vertical space left between components in *horz and vert, respectively*

# **BorderLayout**(contd.)

- BorderLayout defines the following constants
  - 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**:**

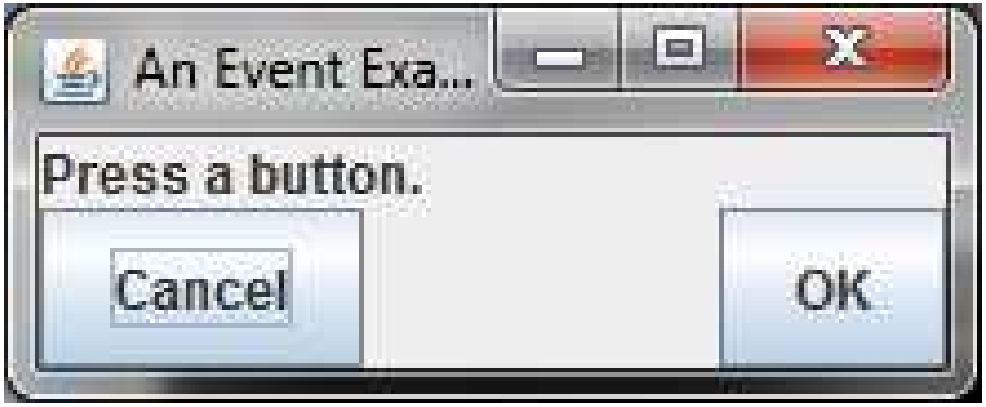
```
void add(Component compObj, Object region)
```

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

```
import java.awt.*;      import java.awt.event.*;
import javax.swing.*;
class EventDemoSwing  extends JFrame implements
      ActionListener{
      JLabel jlab;
      JFrame jfrm;
      JButton jbtnOk;
      JButton jbtnCancel;
      EventDemoSwing()
      {
jfrm = new JFrame("An Event Eg");
jfrm.setLayout(new BorderLayout ());
jfrm.setSize(220, 90);
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
jbtnOk = new JButton("OK");
JButton jbtnCancel = new JButton("Cancel");
jbtnOk.setToolTipText("click");
jbtnOk.addActionListener(this);
jbtnCancel.addActionListener(this);
jfrm.add(jbtnOk,BorderLayout.EAST);
jfrm.add(jbtnCancel, BorderLayout.WEST);
jlab = new JLabel("Press a button.");
jfrm.add(jlab BorderLayout.NORTH);
jfrm.setVisible(true);          }
```

```
public void actionPerformed(ActionEvent ae)
    {
    String s = ae.getActionCommand();
    if(s.equalsIgnoreCase("ok"))
          jlab.setText("OK  pressed.");
    else if(s.equalsIgnoreCase("cancel"))
          jlab.setText("Cancel  pressed.");
    }
          public static void main(String args[])
   {SwingUtilities.invokeLater(new Runnable()
           {
          public void run()
           { new EventDemoSwing();
          }
          } );
       }   }
```

# **BorderLayout**(contd.)



Press a button.

# Using Insets

- Sometimes we may want to **leave a small amount of space** between the container that holds the components and the window that contains it.
  - To do this, override the **getInsets( ) method** that is defined by **Container.**
  - **getInsets( ) method** 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*

# **Using Insets**(contd.)

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

  Insets **getInsets**( )

  - When overriding this method, we must return a new **Insets object** that contains the inset spacing you desire.

# GridLayout

- **GridLayout** lays out components in a **two-dimensional grid**.
  - We can define the number of rows and columns.
- The constructors supported by GridLayout are

GridLayout( )

GridLayout(int *numRows, int numColumns)*

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

  - GridLayout( ) creates a single-column grid
  - GridLayout(int *numRows, int numColumns)* creates a grid layout with the specified number of rows and columns.
  - GridLayolayoutut(int *numRows, int numColumns, int horz, int vert)* allows us 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.

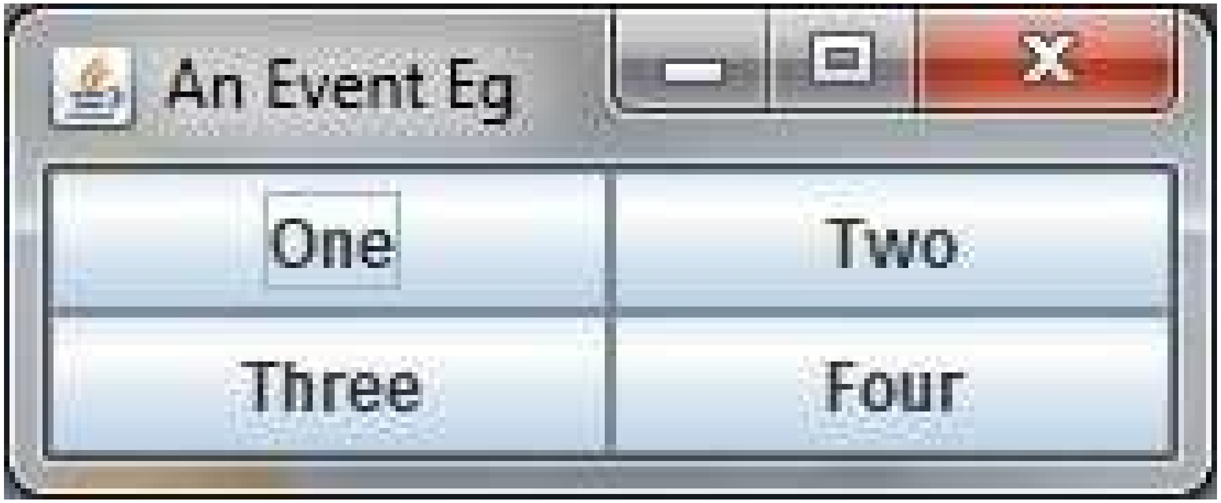
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class GridDemo  extends JFrame implements  ActionListener
{    JLabel jlab;
     JFrame jfrm;
     GridDemo()
     { jfrm = new JFrame("An Event Eg");
     jfrm.setLayout(new GridLayout (2,2));
     jfrm.setSize(220, 90);
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
     JButton jbtnOne = new JButton("One");
     JButton jbtnTwo= new JButton("Two");
     JButton jbtnThree = new JButton("Three");
     JButton jbtnFour = new JButton("Four");
     jbtnOne.addActionListener(this);
     jbtnTwo.addActionListener(this);
     jbtnThree.addActionListener(this);
     jbtnFour.addActionListener(this);
     jfrm.add(jbtnOne);
     jfrm.add(jbtnTwo);
     jfrm.add(jbtnThree);
     jfrm.add(jbtnFour);
     jfrm.setVisible(true);
     }
                                                    }
```

```
public void actionPerformed(ActionEvent ae)
   {       String s = ae.getActionCommand();
           if(s.equalsIgnoreCase("one"))
                       jlab.setText("One  pressed.");
           else if(s.equalsIgnoreCase("two"))
                       jlab.setText("Two pressed.");
           else if(s.equalsIgnoreCase("three"))
           jlab.setText("Three pressed.");
           else if(s.equalsIgnoreCase("four"))
           jlab.setText("Four pressed.");
   }
   public static void main(String args[])
  {SwingUtilities.invokeLater(new Runnable()
          { public void run()
          { new GridDemo(); }                 }
   );
   }
```

**jfrm.setLayout(new GridLayout (2,2));**

- This set 2 rows and 2 columns
- Components are filled in order from first row first column

| | |
|---|---|
| (0,0) | (0,1) |
| (1,0) | (1,1) |



# 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.
- **CardLayout** provides these two constructors:

CardLayout( )

CardLayout(int *horz, int vert)*

- – CardLayout( ) creates a default card layout.
- – CardLayout(int *horz, int vert)* allows to specify the horizontal and vertical space left between components in *horz and vert, respectively.*

# **CardLayout**(contd.)

- 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**

# CardLayout(contd.)

- We must **create**
  - **a panel** that contains the deck and
  - a **panel** for each card in the deck.
- Next, we **add components that form each card** to the appropriate panel.
- We then **add these panels to the panel** for which CardLayout is the layout manager.
- Finally, we **add this panel to the window.**

- Once these steps are complete, we 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.

# **CardLayout**(contd.)

- Use **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 we have created a deck, our 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)*

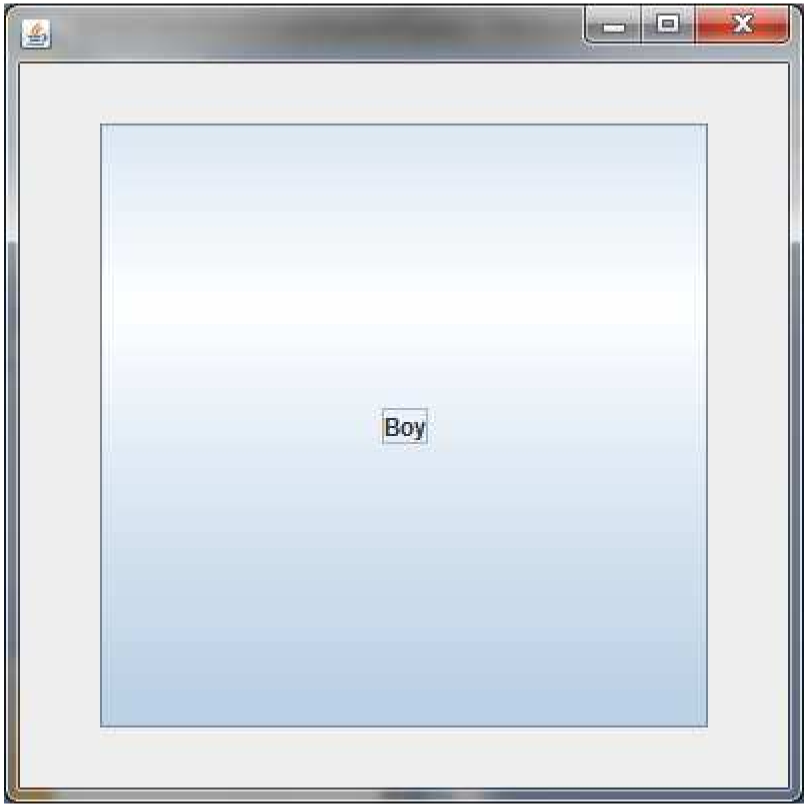
  void show(Container *deck, String cardName)*

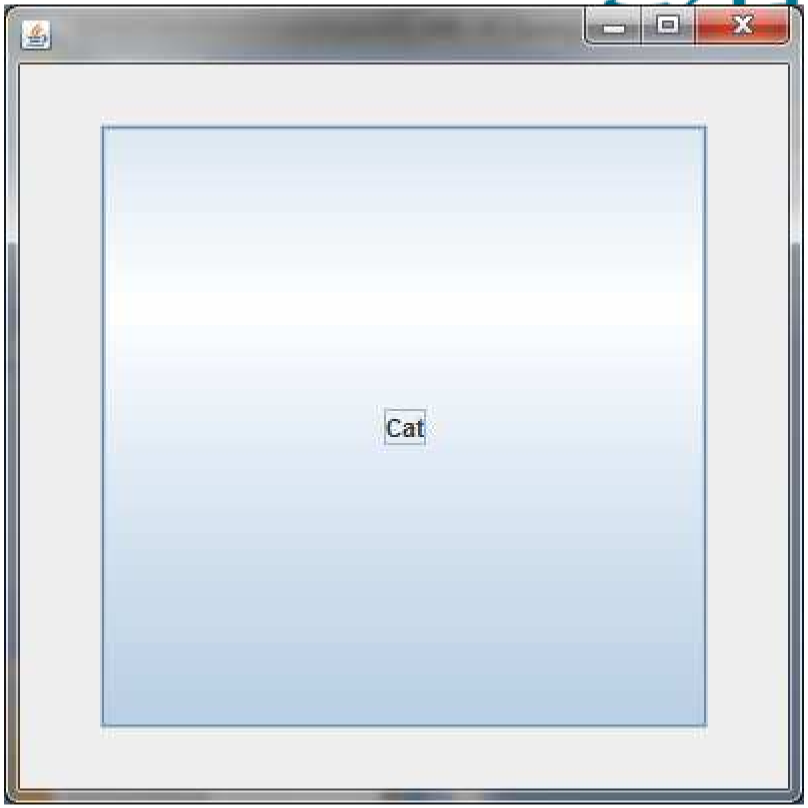
  - Here, *deck* is a reference to the container (usually a panel) that holds the cards, and *cardName* is the name of a card.
  - Calling first( ) causes the first card in the deck to be shown. To show the last card, call last( ). To show the next card, call next( ). To show the previous card, call previous(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

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class CardLayoutDemo extends JFrame implements
     ActionListener
{
CardLayout card;
JButton b1,b2,b3;
Container c;
   CardLayoutDemo()
     {
     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(b1);c.add(b2);c.add(b3);

   }
```

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

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





- When we click the button it changes to next button like cards

# GridBagLayout

- We can specify the **relative placement of components by specifying their positions** within cells inside a grid using **GridBagLayout**.
- The key to the grid bag is that each component can be a **different size**, and each **row** in the grid can have a **different number of columns**.
  - This is why the layout is called a *grid bag*.
- It's a **collection of small grids joined together**.
- The location and size of each component in a grid bag are determined by a set of **constraints** linked to it.
- The **constraints** are contained in an object of type **GridBagConstraints.**
  - **Constraints** include the height and width of a cell, and the placement of a component, its alignment, and its anchor point within the cell.

# **GridBagLayout**(contd.)

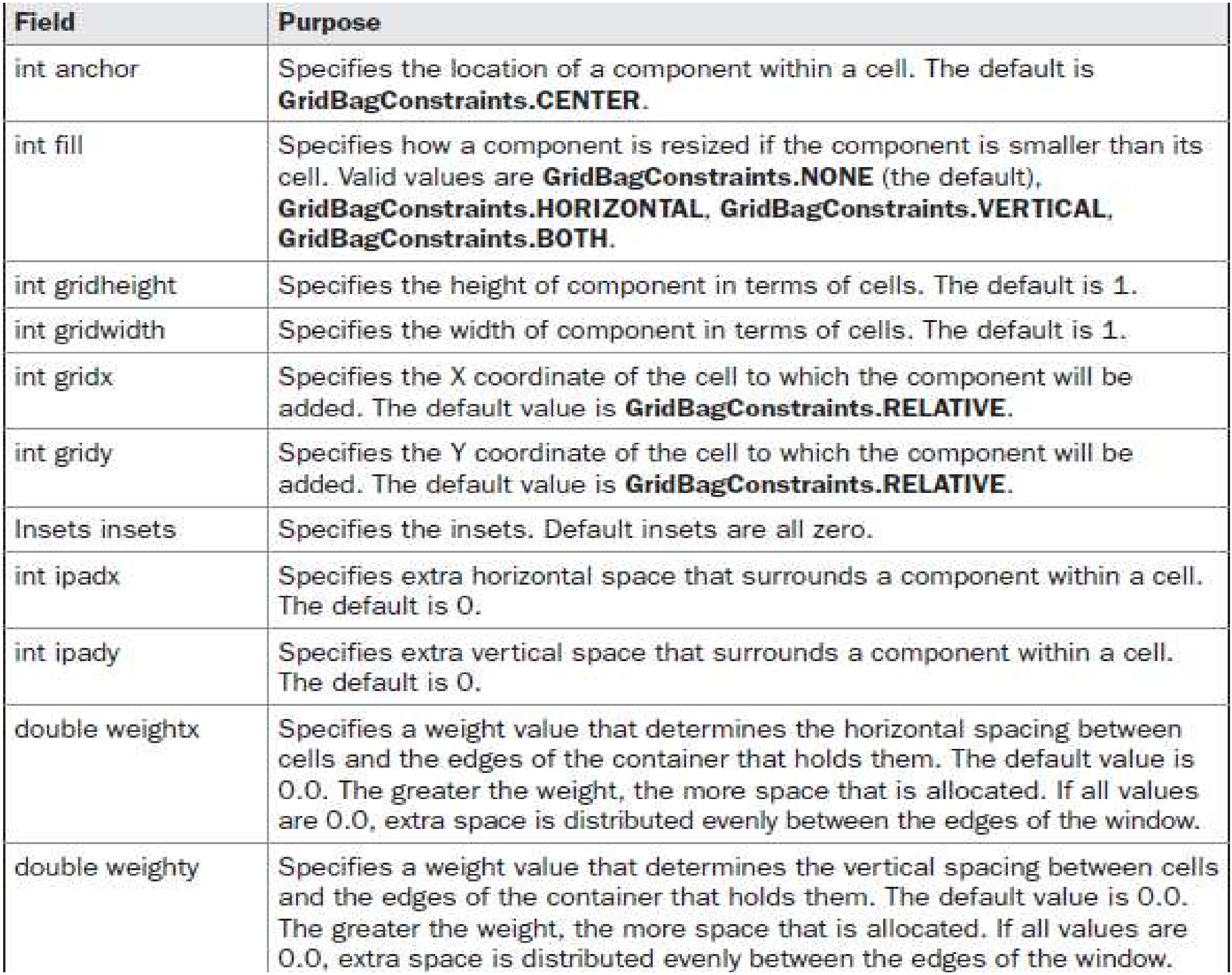
- The general procedure for using a grid bag is to
  - first create a new **GridBagLayout object** and to make it the current layout manager.
  - Then, **set the constraints** that apply to each component that will be added to the grid bag.
  - Finally, add the components to the layout manager.
- **GridBagLayout**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

  GridBagLayout( )
- GridBagLayout defines several methods, of which many are protected and not for general use.
- One method is setConstraints( )

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

# GridBagLayout(contd.)

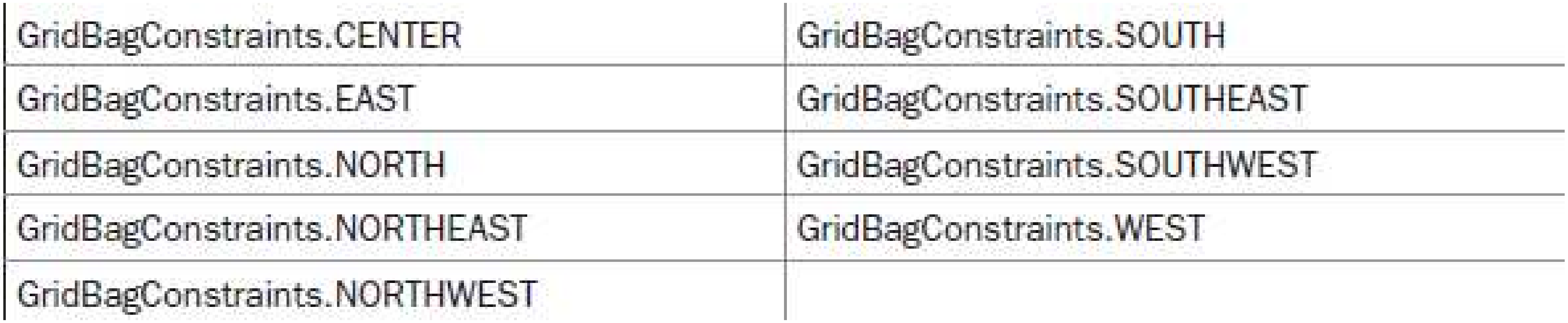
**GridBagConstraints defines several** fields that to govern the size, placement, and spacing of a component.



| Field | Purpose |
|---|---|
| int anchor | Specifies the location of a component within a cell. The default is **GridBagConstraints.CENTER**. |
| int fill | Specifies how a component is resized if the component is smaller than its cell. Valid values are **GridBagConstraints.NONE** (the default), **GridBagConstraints.HORIZONTAL**, **GridBagConstraints.VERTICAL**, **GridBagConstraints.BOTH**. |
| int gridheight | Specifies the height of component in terms of cells. The default is 1. |
| int gridwidth | Specifies the width of component in terms of cells. The default is 1. |
| int gridx | Specifies the X coordinate of the cell to which the component will be added. The default value is **GridBagConstraints.RELATIVE**. |
| int gridy | Specifies the Y coordinate of the cell to which the component will be added. The default value is **GridBagConstraints.RELATIVE**. |
| Insets insets | Specifies the insets. Default insets are all zero. |
| int ipadx | Specifies extra horizontal space that surrounds a component within a cell. The default is 0. |
| int ipady | Specifies extra vertical space that surrounds a component within a cell. The default is 0. |
| double weightx | Specifies a weight value that determines the horizontal spacing between cells and the edges of the container that holds them. The default value is 0.0. The greater the weight, the more space that is allocated. If all values are 0.0, extra space is distributed evenly between the edges of the window. |
| double weighty | Specifies a weight value that determines the vertical spacing between cells and the edges of the container that holds them. The default value is 0.0. The greater the weight, the more space that is allocated. If all values are 0.0, extra space is distributed evenly between the edges of the window. |

# **GridBagLayout**(contd.)

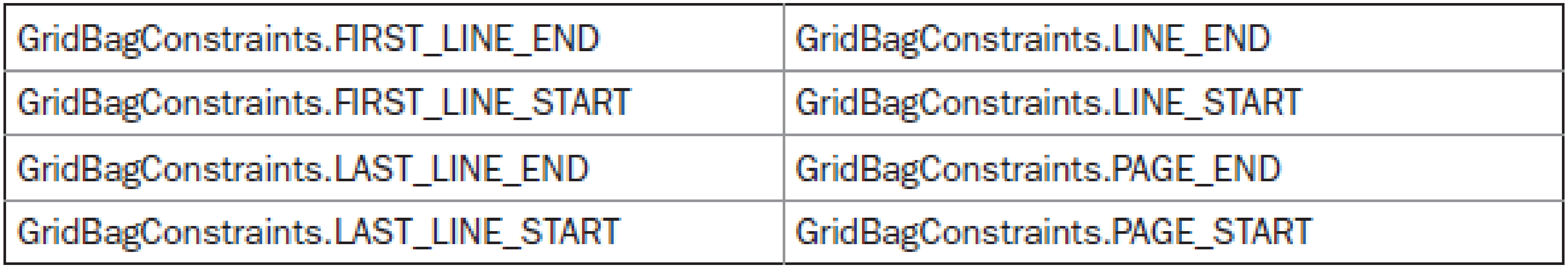
- GridBagConstraints also defines several static fields that contain standard constraint values, such as
  - GridBagConstraints.CENTER
  - GridBagConstraints.VERTICAL
- When a component is smaller than its cell, you can use the anchor field to specify where within the cell the component's top-left corner will be locate



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

# **GridBagLayout**(contd.)

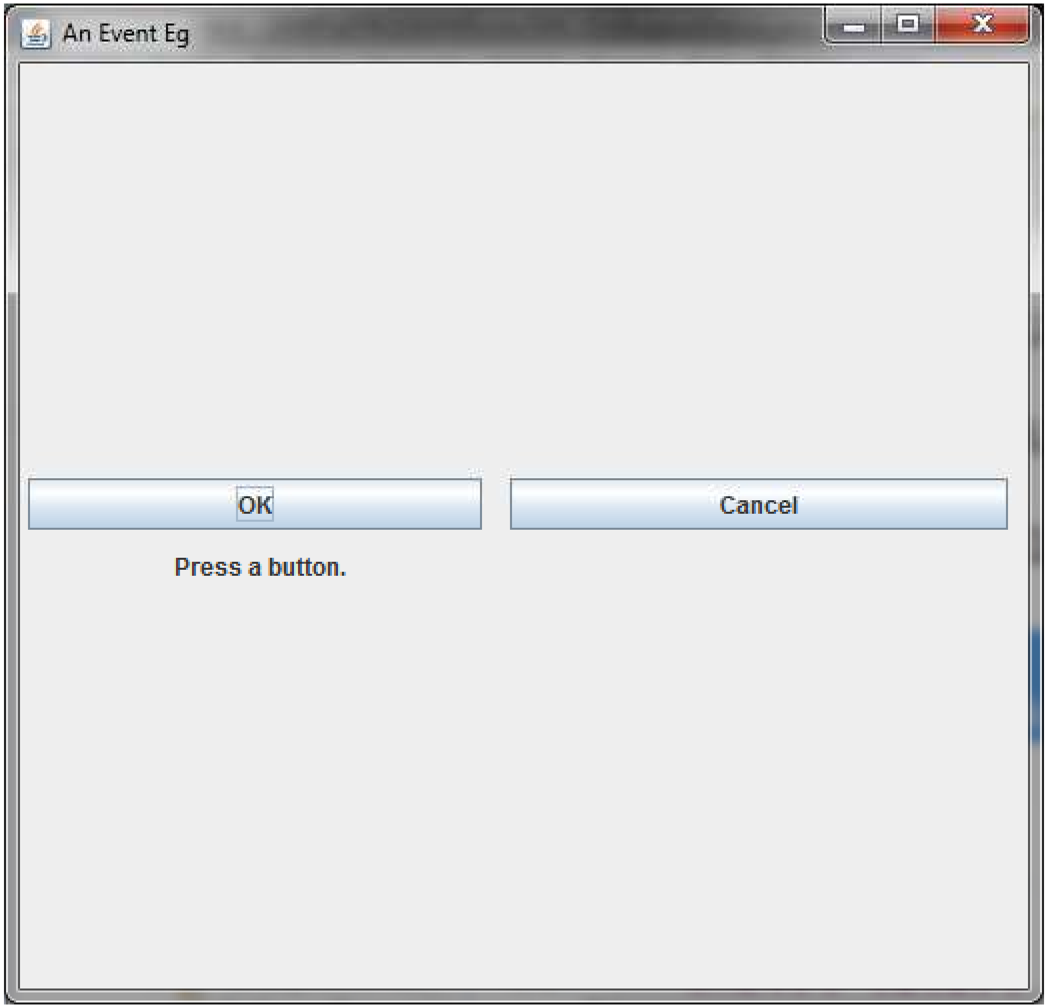
- The second type of values that can be given to anchor is relative, which means the values are relative to the container’s orientation,



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

```
import java.awt.*;       import java.awt.event.*;  import javax.swing.*;
public GridLayoutDemo() {
                jfrm = new JFrame("An Event Eg");
        GridBagLayout gbag = new GridBagLayout();
        GridBagConstraints gbc = new GridBagConstraints();
                jfrm.setLayout(gbag);
                jfrm.setSize(520, 500);
                jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                jbtnOk = new JButton("OK");
                jbtnCancel = new JButton("Cancel");
                jbtnOk.setToolTipText("click");
                jbtnOk.addActionListener(this);
                jbtnCancel.addActionListener(this);
//Define the grid bag. // Use default row weight of 0 for first row.
        gbc.weightx = 1.0;                      // use a column weight of 1
        gbc.ipadx = 200;                        // pad by 200 units
        gbc.insets = new Insets(4, 4, 10, 10);  // inset slightly from top left
        gbc.anchor = GridBagConstraints.NORTH;
        gbc.gridwidth = GridBagConstraints.RELATIVE;
        gbag.setConstraints(jbtnOk, gbc);
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbag.setConstraints(jbtnCancel, gbc);
```

```
                        jlab = new JLabel("Press a button.");

                        jfrm.add(jbtnOk);
                        jfrm.add(jbtnCancel);
                        jfrm.add(jlab);

                        jfrm.setVisible(true);
    }
    public void actionPerformed(ActionEvent ae)
    {
        String s = ae.getActionCommand();
        if(s.equalsIgnoreCase("ok"))
                jlab.setText("OK  pressed.");
        else if(s.equalsIgnoreCase("cancel"))
                jlab.setText("Cancel  pressed.");
    }

}
```



Press a button.

# Reference

- Herbert Schildt, Java: **The Complete Reference, 8/e, Tata McGraw Hill, 2011**.