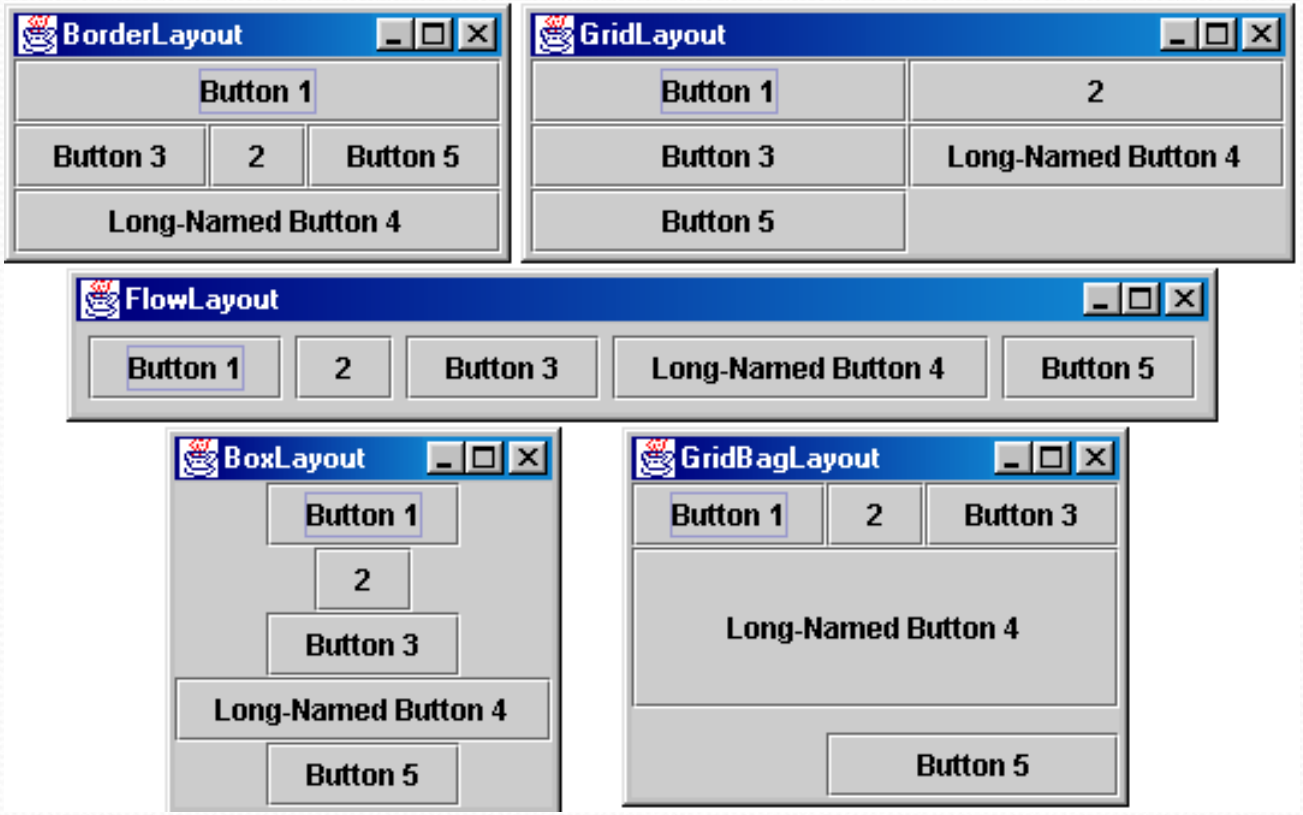
CS 142

Lecture 27: GUIs

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

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.

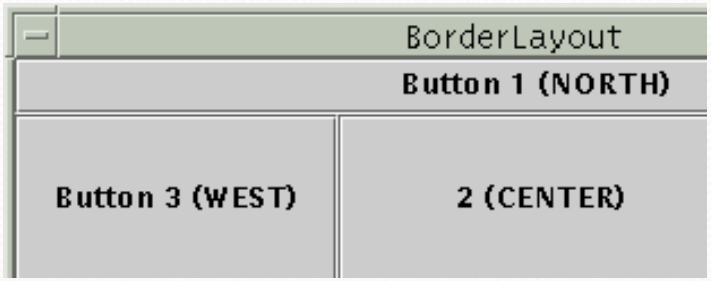
- `public void` **`validate`**`()`
  Refreshes the layout (if it changes after the container is onscreen).

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

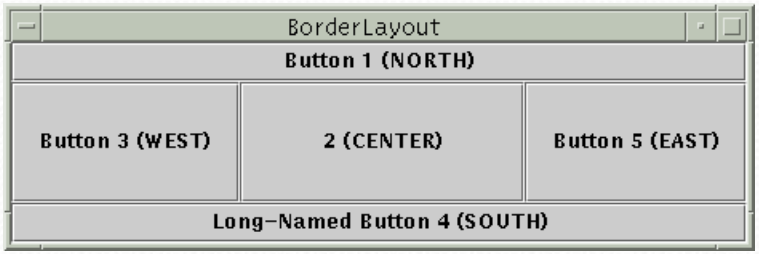
```
public FlowLayout()
```

- 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.

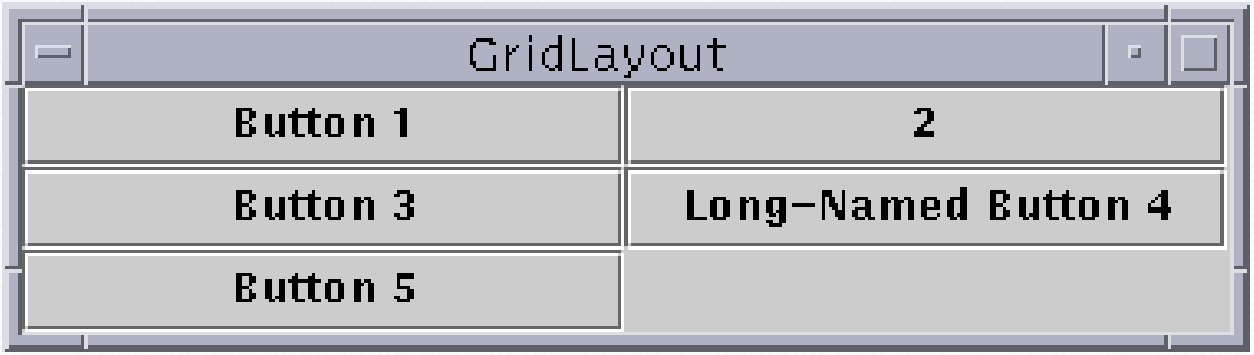
```
myFrame.setLayout(new BorderLayout());
myFrame.add(new JButton("Button 1"),
  BorderLayout.NORTH);
```

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

GridLayout

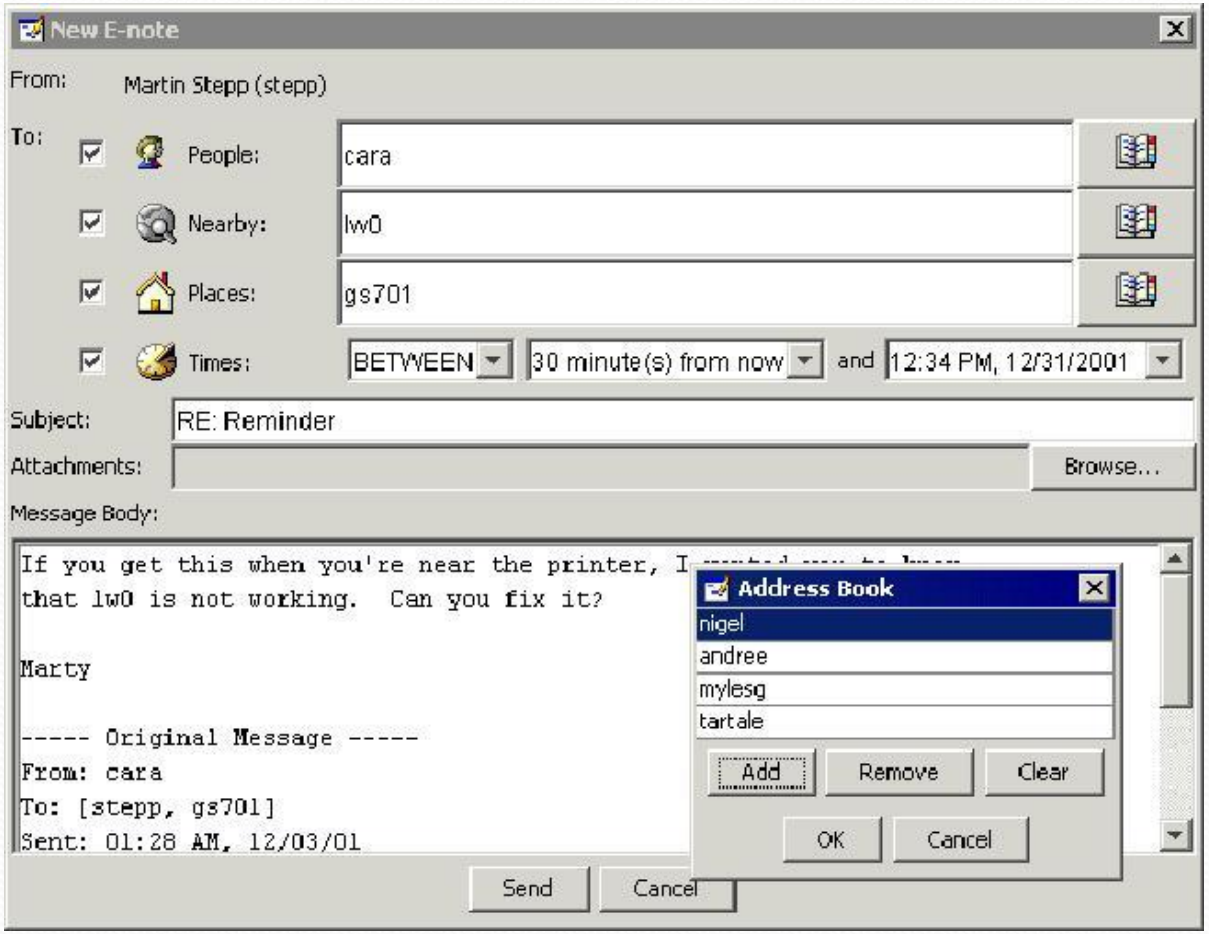
```
public GridLayout(int rows, int columns)
```

- 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.



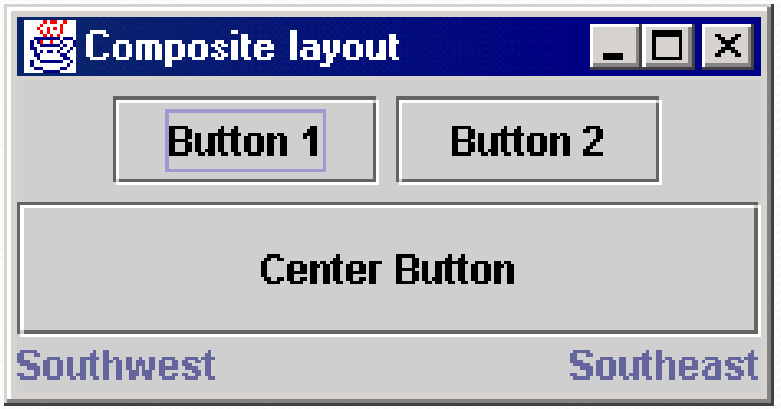
Complex layouts

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



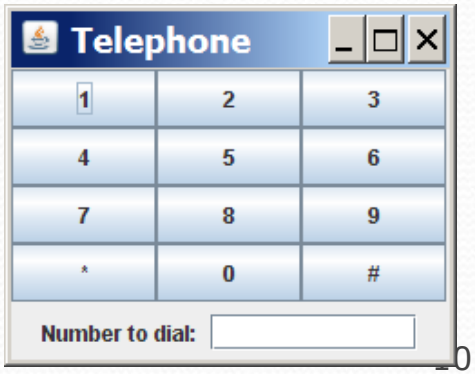
Solution: composite layout

- 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?



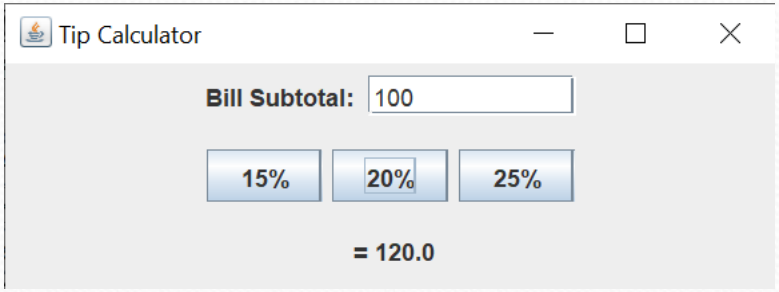
Composite layout example

```
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



Making your own `DrawingPanel`

- To draw shapes on a window we need to add an image
  - `BufferedImage` **`name`** `= new BufferedImage(`**`width, height, colorType`**`)`
  - We will be using color type `BufferedImage.TYPE_INT_RGB`

- An image can't go directly in the frame. It must go inside an `ImageIcon` and the `ImageIcon` must go in a `Label` and then ***finally*** we can put the `Label` on the `Frame`

```
Jframe frame = new JFrame();
BufferedImage img =
      new BufferedImage(400, 500, BufferedImage.TYPE_INT_RGB);
JLabel surround = new JLabel(new ImageIcon(img));
frame.add(surround);
```

Drawing

- Just like with the `DrawingPanel`, we need a pen to draw shapes.
  - You can get a pen from the image the same way you did from the DrawingPanel:

```
Graphics name = yourBufferedImage.getGraphics()
```

Now you can call all the same methods you called on the `Graphics` object you got from `DrawingPanel`.

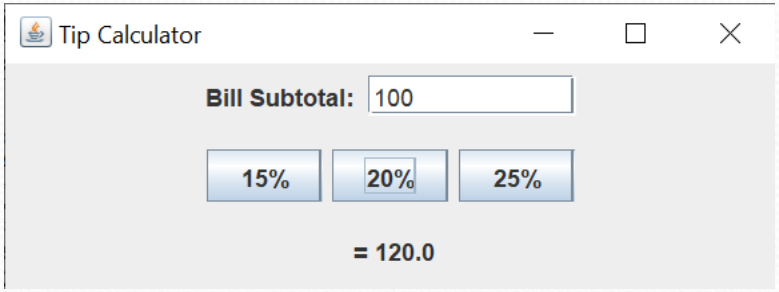
Animation

- We need to make some code to run every ***n*** amount of seconds
  - Use the `Timer` class
  - Create a class that extends `TimerTask` and give it a `run()` method
    - Think of the timer as an alarm clock that only has a snooze button

```
Timer timer = new Timer();          // from java.util
TimerTask task = new MyTimer();
timer.schedule(task, 200, 100); // starts timer to run task
                // after 200ms and then every 100ms


public class MyTimer extends TimerTask {
   private int count;
   public void run() {
      count++;
      pen.drawRect(10 * count, 20 * count, 40, 30);
      frame.repaint(); // redraws the picture on the screen
   }                   // so we can see the update
}
```

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



Making our GUI an Object

- Usually we do not create a GUI in main, instead we make it in an object. Then in main we can write:

```
public static void main(String[] args) {
    new MyGui();
}
```

Accessing `ActionEvent`

- `public Object getSource()`
  Returns the object on which the Event initially occurred.

- `public String getActionCommand()`
  Returns the command string associated with this action.
  - this will be the button text if a button triggered the event

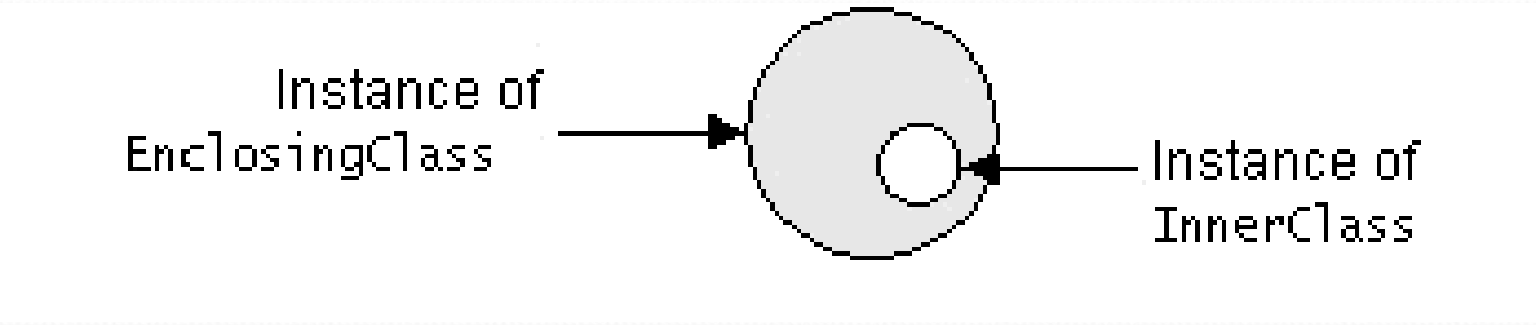
How can we access our `TextField`?

- We can't get it from the `ActionEvent` as no event has occurred on it
- We can't change the method header of `actionPerformed`
- It only exists in the constructor of our `TipCalculator` class

- Solution: make it a field?

Nested classes

- **nested class**: A class defined inside of another class.
- Usefulness:
  - Nested classes are hidden from other classes (encapsulated).
  - Nested objects can access/modify the fields of their outer object.
- Event listeners are often defined as nested classes inside a GUI.



Nested class syntax

```
// enclosing outer class
public class name {
    ...

    // nested inner class
    private class name {
        ...
    }
}
```

- Only the outer class can see the nested class or make objects of it.
- Each nested object is associated with the outer object that created it, so it can access/modify that outer object's methods/fields.
  - If necessary, can refer to outer object as **OuterClassName** . `this`

Static inner classes

```
// enclosing outer class
public class name {
    ...

    // non-nested static inner class
    public static class name {
        ...
    }
}
```

- Static inner classes are *not* associated with a particular outer object.
- They cannot see the fields of the enclosing class.
- *Usefulness:* Clients can refer to and instantiate static inner classes:

```
Outer.Inner name = new Outer.Inner(params);
```

GUI event example

```
public class MyGUI {
    private JFrame frame;
    private JButton stutter;
    private JTextField textfield;

    public MyGUI() {
        ...
        stutter.addActionListener(new StutterListener());
    }
    ...

    // When button is clicked, doubles the field's text.
    private class StutterListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            String text = textfield.getText();
            textfield.setText(text + text);
        }
    }
}
```

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.

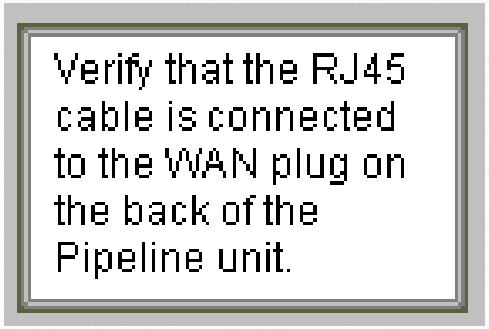
- `public String getText()`
  Returns the text showing on the button / label.

- `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.