Layout Managers

CSCI 201L

Jeffrey Miller, Ph.D.

jeffrey.miller@usc.edu

HTTP://WWW-SCF.USC.EDU/~CSCI201
Outline

- Layout Managers
- Custom Layout Managers
- Program
Layout Managers Overview

- Instead of solely using hard-coded values for placing components in a container, Java has layout managers
  - Layout managers provide a level of abstraction so the user interface behaves and looks the same on all systems.
- Layout managers determine the location at which components will be placed.
- To set a layout manager on a container, use the `setLayout(LayoutManager)` method.
LayoutManager Classes

- `java.awt.LayoutManager` is an interface
- There are a number of classes that implement the `LayoutManager` interface
  - `BasicComboBoxUI.ComboBoxLayoutManager`, `BasicInternalFrameTitlePane.TitlePaneLayout`, `BasicInternalFrameUI.InternalFrameLayout`, `BasicOptionPaneUI.ButtonAreaLayout`, `BasicScrollBarUI`, `BasicSplitPaneDivider.DividerLayout`, `BasicSplitPaneUI.BasicHorizontalLayoutManager`, `BasicSplitPaneUI.BasicVerticalLayoutManager`, `BasicTabbedPaneUI.TabbedPaneLayout`, `BorderLayout`, `BoxLayout`, `CardLayout`, `DefaultMenuLayout`, `FlowLayout`, `GridBagLayout`, `GridLayout`, `GroupLayout`, `JRootPane.RootLayout`, `JSpinner.DateEditor`, `JSpinner.DefaultEditor`, `JSpinner.ListEditor`, `JSpinner.NumberEditor`, `MetalComboBoxUI.MetalComboBoxLayoutManager`, `MetalScrollBarUI`, `MetalTabbedPaneUI.TabbedPaneLayout`, `OverlayLayout`, `ScrollPaneLayout`, `ScrollPaneLayout.UIResource`, `SpringLayout`, `SynthScrollBarUI`, `ViewportLayout`

- We will cover a few of the layout managers in this lecture, though other layout managers are used for other applications
When you add a component, you can specify you want to add it to the `BorderLayout.NORTH`, `BorderLayout.SOUTH`, `BorderLayout.EAST`, `BorderLayout.WEST`, or `BorderLayout.CENTER`

- Components in the north and south will have their height acknowledged but the width will be expanded to the width of the frame.
- Components in the east and west will have their width acknowledged but the height will be expanded to the height of the frame.
- Components in the center will have their heights and widths expanded to take up whatever space is remaining in the frame after the other four regions are rendered.

The default layout manager for a `JFrame` is `BorderLayout`
import java.awt.BorderLayout;
import javax.swing.JButton;
import javax.swing.JFrame;

public class Test extends JFrame {
    public Test() {
        super("My Frame");
        setSize(300, 400);
        setLocation(200, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JButton northButton = new JButton("North");
        add(northButton, BorderLayout.NORTH);
        JButton southButton = new JButton("South");
        add(southButton, BorderLayout.SOUTH);
        JButton eastButton = new JButton("East");
        add(eastButton, BorderLayout.EAST);
        JButton westButton = new JButton("West");
        add(westButton, BorderLayout.WEST);
        JButton centerButton = new JButton("Center");
        add(centerButton, BorderLayout.CENTER);

        setVisible(true);
    }

    public static void main(String [] args) {
        Test t = new Test();
    }
}
FlowLayout

- When you add a component, the components will be added from left to right, top to bottom
  - Both the preferred height and width of each component will be acknowledged
- If the components are bigger than the size of the container, they will still exist but will be displayed off the screen until the container is resized
  - Scrollbars do not automatically appear in containers but they can be added
import java.awt.FlowLayout;
import javax.swing.JButton;
import javax.swing.JFrame;

public class Test extends JFrame {
  public Test() {
    super("CSCI 201 Window");
    setSize(200, 220);
    setLocation(200, 200);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setLayout(new FlowLayout());
    JButton button1 = new JButton("1");
    add(button1);
    JButton button2 = new JButton("2");
    add(button2);
    JButton button3 = new JButton("3");
    add(button3);
    JButton button4 = new JButton("4");
    add(button4);
    JButton button5 = new JButton("5");
    add(button5);
    setVisible(true);
  }

  public static void main(String[] args) {
    Test t = new Test();
  }
}
CardLayout

- If multiple containers exist in the application but only one is to be displayed at a time, you could use **CardLayout**
- The programmer can choose which “card” is displayed
  - Think of this layout manager like a stack of playing cards where all of the cards are there, but only one can be seen at a time
```java
import java.awt.CardLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Test extends JFrame {

    public Test() {
        super("CSCI 201 Window");
        setSize(200, 220);
        setLocation(200, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JPanel outerPanel = new JPanel();
        outerPanel.setLayout(new CardLayout());

        JPanel firstPanel = new JPanel();
        JButton button1 = new JButton("1");
        button1.addActionListener(new ButtonClicked("second", outerPanel));
        firstPanel.add(button1);

        JPanel secondPanel = new JPanel();
        JButton button2 = new JButton("2");
        button2.addActionListener(new ButtonClicked("first", outerPanel));
        secondPanel.add(button2);

        outerPanel.add(firstPanel, "first");
        outerPanel.add(secondPanel, "second");
        add(outerPanel);
        setVisible(true);
    }

    public static void main(String[] args) {
        Test t = new Test();
    }
}

class ButtonClicked implements ActionListener {
    private String numberString;
    private JPanel jp;
    public ButtonClicked(String numberString, JPanel jp) {
        this.numberString = numberString;
        this.jp = jp;
    }
    public void actionPerformed(ActionEvent ae) {
        CardLayout cl = (CardLayout)jp.getLayout();
        cl.show(jp, numberString);
    }
}
```

The code above demonstrates the use of the `CardLayout` in a Java Swing application. It creates two panels, `firstPanel` and `secondPanel`, and adds buttons to these panels. The `ButtonClicked` class is used to handle the action listeners for these buttons. When a button is clicked, the `CardLayout` is used to switch between the panels.

When button 1 is clicked, the application displays the first panel. When button 2 is clicked, the application displays the second panel.
JTabbedPane

- **JTabbedPane** is not a layout manager, but it works similarly to the **CardLayout**
- Instead of having the other containers completely hidden, a **JTabbedPane** allows most of the container to be hidden except a small tab at the top
  - The tab can have text on it
JTabbedPane Example

```java
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTabbedPane;

public class Test extends JFrame {

    public Test() {
        super("CSCI 201 Window");
        setSize(200, 220);
        setLocation(200, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JTabbedPane tabbedPane = new JTabbedPane();

        JPanel firstPanel = new JPanel();
        JButton button1 = new JButton("1");
        firstPanel.add(button1);

        JPanel secondPanel = new JPanel();
        JButton button2 = new JButton("2");
        secondPanel.add(button2);

        tabbedPane.add("First", firstPanel);
        tabbedPane.add("Second", secondPanel);

        add(tabbedPane);
        setVisible(true);
    }

    public static void main(String[] args) {
        Test t = new Test();
    }
}
```

Layout Managers
GridLayout

- **GridLayout** allows you to have your container emulate a grid, with columns and rows
  - The components are placed into each cell in the grid
- The **GridLayout** will take up the entire location in the container it occupies
  - Each cell will be the same size, which will possibly not allow either the height or width to be acknowledged
GridLayout Example

```java
import java.awt.GridLayout;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Test extends JFrame {

    public Test() {
        super("CSCI 201 Window");
        setSize(200, 220);
        setLocation(200, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JPanel jp = new JPanel();
        jp.setLayout(new GridLayout(3, 3));

        for (int i=1; i < 10; i++) {
            JButton button = new JButton("" + i);
            jp.add(button);
        }

        add(jp);
        setVisible(true);
    }

    public static void main(String[] args) {
        Test t = new Test();
    }
}
```
The GridBagLayout is one of the most complex and flexible layout managers. There is a grid of rows and columns, but the heights and widths do not have to be the same. Components can take up multiple cells as well.

GridBagLayout tries to acknowledge the preferred height and width of the components.

Each component gets added to the grid with a GridBagConstraints object to set the parameters for the component.

More information can be found at http://docs.oracle.com/javase/tutorial/uiswing/layout/gridbag.html
GridBagConstraints Object

The GridBagConstraints object has the following properties:

- **gridx, gridy**
  - Specify the row and column at the upper left corner of the component

- **gridwidth, gridheight**
  - Specify the number of rows or columns in the component’s display area

- **fill**
  - Used when the component’s display area is larger than the component’s preferred size
  - Values can be NONE, VERTICAL, HORIZONTAL, or BOTH

- **ipadx, ipady**
  - Specifies the internal padding of the component

- **insets**
  - Specifies the external padding of the component (through Insets object)

- **anchor**
  - Used when the component is smaller than its display area
  - Values an be CENTER (the default), PAGE_START, PAGE_END, LINE_START, LINE_END, FIRST_LINE_START, FIRST_LINE_END, LAST_LINE_END, and LAST_LINE_START

- **weightx, weighty**
  - Weights are used to determine how to distribute space among columns and rows, specified between 0.0 and 1.0
GridBagLayout Example

```java
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Test extends JFrame {
    public Test() {
        super("CSCI 201 Window");
        setSize(200, 220);
        setLocation(200, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JPanel jp = new JPanel();
        jp.setLayout(new GridBagLayout());

        GridBagConstraints gbc = new GridBagConstraints();
        JButton button1 = new JButton("Button 1");
        gbc.gridx = 0;
        gbc.gridy = 0;
        jp.add(button1, gbc);

        JButton button2 = new JButton("Button 2");
        gbc.gridx = 1;
        gbc.gridy = 0;
        gbc.ipadx = 50;
        jp.add(button2, gbc);

        JButton button3 = new JButton("Button 3");
        gbc.gridx = 0;
        gbc.gridy = 1;
        gbc.gridwidth = 2;
        gbc.ipadx = 0;
        gbc.ipady = 40;
        jp.add(button3, gbc);

        add(jp);
        setVisible(true);
    }

    public static void main(String[] args) {
        Test t = new Test();
    }
}
```
BoxLayout

- **BoxLayout** either stacks all of the components on top of each other or puts them next to each other
  - It works similarly to **FlowLayout** but with more functionality
- **BoxLayout** tries to acknowledge the preferred height and width of the components
  - If no preferred height or width is set, the component can expand based on the arrangement of components
- Rigid area can be specified to allow a fixed-space between components
- Glue can be used to specify where excess space in a layout should go
  - Stretches and compresses based on the size of the container
BoxLayout Example

```java
import javax.swing.BoxLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Test extends JFrame {
    public Test() {
        super("CSCI 201 Window");
        setSize(200, 220);
        setLocation(200, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JPanel jp = new JPanel();
        jp.setLayout(new BoxLayout(jp, BoxLayout.Y_AXIS));

        JButton button1 = new JButton("Button 1");
        jp.add(button1);
        jp.add(Box.createGlue());
        JButton button2 = new JButton("Button 2");
        jp.add(button2);
        JButton button3 = new JButton("Button 3");
        jp.add(button3);

        add(jp);
        setVisible(true);
    }

    public static void main(String[] args) {
        Test t = new Test();
    }
}
```
**GroupLayout**

- **GroupLayout** separates out formatting the horizontal layer from the vertical layer.
- If we are trying to create the following GUI:
  - The horizontal group will be
    ![Horizontal Group](image1)
  - And the vertical group will be
    ![Vertical Group](image2)
GroupLayout

- The horizontal and vertical groups actually overlap
  - GroupLayout will not allow a component to only be added to one group though – they must be added to both the vertical and horizontal group
- Components or groups can be added to other groups so we are not limited by only one component in each group
- GroupLayout is a fairly complex layout manager, but it is also very flexible
import javax.swing.GroupLayout;
import javax.swing.GroupLayout.ParallelGroup;
import javax.swing.GroupLayout.SequentialGroup;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;

public class Test extends JFrame {
  public Test() {
    super("Find");
    setSize(500, 130);
    setLocation(300, 300);
    JLabel label = new JLabel("Find What:");
    JTextField textField = new JTextField();
    JCheckBox caseCheckBox = new JCheckBox("Match Case");
    JCheckBox wrapCheckBox = new JCheckBox("Wrap Around");
    JCheckBox wholeCheckBox = new JCheckBox("Whole Words");
    JCheckBox backCheckBox = new JCheckBox("Search Backwards");
    JButton findButton = new JButton("Find");
    JButton cancelButton = new JButton("Cancel");

    GroupLayout layout = new GroupLayout(getContentPane());
    getContentPane().setLayout(layout);
    layout.setAutoCreateGaps(true);
    layout.setAutoCreateContainerGaps(true);

    SequentialGroup horizontal = layout.createSequentialGroup();
    horizontal.addComponent(label);
    ParallelGroup searchBoxGroup = layout.createParallelGroup();
    searchBoxGroup.addComponent(textField);
    horizontal.addGroup(searchBoxGroup);
    SequentialGroup checkBoxGroup = layout.createSequentialGroup();
    ParallelGroup firstCheckbox = layout.createParallelGroup();
    firstCheckbox.addComponent(caseCheckBox);
    firstCheckbox.addComponent(wholeCheckBox);
    ParallelGroup secondCheckbox = layout.createParallelGroup();
    secondCheckbox.addComponent(wrapCheckBox);
    secondCheckbox.addComponent(backCheckBox);
    checkBoxGroup.addGroup(firstCheckbox);
    checkBoxGroup.addGroup(secondCheckbox);
    searchBoxGroup.addGroup(checkBoxGroup);

    ParallelGroup buttonGroup = layout.createParallelGroup();
    buttonGroup.addComponent(findButton);
    buttonGroup.addComponent(cancelButton);
    horizontal.addGroup(buttonGroup);
    layout.setHorizontalGroup(sequential);

    SequentialGroup vertical = layout.createSequentialGroup();
    ParallelGroup topRow = layout.createParallelGroup();
    topRow.addComponent(label);
    topRow.addComponent(textField);
    topRow.addGroup(findButton);
    vertical.addGroup(topRow);

    ParallelGroup bottomRow = layout.createParallelGroup();
    SequentialGroup buttonGroup1 = layout.createSequentialGroup();
    ParallelGroup firstButtonGroup = layout.createParallelGroup();
    firstButtonGroup.addComponent(caseCheckBox);
    firstButtonGroup.addComponent(wrapCheckBox);
    firstButtonGroup.addGroup(firstCheckbox);
    secondButtonGroup.addGroup(secondCheckbox);
    bottomRow.addGroup(secondButtonGroup);
    bottomRow.addGroup(cancelButton);
    vertical.addGroup(bottomRow);

    layout.setVerticalGroup(vertical);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setVisible(true);
  }

  public static void main(String args[]) {
    Test t = new Test();
  }
}
null Layout

- If you set the layout manager to null, you are able to place components in the container with absolute positioning.
- If the container is resized, your components may not still appear in the same location (or may not be seen).
  - You are able to restrict a container from being resized.
- You may want to set the bounds and size on your components so they appear how you want.
null Layout Example

```java
import java.awt.Dimension;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class Test extends JFrame {
    public Test() {
        super("CSCI 201 Window");
        setSize(300, 200);
        setLocation(300, 300);

        JPanel jp = new JPanel();
        jp.setLayout(null);

        JButton button1 = new JButton("One");
        Dimension button1Dimensions = button1.getPreferredSize();
        button1.setBounds(25, 25, button1Dimensions.width, button1Dimensions.height);
        jp.add(button1);

        JButton button2 = new JButton("Two");
        Dimension button2Dimensions = button2.getPreferredSize();
        button2.setBounds(75, 75, button2Dimensions.width, button2Dimensions.height);
        jp.add(button2);

        JButton button3 = new JButton("Three");
        Dimension button3Dimensions = button3.getPreferredSize();
        button3.setBounds(125, 125, button3Dimensions.width, button3Dimensions.height);
        jp.add(button3);

        add(jp);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }

    public static void main(String args[]) {
        Test t = new Test();
    }
}
```
Outline

- Layout Managers
- Custom Layout Managers
- Program
To create your own layout manager, create a class that implements the `LayoutManager` interface.

You will need to implement the following methods:

- **void addLayoutComponent(String, Component)**: Called by the Container class's `add` methods. Layout managers that do not associate strings with their components generally do nothing in this method.
- **void removeLayoutComponent(Component)**: Called by the Container methods `remove` and `removeAll`. Layout managers override this method to clear an internal state they may have associated with the Component.
- **Dimension preferredLayoutSize(Container)**: Called by the Container class's `getPreferredSize` method, which is itself called under a variety of circumstances. This method should calculate and return the ideal size of the container, assuming that the components it contains will be at or above their preferred sizes. This method must take into account the container's internal borders, which are returned by the `getInsets` method.
- **Dimension minimumLayoutSize(Container)**: Called by the Container `getMinimumSize` method, which is itself called under a variety of circumstances. This method should calculate and return the minimum size of the container, assuming that the components it contains will be at or above their minimum sizes. This method must take into account the container's internal borders, which are returned by the `getInsets` method.
- **void layoutContainer(Container)**: Called to position and size each of the components in the container. A layout manager's `layoutContainer` method does not actually draw components. It simply invokes one or more of each component's setSize, setLocation, and setBounds methods to set the component's size and position.
Outline

- Layout Managers
- Custom Layout Managers
- Program
Programs

- Write a program that produces the GUI below with each tab displaying a different layout manager.
Programs

- Write a program that produces the GUI below without using GroupLayout.

- Write a program that produces the GUI below.