Distributed Programming in Java

Session: 3

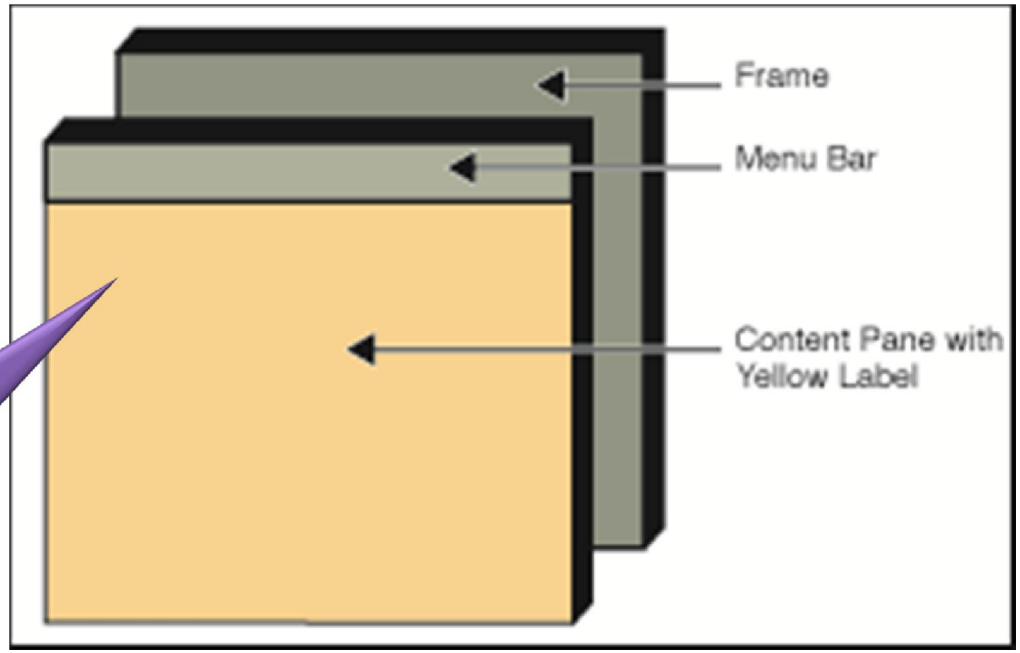
Layout Managers

Objectives

- State the need and purpose of layout manager
- Explain different types of layout manager
- Describe the FlowLayout manager
- Explain how to create and add components to FlowLayout
- Describe the BorderLayout manager
- Describe the GridLayout manager
- Describe the CardLayout manager
- Describe the GridBagLayout manager
- Explain BoxLayout and SpringLayout
- Explain GroupLayout
- Explain Dimension class

Introduction

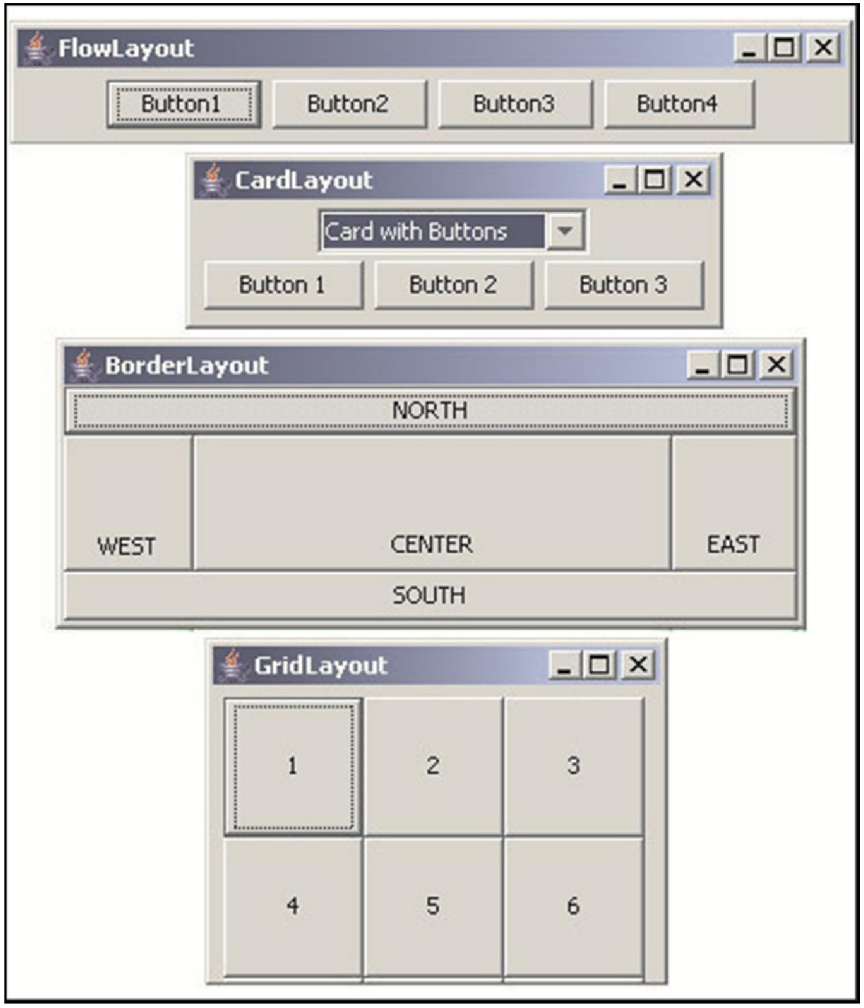
- Top-level containers objects, such as frame or window are:
  - Responsible for providing some default arrangement for the components added into them.
  - Are associated with a content pane that contains the visible components.
  - Figure displays a frame container with a content pane.



The layout managers affect the content pane

Overview of Layout Manager

- A Layout Manager:
  - Is a Java object associated with a container.
  - Governs the placement and size of the component.
  - Simplify the task of adding components to the container.
  - Figure displays the layout managers.
- All containers have a default layout manager associated with it.
- Example: `JFrame` and a `JApplet` has a `BorderLayout` by default.
- When the container is resized, the layout manager is responsible to compute the new location and size of the component as required.



Steps for Using Layout Managers

- Normally a layout manager for a container is set only if the default manager of the container is not desirable.
- The two steps involved in using a layout manager are:
  - First step is to invoke the `setLayout()` method container which is used to set a different layout.
  - The next step in using layout managers is to consider the sizing hints of the components. The sizing hints determine the initial size.
- The following methods are used to specify the sizing hints:
  - **`void setPreferredSize(Dimension preferredSize)`**: Is used to set the preferred size of the component. The layout managers use this size when the component is added to the container.
  - **`void setMinimumSize(Dimension minimumSize)`:** Is used to set the minimum size of the component. The layout managers use this size to determine the minimum size of a component.
  - **`void setMaximumSize(Dimension maximumSize)`**: Is used to set the maximum size of the component. The layout managers use this size to determine the maximum size of a component.

Working of a Layout Manager

- The `pack()` method invokes the `getPreferredSize()` method of the frame to determine the best size for laying out the components.
- The `getPreferredSize()` method is invoked recursively for all the components until the total size of the frame is computed.
- To this computed size, the size required by the edge of the frame, and the menu bar if available is added.
- When the final size is available, the layout manager then lays out the components.

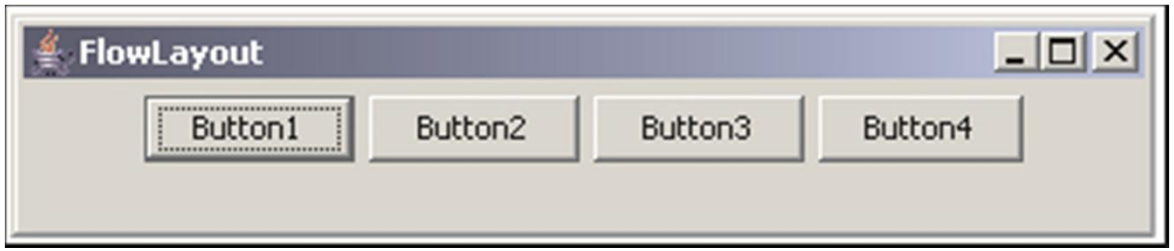
Types of Layout Manager

The different layout managers are as follows:

- **FlowLayout Manager**: Arranges the components from left to right.
- **BorderLayout Manager**: Allows to position components in only five possible locations in a container. The possible locations are East, West, North, South, and Center.
- **GridLayout Manager**: Arranges the components in terms of rows and columns.
- **CardLayout Manager**: Arranges components like a deck of cards. The topmost component is first visible; the rest have to be shuffled.

FlowLayout Manager

- The `FlowLayout` manager lays the components in a row from left to right in a container.
- If there is no space, it continues down in a new row again from left to right.
- The `FlowLayout` manager uses the preferred size of the components when they are laid in a container.
- If the container is wider than necessary for a row of components, then the row is centered by default.
- One can specify the alignment to make it justified left or right.
- One can also specify the horizontal and vertical padding required around the components.
- Figure shows the `FlowLayout` manager.



Create FlowLayout Manager

- The `FlowLayout` manager can be created using one of the following constructors:
  - `FlowLayout()`
  - `FlowLayout (int alignment)`
  - `FlowLayout(int alignment, int horizontal-gap, int vertical-gap)`

Setting the FlowLayout Manager

- To set layout manager, you invoke the `setLayout()` method of the container and pass an instance of the `FlowLayout` class.
- Code Snippet shows how to set the `FlowLayout` manager and add `JLabel` and `JTextField`.

Code Snippet

```
public class Login extends JFrame {
// GUI components declaration
     JLabel lblName;
     JTextField txtName;
 public Login() {
   // Sets the layout of the frame to the flow layout
     setLayout(new FlowLayout());
  // Creates a label with the name "Name"
     lblName = new JLabel("Name");
  // Adds the label to the frame
     getContentPane().add(lblName);
  // Creates a textfield.
     txtName  = new JTextField();
 // Adds the textfield to the frame
     getContentPane().add(txtName);
 }
}
```

Benefits and Drawbacks of FlowLayout Manager

Benefits:

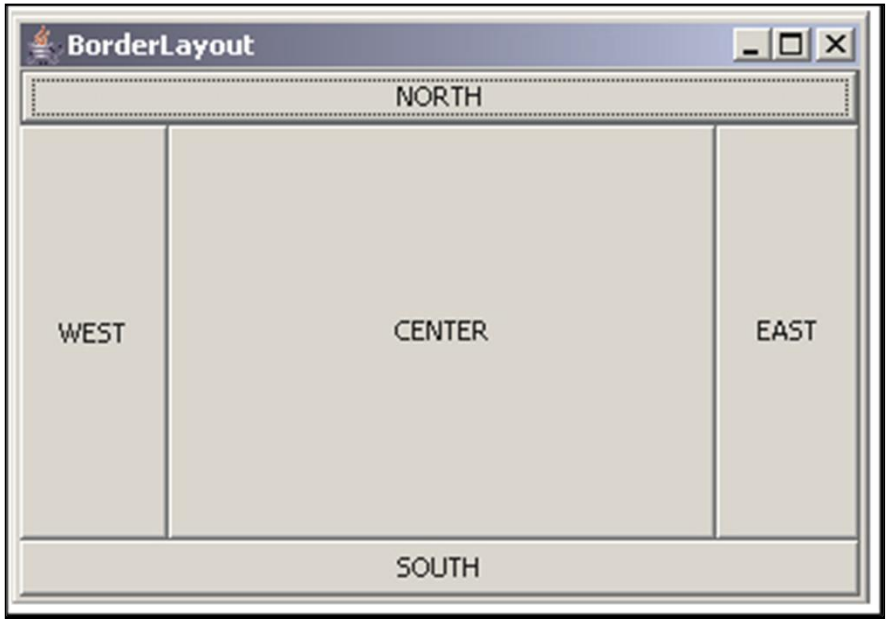
1. Simple to use.
2. Components placed in the center of the container by default.
3. Suitable for `JApplet` which runs as part of a Web Page.

Drawbacks:

1. If the container is resized or looses the original alignment, some components may go up or down depending upon the width.
2. Complex alignment cannot be achieved.

BorderLayout Manager

- The `BorderLayout` manager allows to add components in the east, west, north, south, and center of a container.
- By default, if the direction is not specified then the component is placed in the center.
- In addition, `BorderLayout` supports four relative positioning constants, `BEFORE_FIRST_LINE`, `AFTER_ LAST_LINE`, `BEFORE_LINE_BEGINS`, and `AFTER_LINE_ENDS`.
- Containers such as `JFrame` and `JDialog` have the `BorderLayout` manager as its default layout manager.
- Figure displays the `BorderLayout` manager.



Creating and Setting BorderLayout Manager [1-2]

- The `BorderLayout` manager can be created using one of the following constructors:
  - `BorderLayout()`
  - `BorderLayout(int horizontal-gap, int vertical-gap)`
- If a container does not have a `BorderLayout` manager as its default, one can invoke the `setLayout()` method of the container and pass an instance of the `BorderLayout` class.
- The components are added to the container using following method:
  - `Component add(Component component, int index)`: This method is used to add a component in the specified direction.
- Code Snippet shows how to set the layout the `JPanel` to `BorderLayout` and add components `JLabel` and `JTextArea` to the `JPanel`.

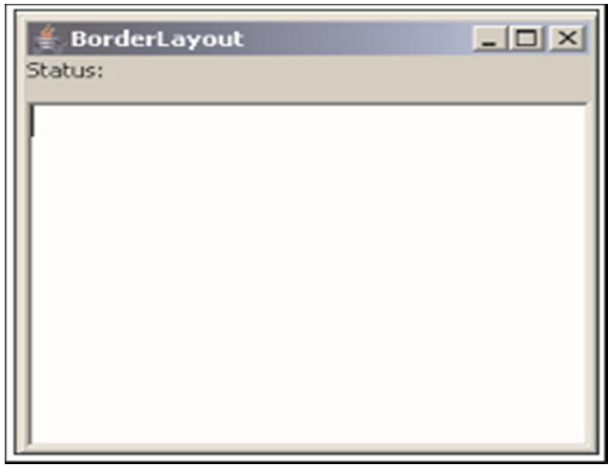
Code Snippet

```
JPanel  pnlPanel;
JLabel lblStatus;
JTextArea txaNotes;
pnlPanel = new JPanel();
```

Creating and Setting BorderLayout Manager [2-2]

```
// Changes the layout of the panel to BorderLayout
   pnlPanel.setLayout(new BorderLayout());
// Creates a label with the name "Status"
   lblStatus = new JLabel("Status");
// Adds the label to the panel in the north direction
   pnlPanel.add(lblStatus, BorderLayout.NORTH);
// Creates a textarea
   txaNotes = new JTextArea(); /
/ Adds the textarea to the panel in the center.
  pnlPanel.add(txaNotes, BorderLayout.CENTER);
. . .
```

- Output:



Benefits and Drawbacks

Benefits:

1. Allows to specify the direction where the component is to be placed.
2. By default, the component is placed in the center.

Drawbacks:

1. Components can be added in only four to five locations in a container.
2. Absolute positioning of components is not possible.

GridLayout Manager

- The `GridLayout` manager places the components in terms of rows and columns.
- Each cell (where the row and column meet) is of the same size.
- If the container is resized, then the available space is again distributed uniformly amongst all the cells.
- `GridLayout` is typically used when all the components in the container are of the same size and are arranged in terms of rows and columns like in a 'calculator'.
- Figure displays the `GridLayout` manager.

Creating and Setting GridLayout Manager [1-2]

- The `GridLayout` manager can be created using any one of the constructors:
  - `GridLayout(int rows, int columns)`
  - `GridLayout(int rows, int columns, int horizontal_gap, int vertical_gap)`
- To set the layout of a container to `GridLayout` manager, invoke its `setLayout()` method and send an instance of the `GridLayout` class as an argument and the `add()` method of the container can be invoked to add components.
- The total number of components added should be equal to the product of number of rows and columns.

Creating and Setting GridLayout Manager [2-2]

- Code Snippet shows how to set the `GridLayout`.

Code Snippet

```
JPanel pnlNumericPad;
JButton  btnOne, btnTwo, btnThree, btnFour;
// Creates a panel
   pnlNumericPad = new JPanel();
// 2 rows and 2 columns
   pnlNumericPad.setLayout(new Gridlayout(2, 2));
// Create buttons 1 to 4
   btnOne = new JButton("1");
   btnTwo = new JButton("2");
   btnThree = new JButton("3");
   btnFour = new JButton("4");
// Add the buttons 1 to 4 to the panel
   pnlNumericPad.add(btnOne);
   pnlNumericPad.add(btnTwo);
   pnlNumericPad.add(btnThree);
   pnlNumericPad.add(btnFour);
```

Benefits and Drawbacks of GridLayout Manager

Benefits:

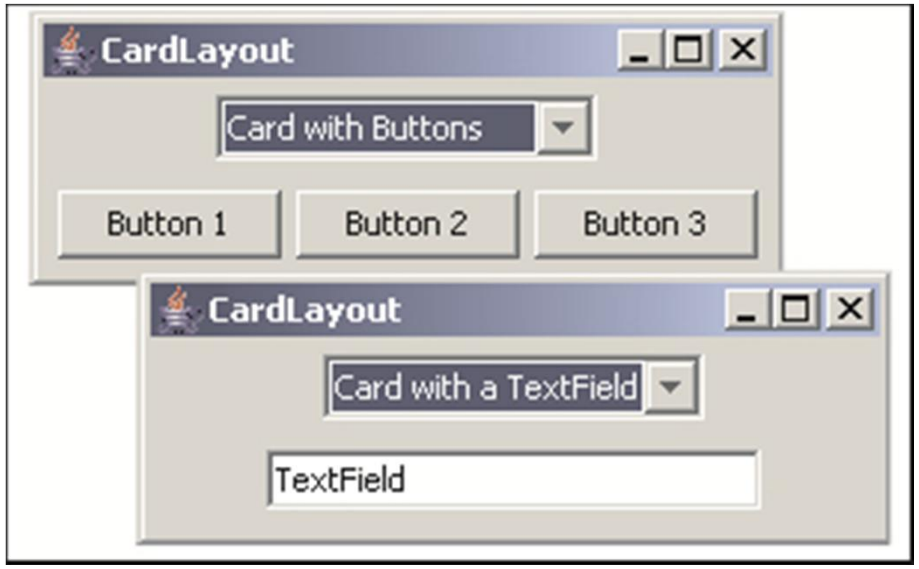
1. Simple to use.
2. Useful for grouping components in a rectangular pattern.
3. Each component is of same size.

Drawbacks:

1. Cannot create a complex layout such as `GridBagLayout` where a cell can span more than one row or column.

CardLayout Manager

- The `CardLayout` manager allows you to stack components one behind another like a deck of cards.
- Only one component is visible at a time and that is the component on top.
- To display other components, one has to flip them based on some event such as button being clicked, or a drop-down to choose a particular one.
- Figure displays the `CardLayout`.



Creating and Setting CardLayout Manager [1-2]

- The `CardLayout` manager can be created using one of the following constructors:
  - `CardLayout()`
  - `CardLayout(int horizontal_gap, int vertical_gap)`
- To set the layout of a container to `CardLayout` manager, invoke the `setLayout()` method of the container.
- An instance of the `CardLayout` class is sent as its argument.
- Once the container is governed by the `CardLayout` manager, you create the cards.
- These cards are typically panels with their associated components and then add them to the container.
- The `add()` method of the container accepts two arguments namely, card and string specifying the name of the card.

Creating and Setting CardLayout Manager [2-2]

- Code Snippet shows how to set the layout of the panel to card layout and add the panels to the main panel.

Code Snippet

```
CardLayout cardLayout;
JPanel pnlSubjects;
JPanel pnlEnglish, pnlScience, pnlMaths;
// Creates the card layout manager
   cardLayout = new CardLayout();
// Creates a main panel
   pnlSubjects = new JPanel();
// Sets the layout of the panel
   pnlSubjects.setLayout(cardLayout);
// Creates panels representing three cards
     pnlEnglish = new JPanel();
     pnlScience = new JPanel();
      pnlMaths = new JPanel();
// Code to add the components to each card
   . . . . . .
// Add the card to the main panel
   pnlSubjects.add(pnlEnglish, "English");
   pnlSubjects.add(pnlScience, "Science");
   pnlSubjects.add(pnlMaths, "Maths");
```

Flipping Components

- The components of a `CardLayout` manager can be flipped by the following methods:
  - **`void first(Container parent)`**: Makes the first card of the container visible. Example: `cl.first(pnlCards);`
  - **`void next(Container parent)`**: Makes the next card of the container visible. If the currently visible card is the last one, this method flips to the first card. Example: `cl.next(pnlCards);`
  - **`void previous(Container parent)`**: Makes the previous card of the container visible. If the currently visible card is the first one, this method flips to the last card. Example: `cl.previous(pnlCards);`
  - **`void last(Container parent)`**: Makes the last card of the container visible. Example: `cl.last(pnlCards);`

Benefits and Drawbacks of CardLayout Manager

Benefits:

1. Allows several containers and their associated components to share the same space in the container.

Drawbacks:

1. Not visually appealing as a Tabbed pane.
2. Requires other components such as button or drop-down to flip through.

Additional Layout Manager

- The additional layout are:
  - `GridBagLayout`
  - `Absolute Positioning`

GridBagLayout Manager

- The `GridBagLayout` class is the most complex and yet the most flexible layout manager.
- The `GridBagLayout` manager allows the user to place components in rows and columns.
- However, these components can span multiple rows or columns.
- All the rows need not necessarily have the same height, and all the columns need not necessarily have the same width.
- The `GridBagLayout` manager works by specifying constraints on each components before it is being laid out.
- To specify these constraints, Swing provides a `GridBagConstraints` class.
- The `GridBagConstraints` class has several constraint variables and objects which allows the user to specify the constraints on the components, before adding components to a container.

Constraints of GridBagConstraints Class [1-5]

- **gridx and gridy**: These constraints specify the row and column. The leftmost column has `gridx = 0`, the top row has `gridy = 0`.
- **gridwidth and gridheight**: `gridwidth` specifies how many columns the component will span. `gridheight` specifies how many rows the component will span.
- **fill**: The `fill` constraint determines whether the component will resize if more space is available then required. Valid values are: `GridBagConstraints.NONE`, `GridBagConstraints.HORIZONTAL`, `GridBagConstraints.VERTICAL`, and `GridBagConstraints.BOTH`.
- **ipadx and ipady**: These constraints specify the internal padding to add to the minimum size of the component if default values are zero. If these constraints are specified, then the width of the component will be at least its minimum width plus `ipadx*2` pixels, and the height of the component will be at least its minimum height plus `ipady*2` pixels.

Constraints of GridBagConstraints Class [2-5]

- **`insets`**: This constraint is used to specify the external padding. The `java.awt.Insets` class is used to specify the gaps in the order top, left, bottom, and right.
- **`anchor`**: This constraint is used when the component is smaller than the available display area. This constraint determines where the component is to be placed within the display area. The `GridBagConstraints` class defines constants such as `CENTER`, `EAST`, `WEST`, `NORTH`, `SOUTH`, `NORTHEAST`, `SOUTHEAST`, `NORTHWEST`, and `SOUTHWEST`.
- **`weightx` and `weighty`**: The constraint `weightx` decides how to distribute space among columns. The constraint `weighty` decides how to distribute space among rows. This constraint is helpful when the container is resized. If this constraint is not set, then all the components will be clubbed at the center of the container.

Constraints of GridBagConstraints Class [3-5]

- Code Snippet shows how to use the `GridBagLayout` manager to add seven buttons to produce a layout.

Code Snippet

```
Container  c;
JButton btnButton1, btnButton2,btnButton3,btnButton4;
JButton btnButton5,btnButton6,btnButton7;

GridBagConstraints gbc;
. . .
c = getContentPane();

c.setLayout(new GridBagLayout());
gbc = new GridBagConstraints();
gbc.gridx = 0;
gbc.gridy = 0;

gbc.gridwidth = 1;
gbc.gridheight = 1;
```

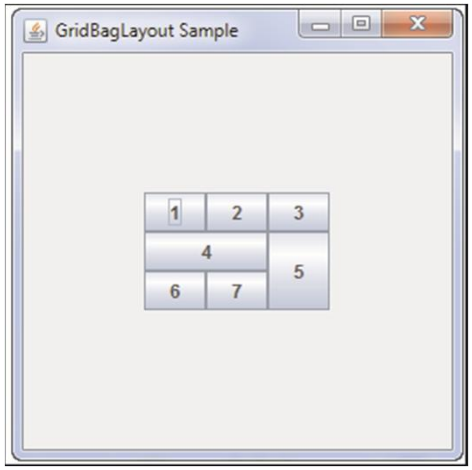
Constraints of GridBagConstraints Class [4-5]

```
btnButton1 = new JButton("1");
c.add(btnButton1, gbc);
gbc.gridx = 1;

btnButton2 = new JButton("2");
c.add(btnButton2, gbc);
gbc.gridx = 2;

btnButton3 = new JButton("3");
c.add(btnButton3, gbc);
gbc.gridx = 0;
gbc.gridy = 1;
gbc.gridwidth = 2;
gbc.gridheight = 1;
gbc.fill = GridBagConstraints.HORIZONTAL;

btnButton4 = new JButton("4");
c.add(btnButton4, gbc);  gbc.gridx = 2;
gbc.gridwidth = 1;
gbc.gridheight = 2;
gbc.fill = GridBagConstraints.VERTICAL;
```

Constraints of GridBagConstraints Class [5-5]

```
btnButton5 = new JButton("5");
c.add(btnButton5, gbc);
gbc.gridx = 0;
gbc.gridy = 2;
gbc.gridwidth = 1;
gbc.gridheight = 1;

btnButton6 = new JButton("6");
c.add(btnButton6, gbc);
gbc.gridx = 1;

btnButton7 = new JButton("7");
c.add(btnButton7, gbc);
setVisible(true);
```

- Output:



Absolute Positioning (null Layout)

- The absolute positing or null layout allows the user to specify the bounds of the component before adding it to the container.
- These bounds include the x and y position in pixels and the width and height in pixels of the component.

  Syntax:

  ```
  setLayout(null);
  ```

- Code Snippet demonstrates how to add components using absolute positioning.

Code Snippet

```
Container container;
JButton btnClick;
JTextField txtText;
. . .
container = getContentPane();
container.setLayout(null);
btnClick = new JButton("Click");
btnClick.setBounds(50,50,75,25);
container.add(btnClick);
txtText = new JTextField();
txtText.setBounds(130,50,200,25);
container.add(txtText);
```

New Layout Manager

- New layout managers have been added to Swings:
  - `BoxLayout`
  - `SpringLayout`
  - `GroupLayout`

BoxLayout Manager [1-2]

- The `BoxLayout` is similar to a `FlowLayout` manager except that it places the components on top of each other in one column, or places them horizontally in a row.
- `BoxLayout` depends on the following axis parameters for the layout:

1. `X_AXIS`: Components are arranged horizontally from left to right.
2. `Y_AXIS`: Components are arranged vertically from top to bottom.
3. `Line_AXIS`: Components are arranged the way words are arranged in a page.
4. `Page_AXIS`: Components are arranged the way text lines are arranged in a page.

BoxLayout Manager [2-2]

- Code Snippet demonstrates how to add components using `BoxLayout` manager by setting the `Y_AXIS` parameter.

Code Snippet

```
. . .
public class BoxLayoutManager {
  public BoxLayoutManager(){  . . .  }
  public static void main(String args[]) {
   JPanel container = new JPanel();
   container.setBorder(BorderFactory.createTitledBorder(" BoxLayout"));
   BoxLayout layout = new BoxLayout(container, BoxLayout.Y_AXIS);
   container.setLayout(layout);
   JButton button1 = new JButton("Button1");
   container.add(button1);
   button2 = new JButton("Button2");
   container.add(button2);
   button3 = new JButton("Button3");
   container.add(button3);
    . . .
  }
}
```

SpringLayout [1-5]

- A `SpringLayout` works by defining relationships or constraints between the edges of components.
- Distances between edges are represented by spring objects.
- Each spring has four properties namely, minimum, preferred, and maximum values, and its actual (current) value.
- The springs associated with each component are collected into a `SpringLayout.Constraints` object.
- Each constraint controls the vertical or horizontal distance between edges of two components.
- The edges can be attributed to any child of the container or to the container itself.
- The x-coordinate of a component can be expressed using a constraint that controls the distance between the west and east edges of the component.
- The y-coordinate of a component can be expressed by setting the distance between the north edge of the component and the north edge of its container.

SpringLayout [2-5]

- The method `putConstraint()` is used to set the constraint between edges of two components or between edges of a component and the container.

  Syntax:

  ```
  public void putConstraint(String edge1, Component comp1, int distance, String edge2, Component c2)
  ```

- Code Snippet demonstrates how to add components using `SpringLayout`

Code Snippet

```
. . .
/**  Defining SpringLayout manager **/
public class SpringLayoutManager extends JFrame{
 public SpringLayoutManager(String title) {
  super(title);
  Container content_pane = this.getContentPane();
  // Create an instance of SpringPanel.
     SpringPanel spring_panel = new SpringPanel();
  // Add it to frame
     content_pane.add (spring_panel);
```

SpringLayout [3-5]

```
    setSize(300, 200);
    setVisible(true);
  }
  public static void main(String[] args){
   SpringLayoutManager slm = new SpringLayoutManager("Spring Layout
    Manager");
  }
}

/** Layout five buttons using a SpringLayout Manager. **/

class SpringPanel extends JPanel {
/**
     * The Constructor creates 5 buttons
     * and constrains each to a particular position relative  to the
     * panel.
 **/
    SpringPanel () {
     SpringLayout layout = new SpringLayout ();
     setLayout (layout);
     JButton btn1  = new JButton ("Button1");
```

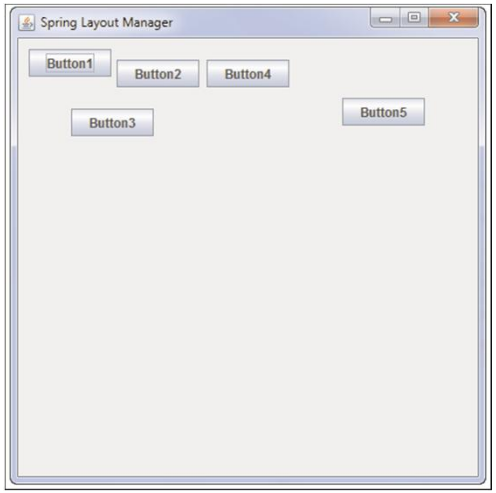
SpringLayout [4-5]

```
   JButton btn2  = new JButton ("Button2");
   JButton btn3  = new JButton ("Button3");
   JButton btn4  = new JButton ("Button4");
   JButton btn5  = new JButton ("Button5");
   add (btn1);
   add (btn2);
   add (btn3);
   add (btn4);
   add (btn5);

// Set the distances between the edges . Put the first button at pixel
// co-ordinates (10,10) relative to the panel's frame
   layout.putConstraint (SpringLayout.WEST, btn1, 10, SpringLayout.WEST, this);
   layout.putConstraint (SpringLayout.NORTH, btn1, 10, SpringLayout.NORTH,
this);

// Put the second button 5 pixels to the right of the first button and 20
// pixels below the top panel edge.
   layout.putConstraint (SpringLayout.WEST, btn2, 5, SpringLayout.EAST, btn1);
   layout.putConstraint (SpringLayout.NORTH, btn2, 20,SpringLayout.NORTH,
this);
```

SpringLayout [5-5]

```
// Put the third button 50 pixels to the left of the
// panel edge and 20 pixels above the second button.
   layout.putConstraint (SpringLayout.WEST, btn3,50, SpringLayout.WEST, this);
   layout.putConstraint (SpringLayout.NORTH, btn3, 20, SpringLayout.SOUTH, btn2);

// Put the fourth button 50 pixels to the right of the
// third button  and 20 pixels below the top panel edge.
   layout.putConstraint (SpringLayout.WEST, btn4,50, SpringLayout.EAST, btn3);
   layout.putConstraint (SpringLayout.NORTH, btn4,20, SpringLayout.NORTH, this);
   layout.putConstraint (SpringLayout.WEST, btn5, 50, SpringLayout.EAST, btn4);
   layout.putConstraint (SpringLayout.NORTH, btn5, 10,SpringLayout.SOUTH, btn4);
 }
}
```

- Output:



GroupLayout Class [1-2]

- Helps in grouping components for better positioning within a container.
- The components can be grouped hierarchically in sequential groups or parallel groups.
- The layout for `GroupLayout` class is defined independently for each dimension.
- Focusing on one dimension at a time is easier than handling both at the same time.
- It is important to define each component twice in this layout or else the `GroupLayout` class will throw an exception.
- `GroupLayout` class uses two types of arrangements - sequential and parallel.
- **Sequential Arrangement**
  - In a sequential group, the components are placed one after another, similar to `BoxLayout` or `FlowLayout` along one axis. The position of each component is relative to the previous component.
- **Parallel Arrangement**
  - Parallel grouping arranges components on top of each other in the same space. Alignment of the components can be specified along the vertical axis or the horizontal axis or both.

GroupLayout Class [2-2]

- Code Snippet shows how to layer three components in a sequence.

Code Snippet

```
horizontal layout = sequential group { c1, c2, c3 }
vertical layout = parallel group (BASELINE) { c1, c2, c3}
```

- Code Snippet shows how to add a component parallel to the component c3.

Code Snippet

```
horizontal layout = sequential group { c1, c2, parallel
group (LEFT) { c3, c4 } }
vertical layout = sequential group { parallel group
(BASELINE) { c1, c2, c3 }, c4 }
```

Dimension Class

- Many of the methods of the Swing components which receives or returns size uses an object of the `Dimension` class.
- This is a convenience class to encapsulate the width and height of the component.
- The `Dimension` class takes two integers as parameter, the width and the height.

Syntax:

`getWidth()`: Returns width of this instance returned by the method in double precision.

`getHeight()`: Returns height of this instance returned by the method in double precision.

- Code Snippet demonstrates the use of `Dimension` class in a `JButton` instance, **`btnOk`**.

Code Snippet

```
JButton btnOk;
. . .
btnOk = new JButton("OK");
btnOk.setMinimumSize(new Dimension(50,20));
btnOk.setMaximumSize(new Dimension(70,25));
btnOk.setPreferredSize(new Dimension(60,25));
```

Custom Layout Manager [1-7]

- A custom layout manager is created only if none of the existing layout managers meet the given requirements.
- The `LayoutManager` interface has five methods that must be implemented by a custom layout that are as follows:
  - `void addLayoutComponent(String name, Component comp)`
  - `void removeLayoutComponent(Component)`
  - `Dimension preferredLayoutSize(Container)`
  - `Dimension minimumLayoutSize(Container)`
  - `void layoutContainer(Container)`
- Code Snippet demonstrates how to create the diagonal layout.

Code Snippet

```
import java.awt.*;
public class DiagonalLayout implements LayoutManager {
private int vgap;
private int minWidth = 0, minHeight = 0;
private int preferredWidth = 0, preferredHeight = 0;
private boolean sizeUnknown = true;
```

Custom Layout Manager [2-7]

```
 public DiagonalLayout() {
    this(5);
 }
 public DiagonalLayout(int v) {
   vgap = v;
 }
/* Required by LayoutManager. */
   public void addLayoutComponent(String name, Component comp) { . . . }
/* Required by LayoutManager. */
   public void removeLayoutComponent(Component comp) { . . . }
   private void setSizes(Container parent) {
     int nComps = parent.getComponentCount();
     Dimension d = null;
   // Reset preferred/minimum width and height.
      preferredWidth = 0;
      preferredHeight = 0;
      minWidth = 0;
      minHeight = 0;
     for (int i = 0; i < nComps; i++) {
       Component c = parent.getComponent(i);
        if (c.isVisible()) {
           d = c.getPreferredSize();
```

Custom Layout Manager [3-7]

```
 if (i > 0) {
    preferredWidth += d.width/2;
    preferredHeight += vgap;
  } else {
    preferredWidth = d.width; }

    preferredHeight += d.height;
    minWidth = Math.max(c.getMinimumSize().width, minWidth);
    minHeight = preferredHeight;
 } }
}
 /* Required by LayoutManager. */
    public Dimension preferredLayoutSize(Container parent) {
       Dimension dim = new Dimension(0, 0);
       int nComps = parent.getComponentCount();
       setSizes(parent);
   //Always add the container's insets!
       Insets insets = parent.getInsets();
       dim.width = preferredWidth + insets.left + insets.right;
       dim.height = preferredHeight + insets.top + insets.bottom;
       sizeUnknown = false;
       return dim;
}
```

Custom Layout Manager [4-7]

```
 /* Required by LayoutManager. */
   public Dimension minimumLayoutSize(Container parent) {
    Dimension dim = new Dimension(0, 0);
    int nComps = parent.getComponentCount();
    //Always add the container's insets!
       Insets insets = parent.getInsets();
       dim.width = minWidth + insets.left + insets.right;
       dim.height = minHeight + insets.top + insets.bottom;
       sizeUnknown = false;
     return dim;
   }

/* Required by LayoutManager. */
/*
* This is called when the panel is first displayed,
* and every time its size changes.
* Note: You CAN'T assume preferredLayoutSize or
* minimumLayoutSize will be called -- in the case
* of applets, at least, they probably won't be.
*/
```

Custom Layout Manager [5-7]

```
public void layoutContainer(Container parent) {
  Insets insets = parent.getInsets();
  int maxWidth = parent.getWidth() - (insets.left + insets.right);
  int maxHeight = parent.getHeight() - (insets.top + insets.bottom);
  int nComps = parent.getComponentCount();
  int previousWidth = 0, previousHeight = 0;
  int x = 0, y = insets.top;
  int rowh = 0, start = 0;
  int xFudge = 0, yFudge = 0;
  boolean oneColumn = false;
 // Go through the components' sizes, if neither
 // preferredLayoutSize nor minimumLayoutSize has been called.
    if (sizeUnknown) {
      setSizes(parent);
    }
     if (maxWidth <= minWidth) {
        oneColumn = true;
     }
    if (maxWidth != preferredWidth) {
       xFudge = (maxWidth - preferredWidth)/(nComps - 1);    }
  x += previousWidth/2 + xFudge;
}
```

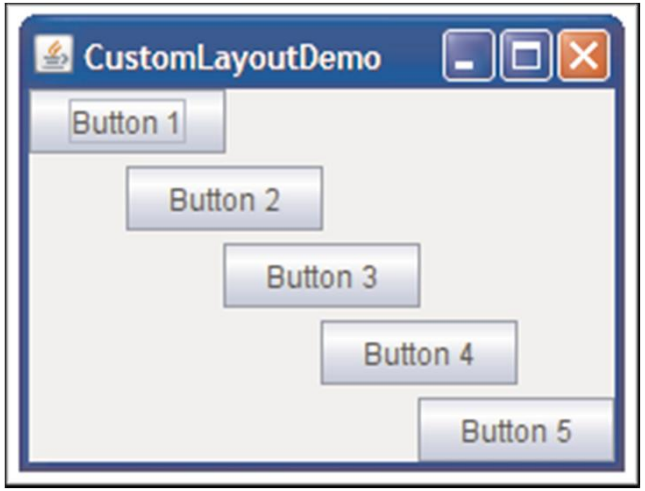
Custom Layout Manager [6-7]

```
 y += previousHeight + vgap + yFudge;
}
  if (maxHeight > preferredHeight) {
     yFudge = (maxHeight - preferredHeight)/(nComps - 1);
  }
  for (int i = 0 ; i < nComps ; i++) {
      Component c = parent.getComponent(i);
      if (c.isVisible()) {
         Dimension d = c.getPreferredSize();
     // increase x and y, if appropriate
         if (i > 0) {
         if (!oneColumn) {
    // If x is too large,
        if ((!oneColumn) && (x + d.width) > (parent.getWidth() - insets.right))
        {
            // reduce x to a reasonable number.
              x = parent.getWidth() - insets.bottom - d.width;
        }
    // If y is too large,
        if ((y + d.height) > (parent.getHeight() - insets.bottom)) {
         // do nothing. Another choice would be to do what we do to x.
        }
```

Custom Layout Manager [7-7]

```
// Set the component's size and position.
        c.setBounds(x, y, d.width, d.height);
        previousWidth = d.width;
        previousHeight = d.height;
    }
} }

 public String toString() {
    String str = "";
    return getClass().getName() + "[vgap=" + vgap + str + "]";
 }
}
```

- Output:



Decorate Swing Components

- Swing components can be decorated by specifying a look and feel such as Windows.
- **`JLayer` Class**:
  - Acts as a universal decorator for all swing components.
  - Allows the user to draw on components and respond to the component events without disturbing the integrity of the underlying components.
- To decorate the component, perform the following steps:
  - Create the target component.
  - Create an instance of a `LayerUI` subclass to do the drawing.
  - Create a `JLayer` object that wraps the target and the `LayerUI` object.
  - Use the `JLayer` object in your user interface just as you would use the target component.
  - Code Snippet shows how to decorate the `JPanel` object.

Code Snippet

```
JFrame f = new JFrame();
JPanel panel = createPanel();
f.add (panel);
LayerUI<JPanel> layerUI = new MyLayerUISubclass();
JLayer<JPanel> jlayer = new JLayer<JPanel>(panel, layerUI);
f.add (jlayer);
```

LayerUI Class

- Custom decoration and event handling for a `JLayer` object is taken care of by the `LayerUI` class.
- The `JLayer` class is usually generic with the exact type of its view component.
- On the other hand, the `LayerUI` class is designed for use with `JLayer` classes of its generic parameter or any of its ancestors.
- Code Snippet shows how to paint a transparent color gradient on a component.

Code Snippet

```
class WallpaperLayerUI extends LayerUI<JComponent> {
  @Override
  public void paint(Graphics g, JComponent c) {
    super.paint(g, c);
    Graphics2D g2 = (Graphics2D) g.create();
    int w = c.getWidth();
    int h = c.getHeight();
    g2.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_OVER,
.5f));
    g2.setPaint(new GradientPaint(0, 0, Color.yellow, 0, h, Color.red));
    g2.fillRect(0, 0, w, h);
    g2.dispose();
}
}
```

Creating Translucent and Shaped Windows [1-5]

- Java Swing also supports creation of translucent and shaped windows in applications.
  - **Translucency for windows**: Make the full window translucent with a specified alpha level.
  - **Per pixel translucency**: Make part of window translucent.
  - **Shaped windows**: Create windows with a certain shape such as circle, oval, and so on.
  - **To Implement Uniform Translucency**
    - Invoking the `setOpacity(float)` method allows the user to create a window where each pixel has the same translucency.
    - Code Snippet shows how to create a window with 55% opaqueness.

Code Snippet

```
// Create window with 55% opaqueness.
package com.mycompany.java7.swing;
import javax.swing.*;
import java.awt.*;
public class TranslucentWindowExample extends JFrame {
   super("TranslucentWindow");
   setSize(300, 200);
   setDefaultCloseOperation(EXIT_ON_CLOSE);
```

Creating Translucent and Shaped Windows [2-5]

```
 getRootPane().setDoubleBuffered(false);
 setOpacity(0.55f);
 setLocationRelativeTo(null);
 setVisible(true);
}


public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable(){
       @Override
        public void run() {
          GraphicsEnvironment ge = GraphicsEnvironment.
getLocalGraphicsEnvironment();
        if(ge.getDefaultScreenDevice().isWindowTranslucencySupported(GraphicsD
evice.WindowTranslucency.TRANSLUCENT)){

            new TranslucentWindow();
        }
       }});
  }
}
```

Creating Translucent and Shaped Windows [3-5]

- **To Implement Per Pixel Translucency**
  - Implementing per pixel translucency involves defining alpha values for the area that a window occupies.
  - When the alpha value is zero, the window is fully transparent.
  - When the alpha value is 255, the window is fully opaque.
  - The `gradientPaint` class allows the user to create a smooth interpolation between alpha values.
  - The `isOpaque` method helps to find out if a window is using per pixel translucency.
  - Here, are the steps required to implement the `isOpaque` method:
    - Invoke `setBackground(new Color(0,0,0,0))` on the window.
    - Create a `JPanel` instance that overrides the `paintComponent` method.
    - In the `paintComponent` method, create a `GradientPaint` instance.
    - Set the `GradientPaint` instance as the panel's paint method.

Creating Translucent and Shaped Windows [4-5]

- Code Snippet shows how to create a gradient window.

Code Snippet

```
setBackground(new Color(0,0,0,0));
setSize(new Dimension(300,200));
setLocationRelativeTo(null);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  JPanel panel = new JPanel() {
   @Override
       protected void paintComponent(Graphics g) {
         if (g instanceof Graphics2D) {
            final int R = 240;
            final int G = 240;
            final int B = 240;
Paint p = new GradientPaint(0.0f, 0.0f, new Color(R, G, B, 0),
0.0f, getHeight(), new Color(R, G, B, 255), true);
Graphics2D g2d = (Graphics2D)g;
         g2d.setPaint(p);
        g2d.fillRect(0, 0, getWidth(), getHeight());
      }
   }};
```

Creating Translucent and Shaped Windows [5-5]

- **To Create a Shaped Window**
  - The `setShape()` method allows the user to create a shaped window.
  - To set the window's shape, the best practice is to invoke `setShape()` in the `componentResized()` method of the component event listener.
  - This makes sure that the shape is calculated correctly for the actual size of the window.
  - Code Snippet shows how to implement shaped windows.

Code Snippet

```
addComponentListener(new ComponentAdapter() {
   // Give the window an elliptical shape.
      public void componentResized(ComponentEvent e) {
        SetShape(new Ellipse2D.Double(0,0,getWidth(),getHeight()));
      }});
setUndecorated(true);
GraphicsEnvironment ge =
GraphicsEnvironment.getLocalGraphicsEnvironment();
GraphicsDevice gd = ge.getDefaultScreenDevice();
 if (!gd.isWindowTranslucencySupported(PERPIXEL_TRANSPARENT)) {
      System.err.println("Shaped windows are not supported");
      System.exit(0);
 }
```

Responding to Events

- All the events from the related components need to be received and processed.
- `LayerUI` subclass can receive all the events, but it has to register for specific event.
- The `setLayerEventMask()` method is called to indicate the `LayerUI` subclass's willingness in receiving the mouse and mouse motion events.
- Code Snippet shows a portion of `LayerUI` subclass that registers to receive mouse events.

Code Snippet

```
public void installUI(JComponent c) {
      super.installUI(c);
JLayer jlayer = (JLayer)c;
 jlayer.setLayerEventMask(
   AWTEvent.MOUSE_EVENT_MASK | AWTEvent.MOUSE_MOTION_EVENT_MASK
  );
```

Summary [1-2]

- A layout manager is a Java object associated with a container which governs the placement and size of the component when it is added to a container.
- The FlowLayout manager lays the components in a row from left to right in a container. The FlowLayout manager uses the preferred size of the components when they are laid in a container.
- The BorderLayout manager allows you to add components in the east, west, north, south, and center direction of a container. By default, if the direction is not specified, then the component is placed in the center.
- The GridLayout manager places the components in terms of rows and columns. Each cell where the row and column meet is of the same size. If the container is resized, then the available space is again distributed uniformly amongst all the cells.

Summary [2-2]

- The CardLayout manager allows you to stack components one behind another like a deck of cards. Only one component is visible at a time and that is the component on top. To display other components one has to flip them.
- The CustomLayout manager is built only when no other existing layout manager meets the given requirements. The custom layout manager is built under a class that implements the LayoutManager interface.
- There are new layout managers in Swing, such as BoxLayout Manager, SpringLayout Manager, and GroupLayout class. These allow you to place components in columns; rows are by defining relationships between components.
- The javax.swing.JLayer class acts as a universal decorator for all swing components.