Object Oriented Programming and Design in Java

Session 9
Instructor: Bert Huang
Announcements

• Homework 2 due Mar. 3rd, 11 AM
• one week to go
• Midterm review Monday, Mar. 8th
• Midterm exam Wednesday, Mar. 10th
Review

• More Swing components
  • JTextArea, JSplitPane
• Listeners in Swing
  • Change listener
  • Focus listener
• Mouse listeners
Today's Plan

• More LayoutManager examples
  • BorderLayout, BoxLayout, GridLayout
• Discussion of Inheritance
Layout Managers

- LayoutManager is an interface in AWT
- Container objects call methods to add components and lay them out
- Responsibilities:
  - Calculate the minimum and preferred size of the Container
  - Lay out the Container's children
LayoutManager

Methods

- addLayoutComponent(String name, Component comp)
- layoutContainer(Container parent)
- Dimension minimumLayoutSize(Container parent)
- Dimension preferredLayoutSize(Container parent)
- removeLayoutComponent(Component comp)
BorderLayout

- Doesn't fit the Strategy pattern
- You specify where you add components
- container.add(Component, LOCATION)
  - BorderLayout.PAGE_START (NORTH)
  - BorderLayout.LINE_START (WEST)
  - BorderLayout.CENTER
  - BorderLayout.LINE_END (EAST)
  - BorderLayout.PAGE_END (SOUTH)
```java
JButton button = new JButton("Button 1 (PAGE_START)" );
pane.add(button, BorderLayout.PAGE_START);

//Make the center component big, since
//that's the typical usage of BorderLayout.
button = new JButton("Button 2 (CENTER)" );
button.setPreferredSize(new Dimension(200, 100));
pane.add(button, BorderLayout.CENTER);

button = new JButton("Button 3 (LINE_START)" );
pane.add(button, BorderLayout.LINE_START);

button = new JButton("Long-Named Button 4 (PAGE_END)" );
pane.add(button, BorderLayout.PAGE_END);

button = new JButton("5 (LINE_END)" );
pane.add(button, BorderLayout.LINE_END);
```
BoxLayout

- Left-to-right or top-to-bottom
- Obeys alignment field of container

- `JComponent.setAlignmentX(float)` // takes value
  `JComponent.setAlignmentY(float)` // between 0 to 1

- `Component.LEFT_ALIGNMENT`
- `Component.RIGHT_ALIGNMENT`
- `Component.CENTER_ALIGNMENT`
- `Component.BOTTOM_ALIGNMENT`
- `Component.TOP_ALIGNMENT`
BoxLayout

- Unusual constructor:
  new BoxLayout(Container, int axis)

- BoxLayout.X_AXIS, BoxLayout.Y_AXIS

- BoxLayout tries to grow components to fill the space, subject to maximum size
public static void addComponentsToPane(Container pane) {
    pane.setLayout(new BoxLayout(pane, BoxLayout.Y_AXIS));

    addAButton("Button 1", pane);
    addAButton("Button 2", pane);
    addAButton("Button 3", pane);
    addAButton("Long-Named Button 4", pane);
    addAButton("5", pane);
}

private static void addAButton(String text, Container container) {
    JButton button = new JButton(text);
    button.setAlignmentX(Component.CENTER_ALIGNMENT);
    container.add(button);
}
GridLayout

- Lays out components on a grid from left to right in rows from top to bottom
- Grows components to fill available space if container is bigger than preferred size
- You specify grid size in constructor: new GridLayout(int rows, int columns)
- One of rows or columns may be 0, which tells AWT to add as many as needed
GridLayout Example

JFrame gridFrame = new JFrame("GridLayout");
gridFrame.setLayout(new GridLayout(2,3));

for (int i=0; i<6; i++)
    gridFrame.add(new JButton("Component "+i));
Inheritance

• Describes a relationship between classes in which a \textit{subclass} is a more specific form of a \textit{superclass}

• Declared in Java with the keyword \texttt{extends}
Why Extend Classes?

- Inheritance may happen naturally
  - AWT's Component first introduced in 1995
  - Swing's JComponent in 1997
- Or it can be by design:
  - we know we want to use fully functioning objects of a general superclass
  - but we also want more specific functionality of some subclasses
Subclasses

• Subclasses often provide additional methods and fields
  • or they may *override* the superclass's methods

• Java allows special keyword `super` to refer to superclass
  • used to invoke superclass's methods, including constructor
Keyword super

- We saw this example last class
- MouseAdapter is the superclass

```java
private static class MyMouseListener extends MouseAdapter {
    public MyMouseListener(MousePanel panel) {
        super();
        myPanel = panel;
    }
    public void mouseClicked(MouseEvent event) {
        ...
    }
}
```
Liskov's Substitution Principle

• Let $q(x)$ be a property provable about objects $x$ of type $T$. Then $q(y)$ should be true for objects $y$ of type $S$ where $S$ is a subtype of $T$. (Liskov)

• You can substitute subclass objects whenever a superclass object is expected

• but not always vice versa (never)
EventObject Hierarchy

- EventObject
- ActionEvent
- ChangeEvent
- MouseEvent
Polymorphism and Inheritance

- Overriding methods can cause some confusion if we're unclear on how inheritance works

- We extended MouseAdapter to make MyMouseListener

  ```java
  MouseAdapter ma = new MyMouseListener();
  ma.mouseClicked(); // what happens?
  ```

- Actual types of objects, not declared types, determine which methods are called
Encapsulation and Inheritance

- Public and private modifiers apply even to subclasses
- Extending a class doesn't grant you access to its private methods
- Otherwise, implementations would not be interchangeable, since subclasses would depend on private class code
- Subclasses must implement their added functionality using only public interface of superclass
Preconditions and Postconditions

- Subclass methods cannot have stricter preconditions than superclass methods.
- Subclass methods cannot have looser postconditions than superclass methods.
- Because all subclass objects must fit Liskov substitution; they must be viewable as superclass objects.
Reading

• Layout examples from: http://java.sun.com/docs/books/tutorial/uiswing/layout/index.html

• Horstmann Ch. 6