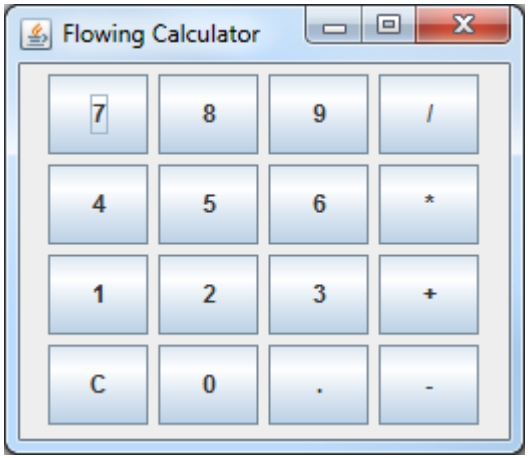
GUI Programming III

Layout Management

What Could Happen to These Calculator Buttons?

```
public class FlowingCalculator extends JFrame {
  /** Default constructor */
  public FlowingCalculator() {
      JPanel calculator = new JPanel();
      add(calculator);
      // add buttons to the panel
      JButton jtn;
      Dimension jtnSize = new Dimension(50, 40);
      for (int i = 7; i <= 9; i++){
          jtn = new JButton(Integer.toString(i));
          jtn.setPreferredSize(jtnSize);
          calculator.add(jtn);
      }
      calculator.add(jtn = new JButton("/"));
      jtn.setPreferredSize(jtnSize);

      for (int i = 4; i <= 6; i++){
          jtn = new JButton(Integer.toString(i));
          jtn.setPreferredSize(jtnSize);
          calculator.add(jtn);
      }
      calculator.add(jtn = new JButton("*"));
      jtn.setPreferredSize(jtnSize);
      for (int i = 1; i <= 3; i++){
          jtn = new JButton(Integer.toString(i));
          jtn.setPreferredSize(jtnSize);
          calculator.add(jtn);
      }
      calculator.add(jtn = new JButton("+"));
      . . .
  }
```



Layout Management

- Java's layout managers provide a level of abstraction to automatically map your user interface on all window systems.
- The GUI components are placed in containers. Each container has a layout manager to arrange the UI components within the container.

Layout Managers

- Provided for arranging GUI components
  - Arrange components in accordance with its own rules and property settings, and the constraints associated with each components
  - Each layout manager has its own set of rules
- Provide basic layout capabilities
  - Size
  - Positions of components
- Process layout details

- Each container has a default layout manager
  - You can always set to a new one
- Layout manager classes implement the **`LayoutManager`** interface
- Programmer can concentrate on basic “look and feel”

Common Layout Managers

Layout manager	Description
`FlowLayout`	• Default for **`JPanel`**. • Places components sequentially (left to right) in the order they were added. • It is also possible to specify the order of the components by using the **`Container`** method **`add`**, which takes a **`Component`** and an integer index position as arguments.
`BorderLayout`	• Default for the content pane of **`JFrames`** (and other windows) and **`JApplets`**. • Arranges the components into five areas: **`NORTH`**, **`SOUTH`**, **`EAST`**, **`WEST`** and **`CENTER`**.
`GridLayout`	• Arranges the components into rows and columns.

Size of a Component in a Container

- Some layout managers have properties that can affect the sizing and location of the components
- The size of a component is determined by many factors, such as
  - The type of layout manager
  - Layout constraints associated with each component
  - Size of the container
  - Certain properties common to all components
    - **`preferredSize, minimumSize, maximumSize, alignmentX and alginmentY`**
    - The **`preferedSize`** proparty indicates the ideal size; it may or may not be considered by a layout manager

Space Between Components

Three factors influence the amount of space between visible components in a container:

- The layout manager
  - Some layout managers automatically put space between components; others do not. Some let you specify the amount of space between components
- Invisible components
  - You can create lightweight components that perform no painting, but that can take up space in the GUI. Often, you use invisible components in containers controlled by **`BoxLayout`**.
- Empty borders
  - No matter what the layout manager, you can affect the apparent amount of space between components by adding empty borders to components.

Setting Layout Manager

- Using the constructor of a container

eg:

```
JPanel panel = new JPanel(new Borderlayout());
```

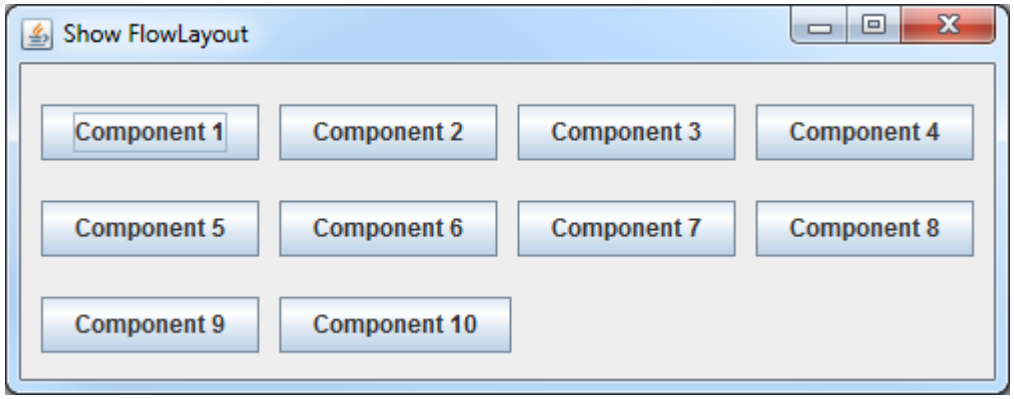
- Using the **`setLayout()`** method of a container

eg:

```
JPanel panel = new Jpanel();
panel.setLayout(new Borderlayout());
```

FlowLayout Manager

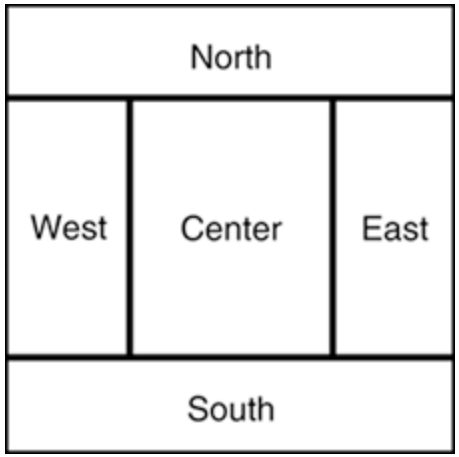
- The most basic layout manager; default for **JPanel**
- With a centred alignment and horizontal and vertical gaps
  - with the default size of 5 pixels
- GUI components placed in container from left to right




```
// Set FlowLayout, aligned left with horizontal gap 10
// and vertical gap 20 between components
container.setLayout(new FlowLayout(FlowLayout.LEFT, 10, 20));

// Add buttons to the frame
for (int i = 1; i <= 10; i++)
  container.add(new JButton("Component " + i));
```

BorderLayout Manager

- The **BorderLayout** manager divides the container into five areas



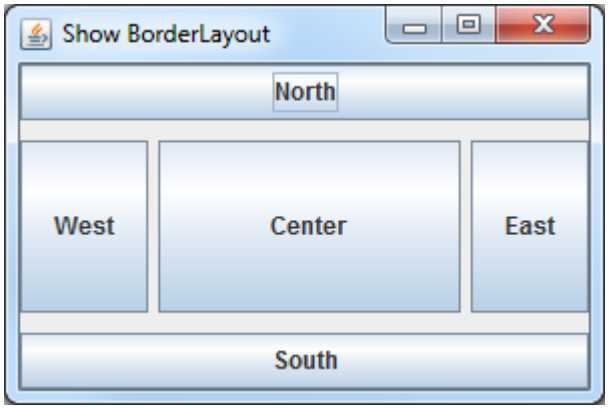
These areas are also called (since 1.4):

- PAGE_START - North
- PAGE_END - South
- LINE_START - West
- LINE_END - East
- CENTER - Center

- Components are added to a **BorderLayout** by using the **add** method with a choice of position

```
aFrame.add(component, BorderLayout.SOUTH);
aFrame.add(component, BorderLayout.PAGE_END);
```

Example: BorderLayout




```
// Set BorderLayout with horizontal gap 5 and vertical gap 10
container.setLayout(new BorderLayout(5, 10));

// Add buttons to the frame
container.add(new JButton("East"), BorderLayout.EAST);
container.add(new JButton("South"), BorderLayout.SOUTH);
container.add(new JButton("West"), BorderLayout.WEST);
container.add(new JButton("North"), BorderLayout.NORTH);
container.add(new JButton("Center"), BorderLayout.CENTER);
```

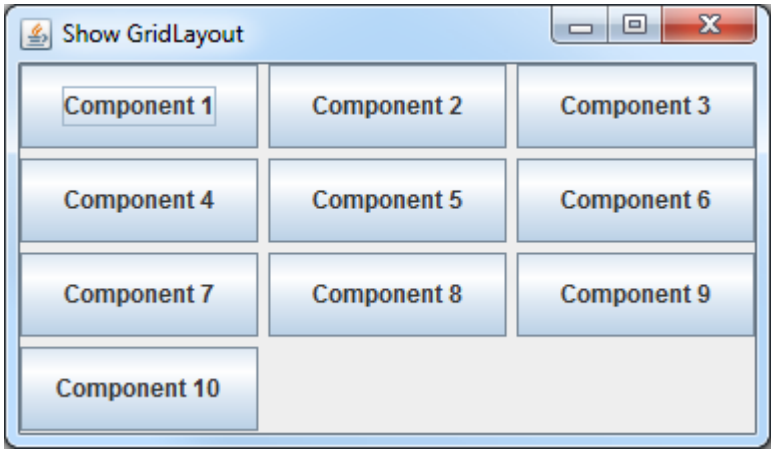
GridLayout Manager

- The GridLayout manager arranges components in a grid (matrix) formation with the number of rows and columns defined by the constructor.
- The components are placed in the grid from left to right starting with the first row, then the second, and so on.

1	2	3
4	5	6
7	8	9

```
new GridLayout(rows, columns);
```

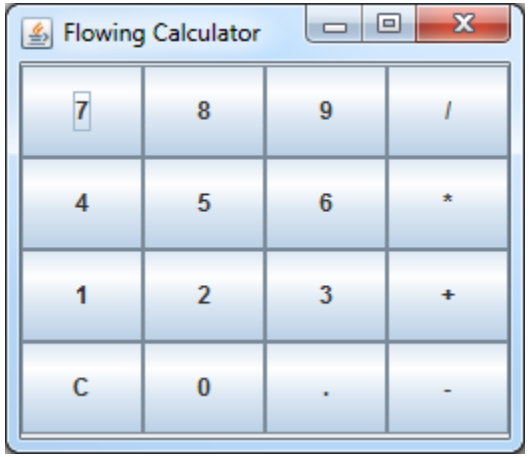
Example: GridLayout



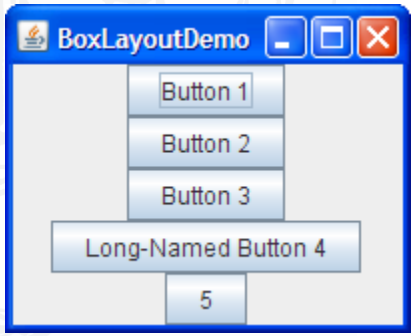

```
// Set GridLayout, 4 rows, 3 columns, and gaps 5 between
// components horizontally and vertically
container.setLayout(new GridLayout(4, 3, 5, 5));

// Add buttons to the frame
for (int i = 1; i <= 10; i++)
  container.add(new JButton("Component " + i));
```

Example: Calculator Buttons on a Grid

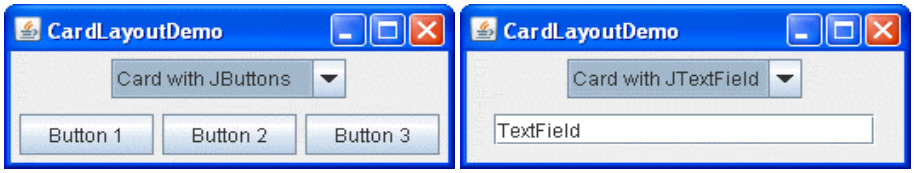
```
public class GridCalculator extends JFrame {
    /** Default constructor */
    public GridCalculator() {

    // Set GridLayout, aligned left with a 4x4 grid
    JPanel calculator = new JPanel(new GridLayout(4,4));
    add(calculator);
    // Add buttons to the panel
    JButton jtn;
    for (int i = 7; i <= 9; i++){
        jtn = new JButton(Integer.toString(i));
        calculator.add(jtn);
    }
    calculator.add(jtn = new JButton("/"));

    for (int i = 4; i <= 6; i++){
        jtn = new JButton(Integer.toString(i));
        calculator.add(jtn);
    }
    calculator.add(jtn = new JButton("*"));

    for (int i = 1; i <= 3; i++){
       jtn = new JButton(Integer.toString(i));
       calculator.add(jtn);
    }
    calculator.add(jtn = new JButton("+"));

    . . .

}
```



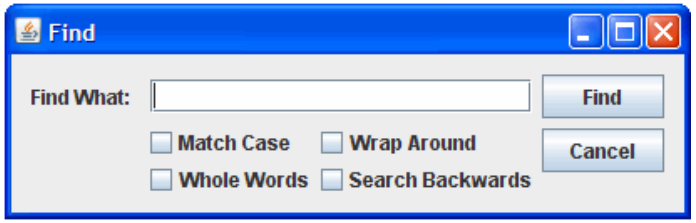
Other Layout Managers



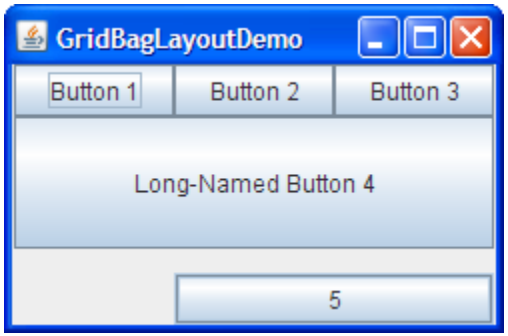
BoxLayout



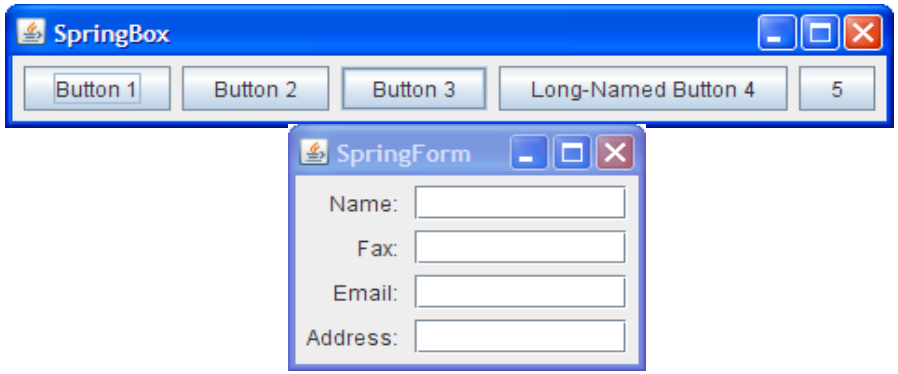
CardLayout



GroupLayout



GridBagLayout



SpringLayout

Null Layout – Absolute Positioning

- Specify the size and position of every component within that container
- It does not adjust well when the top-level container is resized
- It also does not adjust well to differences between users and systems, such as different font sizes and locales

- Steps:
  - Set the container's layout manager to **`null`** by calling

    ```
    setLayout(null)
    ```

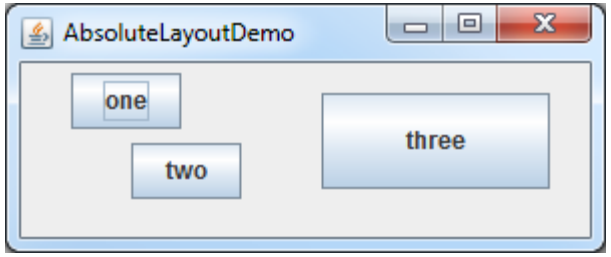
  - Add the component to the container
  - Call the **`Component`** class's **`setBounds`** method for each of the container's children to specify the position and size

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

Although it is possible to do without a layout manager, you should use a layout manager if at all possible.

Example: Absolute Layout

```
public static void addComponentsToPane(Container pane) {

        pane.setLayout(null);

        JButton b1 = new JButton("one");
        JButton b2 = new JButton("two");
        JButton b3 = new JButton("three");

        pane.add(b1);
        pane.add(b2);
        pane.add(b3);

        // The insets of the container indicates
        //     the size of the container's border
        Insets insets = pane.getInsets();
        Dimension size = b1.getPreferredSize();
        b1.setBounds(25 + insets.left, 5 + insets.top,
                     size.width, size.height);
        size = b2.getPreferredSize();
        b2.setBounds(55 + insets.left, 40 + insets.top,
                     size.width, size.height);
        size = b3.getPreferredSize();
        b3.setBounds(150 + insets.left, 15 + insets.top,
                     size.width + 50, size.height + 20);
}
```

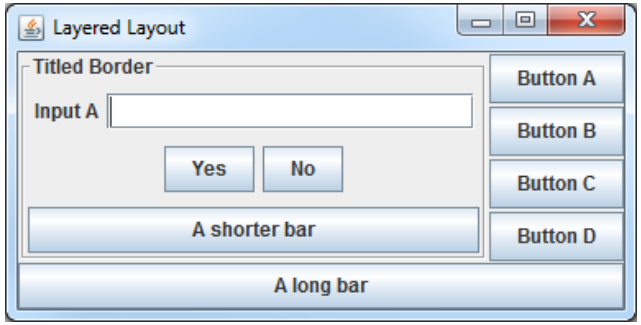


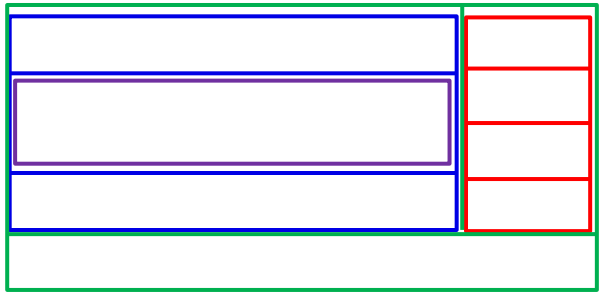
Custom Layout Managers

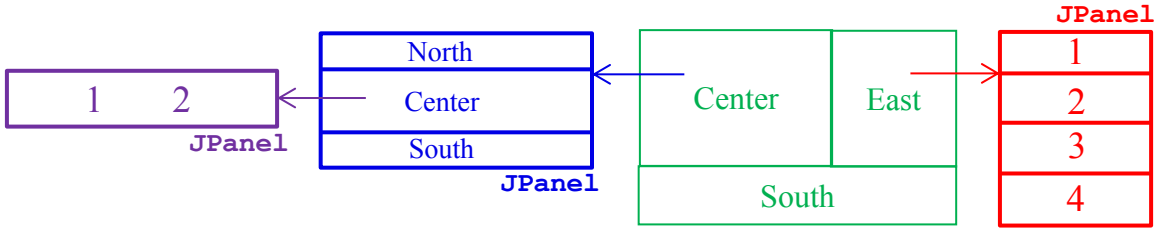
- You can create your own layout manager by implementing the **`LayoutManager`** or **`LayoutManager2`** interface
  - There are 5 or more methods to implement
  - Not easy
- Make sure there is no suitable layout manager before you start creating your own
  - In many cases, the flexible and powerful **`GridBagLayout`** may be useful
- You can try to find layout managers from the Internet
- You can use the **`GroupLayout`** layout manager combined with a builder tool to lay out your GUI
  - If you do not want to write layout code by hand

Using `JPanel` to Compose Complex Layout

- JPanel is a general-purpose container for lightweight components
- JPanel is opaque by default that can be used as a ***content pane***
- Most GUI interfaces use layers of panes
  - Nested panes, with different layout managers, are often used to group components to simplify layout composition
- Borders can be added to JPanel to visually group components

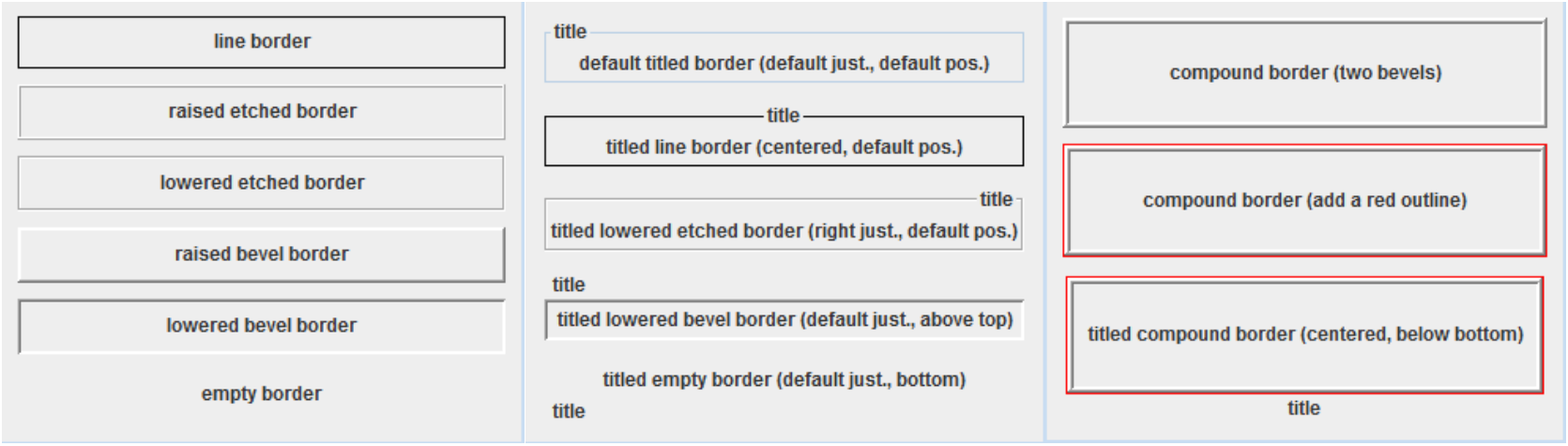




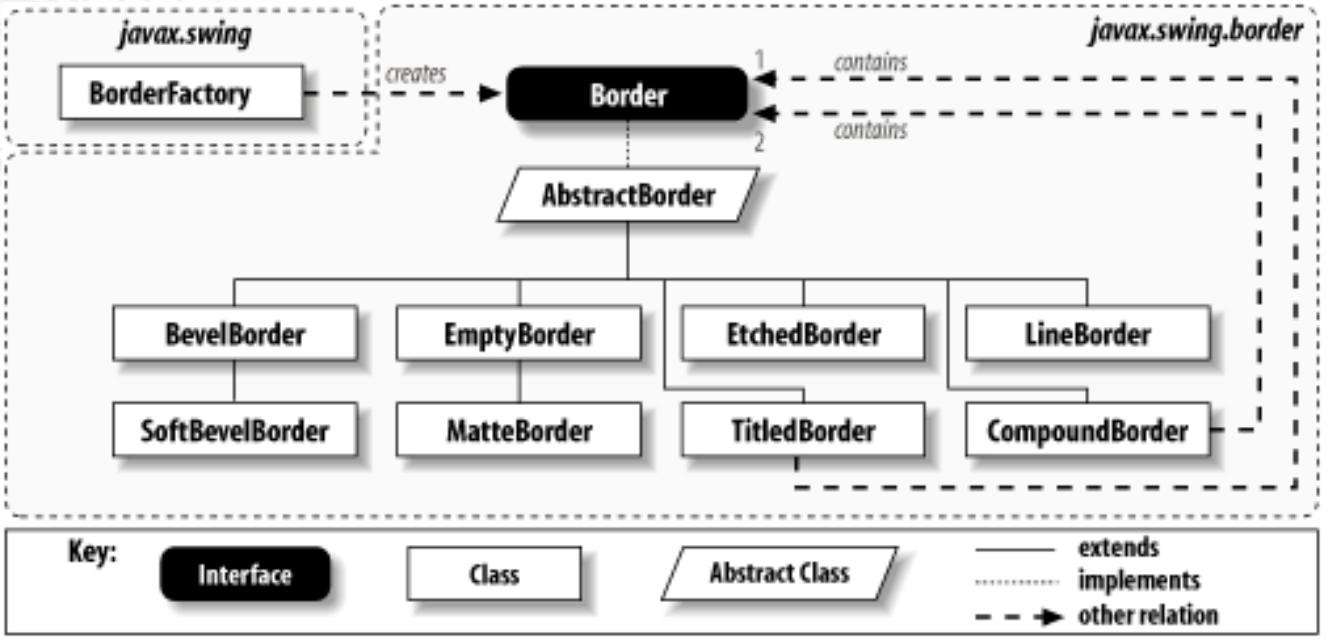


Swing Borders

- Every **`JComponent`** have one or more borders
- Borders are useful not only for decorating the GUI, but also for visually grouping a set of related GUI components
- Borders are useful not only for drawing lines and fancy edges, but also for providing titles and empty space around components



Hierarchy of `Border` Classses



Swing Borders

- Create borders
  - Using **`BorderFactory`** class
    - The factory method may return a readily available border shared with other components for efficiency so that the factory method is preferred, considering that there is few object states to store in individual objects

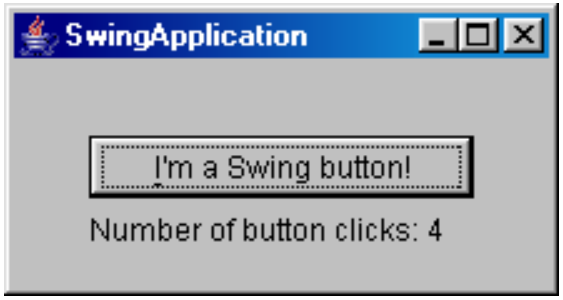
```
// Creating a border; Xxx stands for a border name
Border  aBorder = BorderFactory.createXxxBorder();
// Creating a titled border form a border
Border titledBorder = BorderFactory.createTitledBorder(aBorder);
// Setting the border of a component
component.setBorder(titledBorder);


// Creating a compound border
Border  aBorder = BorderFactory.createCompoundBorder(border1, border2);
```

  - Using constructor
    - Each **`new`** operation (constructor) will create a new object even though an identical one exists

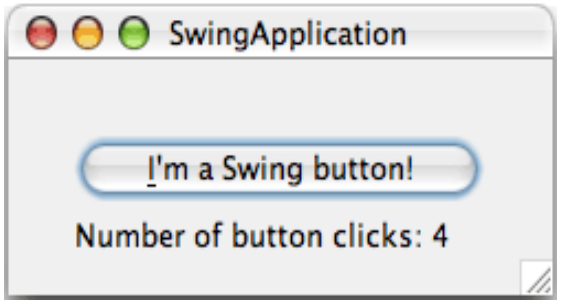
Look and Feel

Java look and feel

GTK+ look and feel



Windows look and feel



Mac OS look and feel

Look –- the appearance of GUI widgets (`JComponents`)
Feel –- the way the widgets behave

Setting Look and Feel

- Programmatically setting the Look and Feel

```
try {
   UIManager.setLookAndFeel(
      UIManager.getCrossPlatformLookAndFeelClassName());
} catch (Exception e) { }
```

- Specifying the Look and Feel: Command Line

```
java -Dswing.defaultlaf=com.sun.java.swing.plaf.windows.WindowsLookAndFeel MyApp
```

- Specifying the Look and Feel: swing.properties
  - located in the lib directory of the Java release

```
# Swing properties
swing.defaultlaf = com.sun.java.swing.plaf.windows.WindowsLookAndFeel
```

- Changing the Look and Feel after Startup

```
UIManager.setLookAndFeel(lnfName);
SwingUtilities.updateComponentTreeUI(frame);
frame.pack();
```

Example: Changing Look-and-Feel

```
/* get installed look-and-feel information */
String[] looks = UIManager.getInstalledLookAndFeels();


/* Change look-and-feel to one of the installed */
private void changeTheLookAndFeel( int value )
   {
      // change look and feel
      try {
         UIManager.setLookAndFeel( looks[value].getClassName());
         SwingUtilities.updateComponentTreeUI( this );
      }

      // process problems changing look and feel
      catch ( Exception exception ) {
         exception.printStackTrace();
      }
   }
```