CS 142

Lecture 26: event driven programming; layout

Thanks to Marty Stepp and Stuart Reges for parts of these slides

Event-driven Programming

- program's execution is indeterminate
- on-screen components cause *events* to occur when they are clicked / interacted with
- events can be handled, causing the program to respond, *driving* the execution thru events (an "event-driven" program)

Action events: `ActionEvent`

- most common / simple event type in Swing
- represent an action occurring on a GUI component
- created by:
  - button clicks
  - check box checking / unchecking
  - menu clicks
  - pressing Enter in a text field
  - etc.

Listening for Events

- attach *listener* to component
- listener's appropriate method will be called when event occurs (e.g. when the button is clicked)
- for Action events, use `ActionListener`

Writing an `ActionListener`

```
// part of Java; you don't write this
public interface ActionListener {
    public void actionPerformed(ActionEvent event);
}

// Prints a message when the button is clicked.
public class MyActionListener
        implements ActionListener {

    public void actionPerformed(ActionEvent event){
        JOptionPane.showMessageDialog(null,
                "An event occurred!");
    }
}
```

Attaching an `ActionListener`

```
JButton button = new JButton("button 1");
ActionListener listener = new MyActionListener();
button.addActionListener(listener);
```

- now `button` will print `"Event occurred!"` when clicked
  - `addActionListener` method exists in many Swing components

Layout

Sizing and positioning

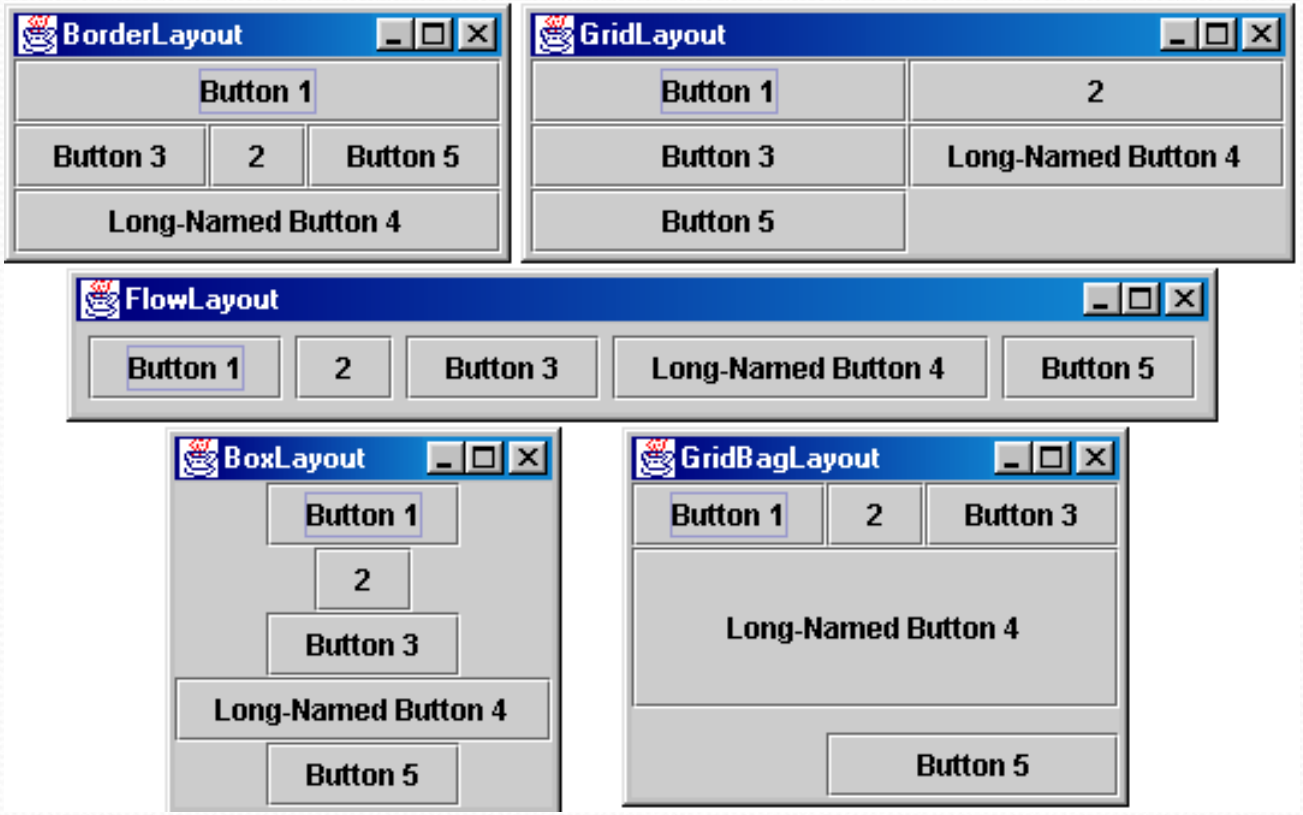
How does the programmer specify where each component appears, how big each component should be, and what the component should do if the window is resized / moved / maximized / etc.?

- **Absolute positioning** (C++, C#, VB, others):
  Programmer specifies exact pixel coordinates of every component.
  - "Put this button at (x=15, y=75) and make it 70x31 px in size."

- **Layout managers** (Java):
  Objects that decide where to position each component based on some general rules or criteria.
  - "Put these four buttons into a 2x2 grid and put these text boxes in a horizontal flow in the south part of the frame."

Containers and layout

- Place components in a *container*; add the container to a frame.
  - **container**: An object that stores components and governs their positions, sizes, and resizing behavior.



JFrame as container

A `JFrame` is a container. Containers have these methods:

- `public void` **`add`**`(Component comp)`
  `public void` **`add`**`(Component comp, Object info)`
  Adds a component to the container, possibly giving extra information about where to place it.

- `public void` **`remove`**`(Component comp)`

- `public void` **`setLayout`**`(LayoutManager mgr)`
  Uses the given layout manager to position components.

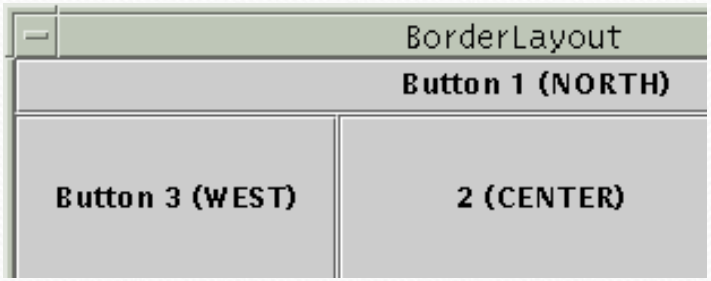
- `public void` **`validate`**`()`
  Refreshes the layout (if it changes after the container is onscreen).

Preferred sizes

- Swing component objects each have a certain size they would "like" to be: Just large enough to fit their contents (text, icons, etc.).
  - This is called the *preferred size* of the component.
  - Some types of layout managers (e.g. `FlowLayout`) choose to size the components inside them to the preferred size.
  - Others (e.g. `BorderLayout`, `GridLayout`) disregard the preferred size and use some other scheme to size the components.

Buttons at preferred size:

Not preferred size:



FlowLayout

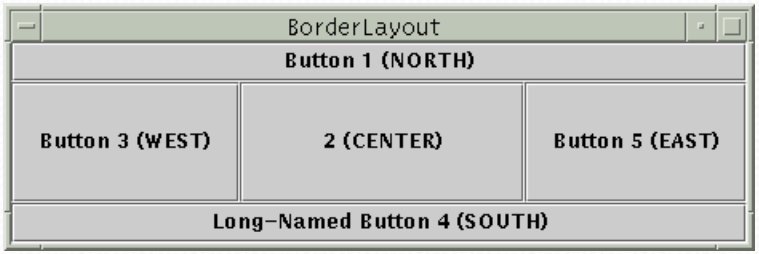
```
public FlowLayout()
```

- treats container as a left-to-right, top-to-bottom "paragraph".
  - Components are given preferred size, horizontally and vertically.
  - Components are positioned in the order added.
  - If too long, components wrap around to the next line.

```
myFrame.setLayout(new FlowLayout());
myFrame.add(new JButton("Button 1"));
```

- The default layout for containers other than `JFrame` (seen later).

BorderLayout




```
public BorderLayout()
```

- Divides container into five regions:
  - `NORTH` and `SOUTH` regions expand to fill region horizontally, and use the component's preferred size vertically.
  - `WEST` and `EAST` regions expand to fill region vertically, and use the component's preferred size horizontally.
  - `CENTER` uses all space not occupied by others.

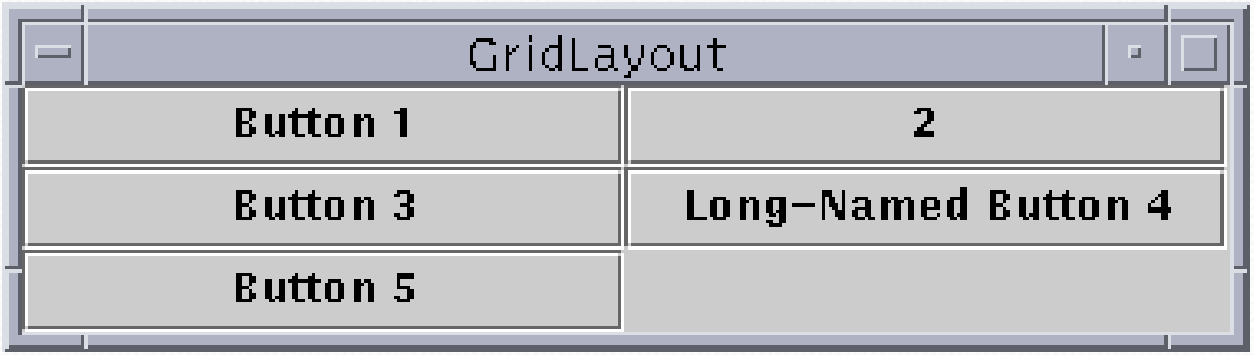
```
myFrame.setLayout(new BorderLayout());
myFrame.add(new JButton("Button 1"),
  BorderLayout.NORTH);
```

  - This is the default layout for a `JFrame`.

GridLayout

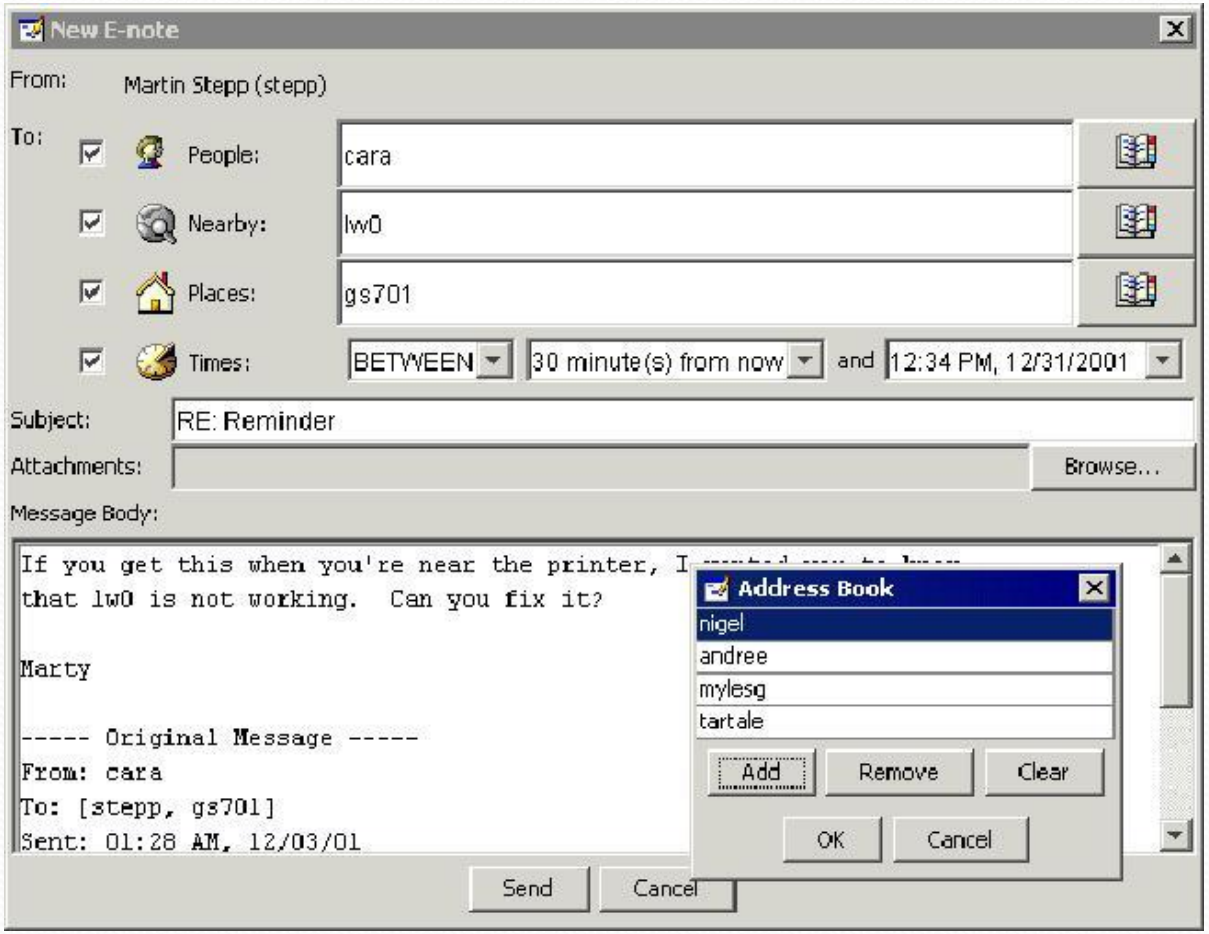
```
public GridLayout(int rows, int columns)
```

- Treats container as a grid of equally-sized rows and columns.
- Components are given equal horizontal / vertical size, disregarding preferred size.
- Can specify 0 rows or columns to indicate expansion in that direction as needed.



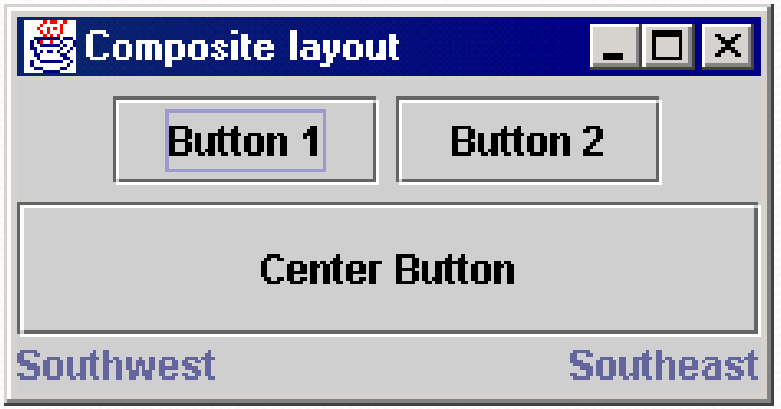
Complex layouts

- How would you create a complex window like this, using the layout managers shown?



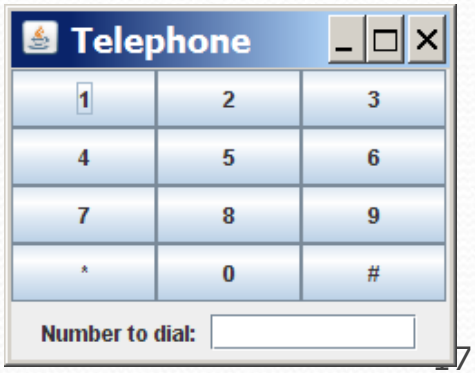
Solution: composite layout

- create panels within panels
- each panel has a different layout, and by combining the layouts, more complex / powerful layout can be achieved
- example:
  - how many panels?
  - what layout in each?



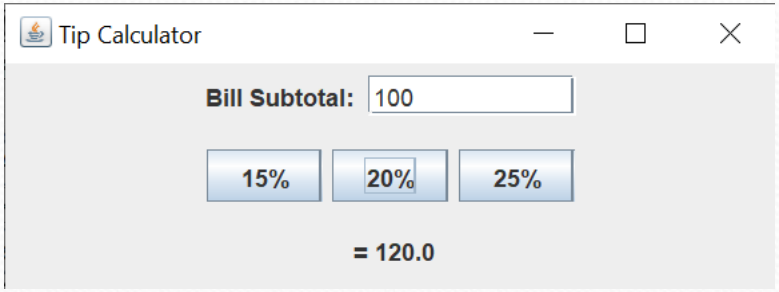
Composite layout example

```
import java.awt.*;
import javax.swing.*;

public class Telephone {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(new Dimension(250, 200));
        frame.setTitle("Telephone");

        frame.setLayout(new BorderLayout());

        JPanel centerPanel = new JPanel(new GridLayout(4, 3));
        for (int i = 1; i <= 9; i++) {
            centerPanel.add(new JButton("" + i));
        }
        centerPanel.add(new JButton("*"));
        centerPanel.add(new JButton("0"));
        centerPanel.add(new JButton("#"));
        frame.add(centerPanel, BorderLayout.CENTER);

        JPanel southPanel = new JPanel(new FlowLayout());
        southPanel.add(new JLabel("Number to dial: "));
        southPanel.add(new JTextField(10));
        frame.add(southPanel, BorderLayout.SOUTH);

        frame.setVisible(true);
    }
}
```



Exercise: Tip Calculator

- Write a program that allows the user to calculate a 15%, 20% or 25% tip by pressing a button.
  - Make the window 400 by 150
  - Program should exit when the window is closed
  - Tip calculation only happens when a button is pressed



JFrame Properties

- JFrames have the following unique properties that you can get or set in your graphical programs:

name	type	description	methods
default close operation	int	what should happen when frame is closed	getDefaultCloseOperation, setDefaultCloseOperation
icon image	Image	icon in the window's title bar	getIconImage, setIconImage
layout	LayoutManager	how the frame should position its components	getLayout, setLayout
resizable	boolean	whether the window can be resized	isResizable, setResizable
title	String	window's title bar text	getTitle, setTitle

Component Properties

- All components also have the following properties:

name	type	description	methods
background	`Color`	background color	`getBackground`, `setBackground`
enabled	`boolean`	whether the component can be interacted with	`isEnabled`, `setEnabled`
font	`Font`	font used to display any text on the component	`getFont`, `setFont`
foreground	`Color`	foreground color	`getForeground`, `setForeground`
location	`Point`	(x, y) position of component on screen	`getLocation`, `setLocation`
size	`Dimension`	width, height of component	`getSize`, `setSize`
preferred size	`Dimension`	width, height that the component wants to be	`getPreferredSize`, `setPreferredSize`
visible	`boolean`	whether the component can be seen on screen	`isVisible`, `setVisible`

JButton and JLabel

The most common component—a button is a clickable onscreen region that the user interacts with to perform a single command

A text label is simply a string of text displayed on screen in a graphical program. Labels often give information or describe other components

- `public JButton(String text)`
  `public JLabel(String text)`
  Creates a new button / label with the given string as its text.

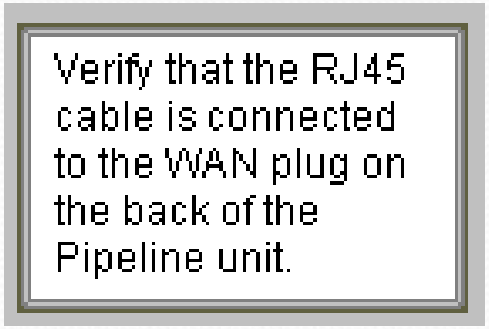
- `public String getText()`
  Returns the text showing on the button / label.

- `public void setText(String text)`
  Sets button / label's text to be the given string.

JTextField and JTextArea

A text field is like a label, except that the text in it can be edited and modified by the user. Text fields are commonly used for user input, where the user types information in the field and the program reads it

A text area is a multi-line text field



- `public JTextField(int columns)`
- `public JTextArea(int lines, int columns)`
  Creates a new text field that is the given number of columns (letters) wide.

- `public String getText()`
  Returns the text currently in the field.

- `public void setText(String text)`
  Sets field's text to be the given string.