

# GUI in Java (Basic Concepts)



- \*\* What is GUI (Graphical User Interface)
- \*\* Making GUIs (**JFrame**, **JBUTTON** in `javax.swing.*`)
- \*\* Event Handling: User Events, Listener Interfaces and Event Sources (`java.awt.event.*`)
- \*\* Layout Managers: **FlowLayout**, **GridLayout**, **BorderLayout**



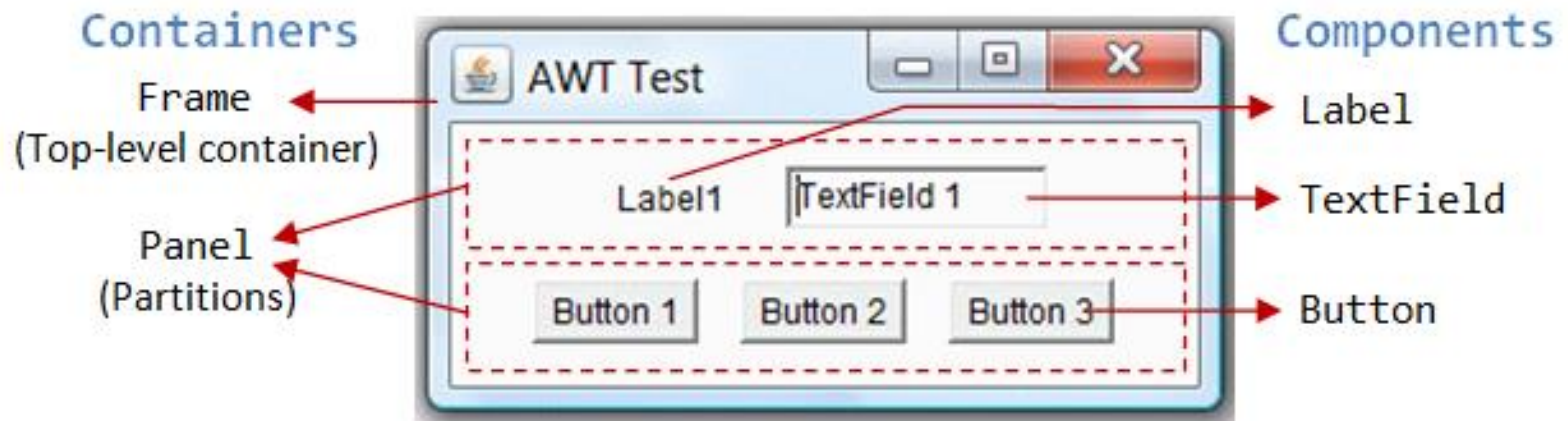
Chapters 10+11 (sections 10.1-10.2, 10.4, 11.1-11.4 – “Core Java” book  
Chapters 14, 15 – “Head First Java” book  
Chapters 16+17 (sections 16.1-16.4, 16.8, 17.1-17.5) – “Introduction to Java Programming” book

# What is GUI?

- **GUI**: Method for interacting with a computer via the manipulation of text, images and “widgets”.
  - GUIs display visual elements, e.g. buttons, icons, windows.
  - **Examples** of operating systems that support GUIs: MAC OS, Microsoft Windows.
- GUIs were introduced to address some of the issues with text based user interfaces (*aka* as CLIs – Command Line Interfaces), e.g. CLIs often require long command words to be typed in.
- **Widgets**: Things you can put in a window, such as a button.

# What is (in a) GUI?

- There are *3 main concepts* when doing GUI programming in Java:
  - Component**: An *object* that the user can see on the screen and can also interact with.
  - Container**: A *component* that can hold other components.
  - Event**: An *action* triggered by the user (e.g. pressing a key, clicking a mouse button).
- Designing a GUI** involves creating *components*, putting them into *containers*, and arranging for the program to respond to *events* (e.g. by responding to mouse clicks).



# java.awt versus javax.swing

- Historical **java.awt** **problems**: runtime peer resources were required.
  - **slow** on some **platforms** (e.g. Windows);
  - **portability problems** (slightly different look and behaviour).
- Why **javax.swing** **is better** ...
  - *More efficient use of resources*: Lightweight components are really "lighter" than heavyweight components.
  - *More consistency across platforms* because **Swing** is written entirely in Java.
  - *Cleaner look-and-feel integration*: Can give a set of *components* a matching look-and-feel by implementing them using **Swing**.



**javax.swing** components: e.g.  
JLabel, JList, JMenuBar.

# Some Background: `java.awt` Package (1/2)

- The `java.awt` package contains most of the classes needed to create GUI applications and Applets in Java.
- There are over 40 classes in the AWT package. They fall into the following general class types:
  - **Container Classes**: Graphical *widgets* capable of containing collections of other graphical widgets (i.e. `Panel`, `Window`, `Dialog` and `Frame`).
  - **Component Classes**: Atomic graphical widgets like `Button`, `Menu` and `List`.
  - **Layout Manager Classes**: Control the layout of component objects on/in container objects.

# Some Background: `java.awt` Package (2/2)

- More general class types in `java.awt` package:
  - **Primitive Graphics Classes**: Control and access primitive graphics like `Point`, `Rectangle` and `Polygon`.
  - **Event Handling Classes**: Deal with events received from the GUI and other system items.
  - **Listener Classes**: Receive events from graphical components and act on them.

# A First GUI

- Steps to **making a GUI**:

1. **Make a frame**: create an instance of **JFrame**

```
JFrame myFrame = new JFrame();
```

2. **Make a widget** (e.g. make a button or text field)

```
JButton myButton = new JButton("Click me");
```

3. **Add the widget to the frame**

```
myFrame.getContentPane().add(myButton);
```

4. **Display the frame**: must *give it a size* and *make it visible*

```
myFrame.setSize(100, 100);
```

```
myFrame.setVisible(true);
```

# Example: Simple GUI

```
import javax.swing.*;

public class SimpleGui {
    public static void main(String[] args) {
        JFrame myFrame = new JFrame();
        JButton myButton = new JButton("Click me");
        myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        myFrame.getContentPane().add(myButton);
        myFrame.setSize(200, 200);
        myFrame.setVisible(true);
    }
}
```



What happens when  
you click the button?

Running program ...

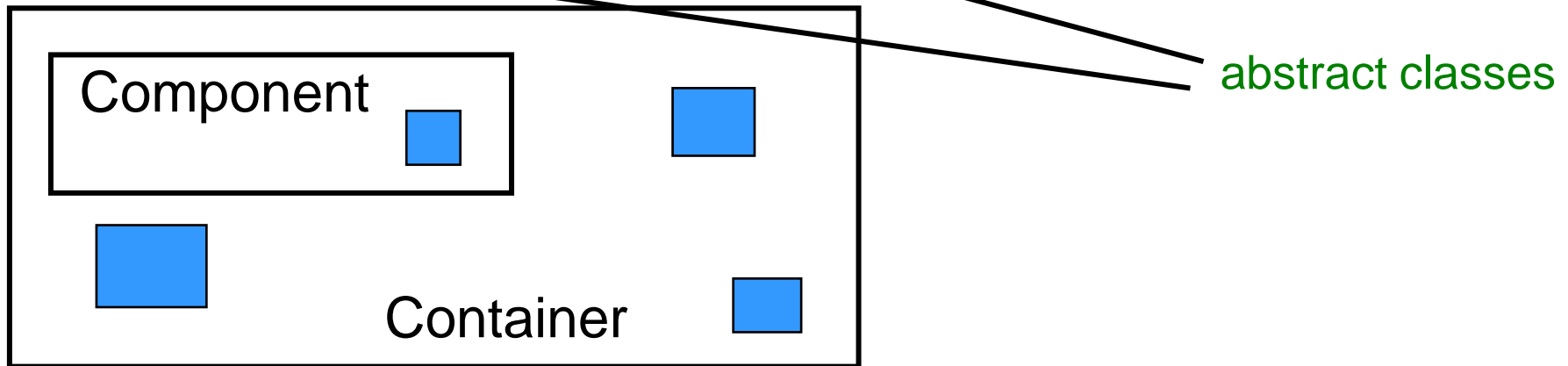
```
> java SimpleGui
```





# Containers *versus* Components: What

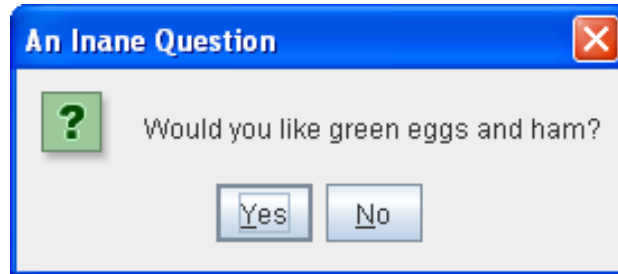
- **Containers**: objects capable of containing other **Component** objects.
- **Components**: single entities with no *containment* abilities.



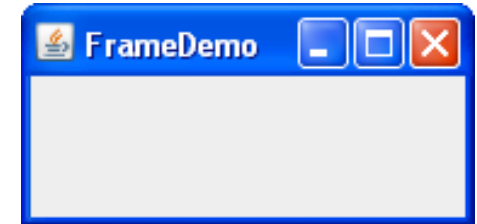
The filled boxes are **components**. They can be buttons on screen. There is **no nesting** for components.

# Containers: Examples

- *Top-level Containers*: At least one of these containers *must* be present in any **Swing** application.

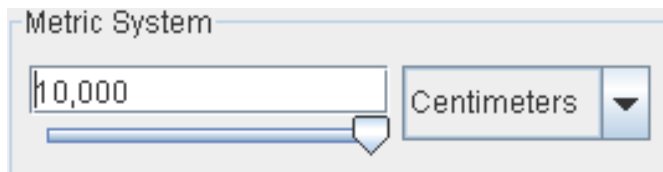


**JDialog**



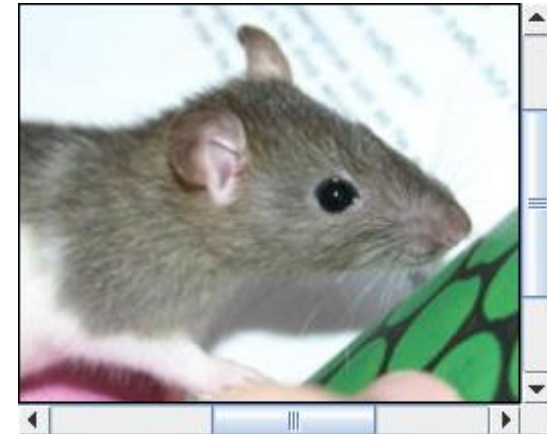
**JFrame**

- *General-purpose Containers*:  
Found in most **Swing** applications.



**JPanel**

**JScrollPane**

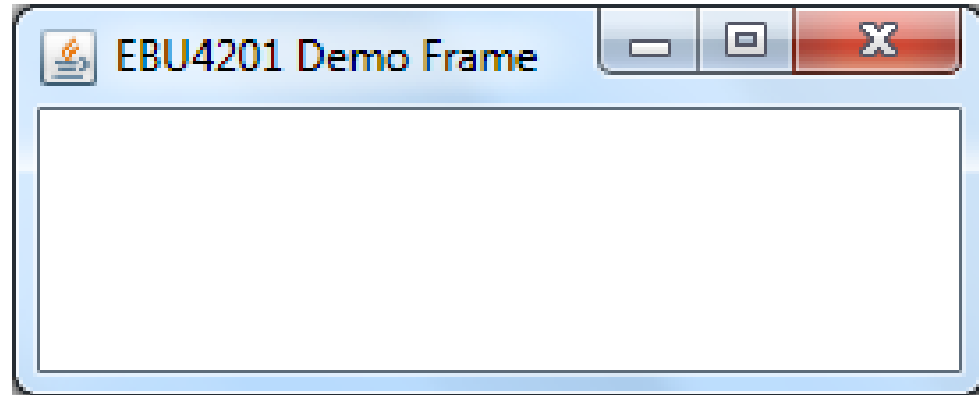


Other relevant examples (and how to use them) at

<http://docs.oracle.com/javase/tutorial/uiswing/components/componentlist.html>

# Frame: one of the most used containers

- A **Frame** is a simple, resizable **window** with a border, title bar and possibly a menu bar.



- You can **extend Frame** in your program (more common), or **instantiate the Frame** class in your own class (less common) to build a basic GUI.
- Frame **defaults**:
  - Initially created with **0** size → `setSize(int,int);`
  - Initially created **invisible** → `setVisible(boolean);`
- To **change the text in a Frame object's title bar**, use the method **`setTitle(String);`**.

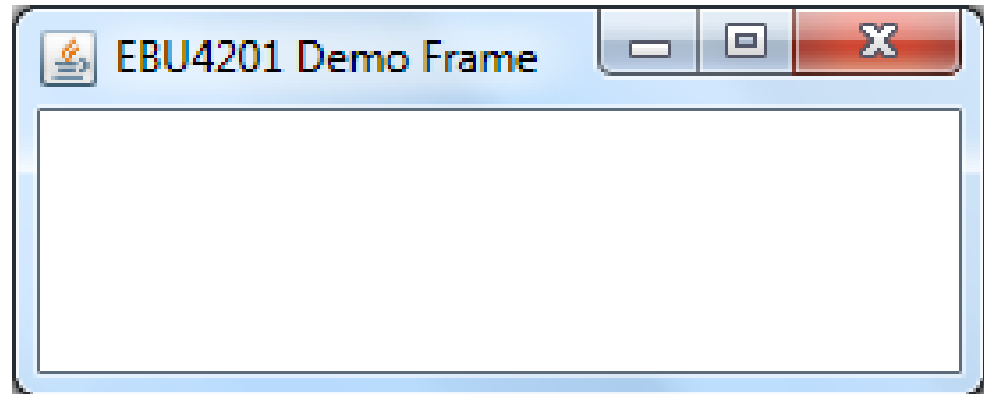
# Example: Using java.awt package

```
import java.awt.Frame;

public class FrameDemo extends Frame {
    public FrameDemo() {
        this.setTitle("EBU4201 Demo Frame");
        this.setSize(250,100);
        this.setVisible(true);
    }
    public static void main(String[] args) {
        FrameDemo myFrame = new FrameDemo();
    }
}
```

Output is ...

> java FrameDemo



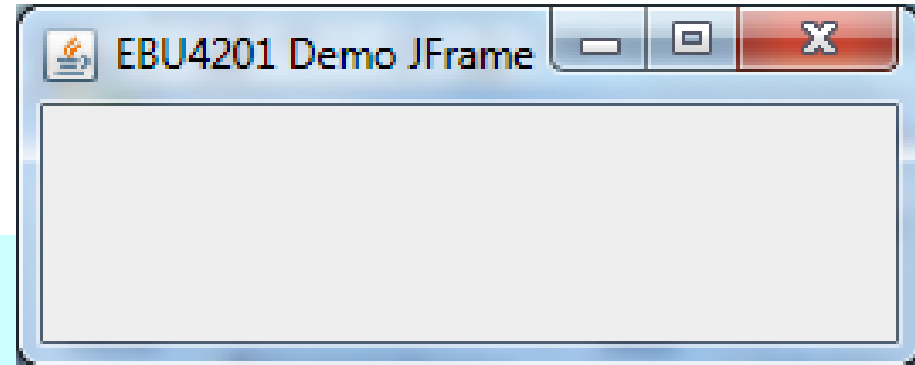
# Example: Using javax.swing package

```
import javax.swing.JFrame;

public class FrameDemo extends JFrame {
    public FrameDemo() {
        this.setTitle("EBU4201 Demo JFrame");
        this.setSize(250, 100);
        this.setVisible(true);
    }
    public static void main(String[] args) {
        FrameDemo myFrame = new FrameDemo();
    }
}
```

Output is ...

> java FrameDemo



How to use the  
javax.swing package (tutorial):

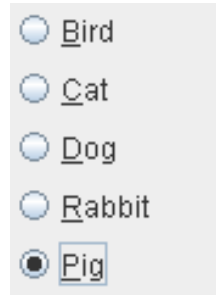
<http://docs.oracle.com/javase/tutorial/uiswing/>

# Components: Examples

- *Basic Swing Components*: Used mainly for getting input from the user.



**JButton**



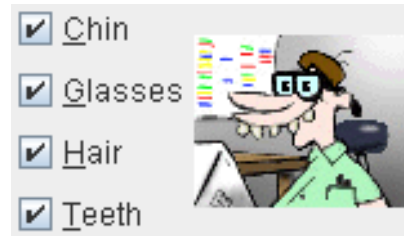
**JRadioButton**



**JList**



**JTextField**



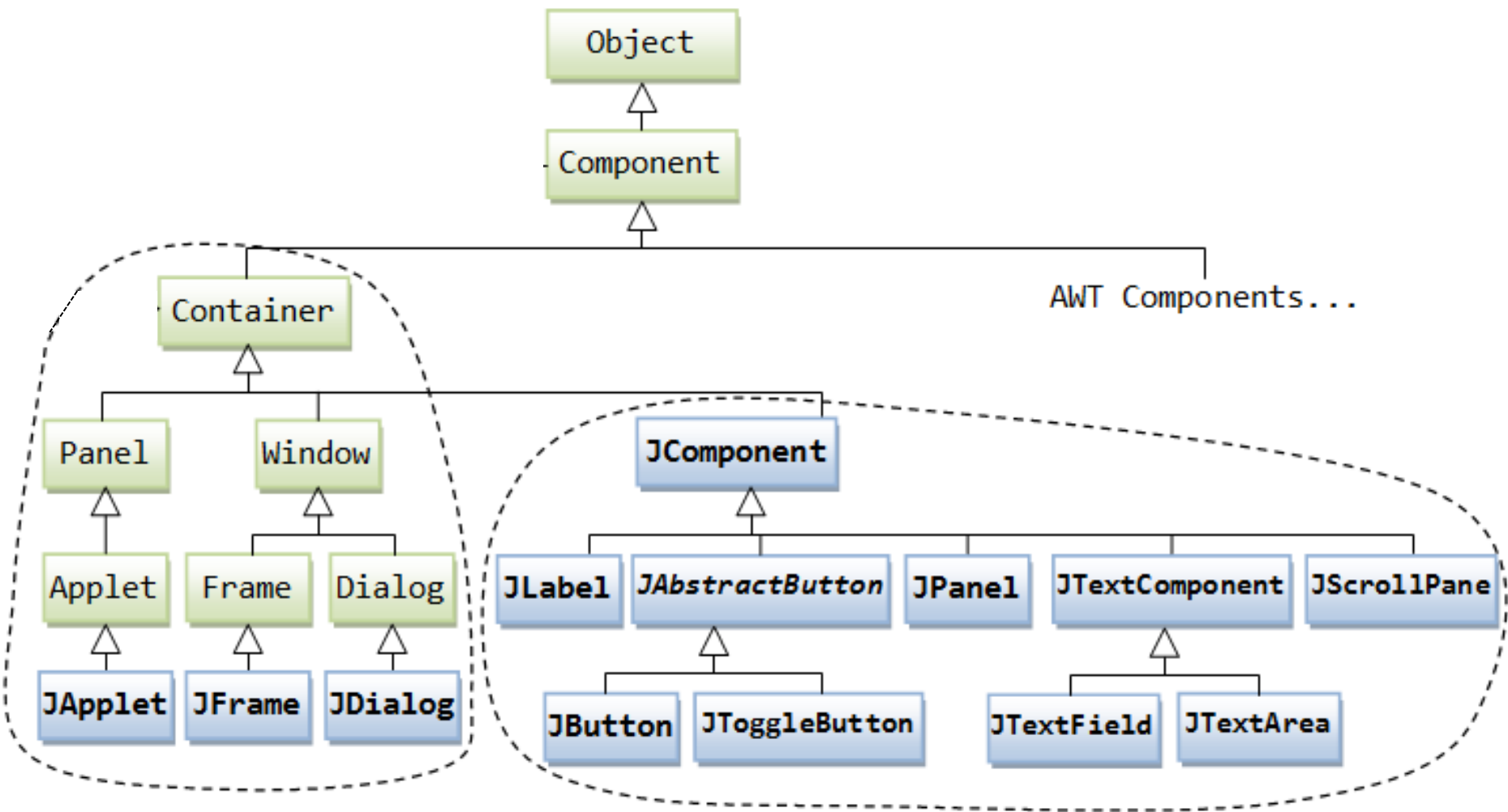
**JCheckBox**



Other relevant examples (and how to use them) at

<http://docs.oracle.com/javase/tutorial/uiswing/components/componentlist.html>

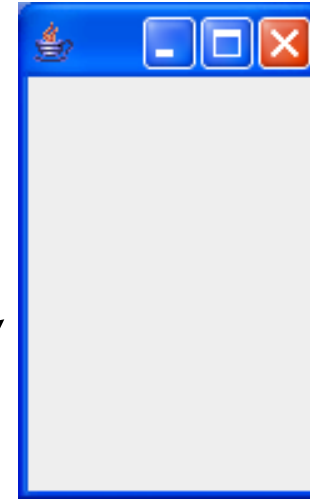
# Containers and Components : Where in the Java API



# Practice Exercise 1

- The program below is supposed to display a message on the panel, but **nothing is displayed**. There are 2 problems; identify them.

```
import javax.swing.JFrame;
public class TestDrawMessage extends JFrame {
    public void TestDrawMessage() {
        getContentPane().add(new DrawMessage());
    }
    public static void main(String[] args) {
        JFrame frame = new TestDrawMessage();
        frame.setSize(100,200);
        frame.setVisible(true);
    }
}
```



```
import javax.swing.JPanel;
import java.awt.Graphics;
class DrawMessage extends JPanel {
    protected void PaintComponent (Graphics g) {
        super.paintComponent(g);
        g.drawString("Welcome to Java",20,20);
    }
}
```





---

# **EVENTS AND DEALING WITH USER INTERACTIONS**

# Java & Event Driven Programming

- A (user) *event* is triggered any time when some sort of defined signal is received by the program.
  - An event is generated by external user actions, e.g.
    - typing a character;
    - mouse button clicks or movement;or by the operating system, e.g. a timer going off.
- **Event handling**: the process of getting and handling user events.

# Example *Events*

## Event Generating Action

User clicks a button, presses Return while typing in a text field, or chooses a menu item

## Listener Type

ActionListener

## Event Type

ActionEvent

User closes a frame (main window)

WindowListener

WindowEvent

User presses a mouse button while the cursor is over a component

MouseListener  
MouseMotionListener

MouseEvent  
MouseEvent

User moves the mouse over a component

ComponentListener

ComponentEvent

Component becomes visible

FocusListener

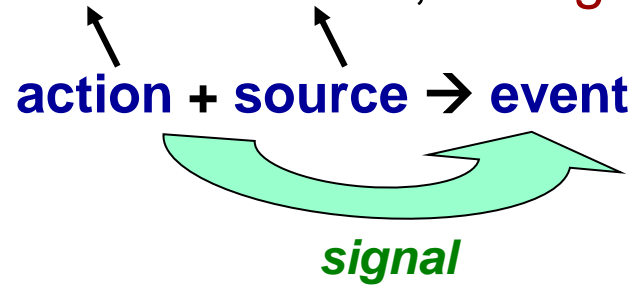
FocusEvent



Other **event listeners and types** available in `java.awt.event.*`.

# Doings Things in GUI

- **Example:** When I **click** on **button**, **change** the button text.



- **Need to know:**
  - Which user **action** leads to a **change**: e.g. clicking, moving mouse, pressing return key ...
  - The corresponding widget (**component**).
  - What needs to happen (or **change**) as a result of the **action** on the source (i.e. the event).
    - But still need to be able to get (and **handle**) **the event**.

# Events, Sources & Listeners

- You need an **listener** and a **source** for each **event**
- **Listener Interface**: the *bridge* between the **listener** (the user code) and the **event source** (e.g. the button).
  - Implementing a listener interface gives the button a way to call the user code back.
- **Event source**: **object** that can **turn user actions** (e.g. click a mouse, close a window) **into events**.
- Every **event type** has a matching listener interface.
  - **Example**: For **MouseEvents**, you need to implement the **MouseListener** interface.
  - You must provide **implementations for its methods**.

# Steps: Writing an Event Handler

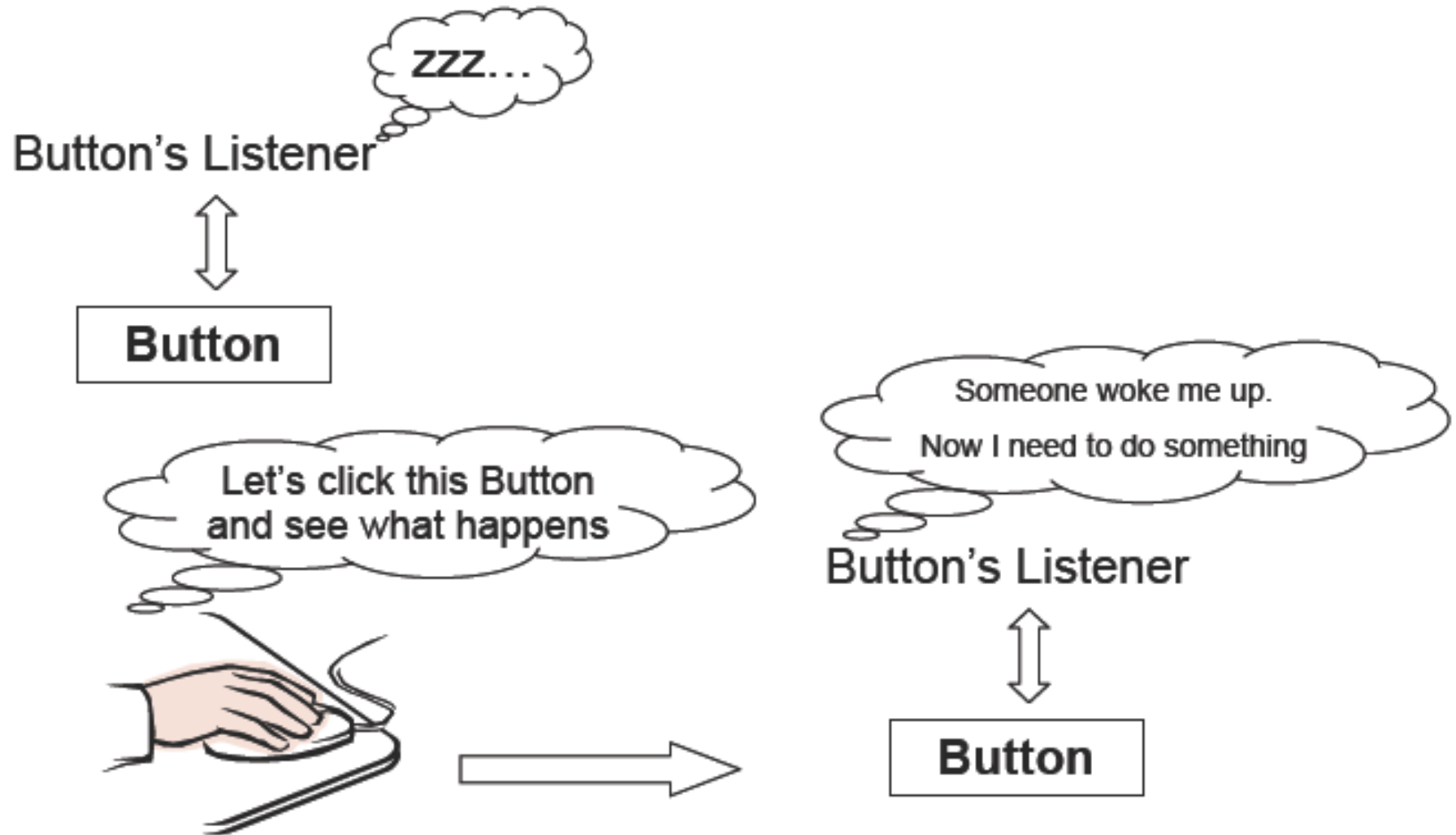
- Every **event handler** (e.g. how to get a button's **ActionEvent**) requires three bits of code:
  1. **Implement the ActionListener interface**: In the declaration for the event handler class, code specifies that class either implements a listener interface **or** extends a class that implements a listener interface.

```
public class MyClass implements ActionListener {...}
```
  2. **Register with the widget**: Code indicates that your program wants to listen for events, by registering an instance of the event handler class as a listener upon one or more components.

```
someComponent.addActionListener(instanceOfMyClass);
```
  3. **Define the event-handling method**: Code implements the methods in the listener interface.

```
public void actionPerformed(ActionEvent e) {  
    // code that reacts to the action ...  
}
```

# Writing an Event Handler (*visual interpretation*)



# Example: Event Handler (1/2)

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

public class AnotherSimpleGui implements ActionListener {
    JButton myButton;

    public static void main(String[] args) {
        AnotherSimpleGui myGui = new AnotherSimpleGui();
        myGui.go();
    }

    public void go() {
        JFrame myFrame = new JFrame();
        myButton = new JButton("Click me");
        myButton.addActionListener(this);
        myFrame.add(myButton);
        myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        myFrame.setSize(200, 200);
        myFrame.setVisible(true);
    }

    public void actionPerformed(ActionEvent event) {
        myButton.setText("I've been clicked");
    }
}
```

**Step 1** – implement interface **ActionListener**

**Step 2** – register interest with button (tell button “I want to listen to actions on you”).

**Step 3** – implement **ActionListener**’s interface method (it handles the event).



# Example: Event Handler (2/2)

Running program ...

```
> java AnotherSimpleGui
```



upon clicking button ...



**Homework:** Add code so that you can change the colour of the button.

# Other Event Types: MouseEvents

## Action

- Click
- Press
- Release
- Enter
- Exit
- Move
- Drag

## Source



## Event

mouseClicked(MouseEvent e)

mouse\_\_\_\_\_ (MouseEvent e)

mouse\_\_\_\_\_ (MouseEvent e)

mouse\_\_\_\_\_ (MouseEvent e)

mouse\_\_\_\_\_ (MouseEvent e)

mouse\_\_\_\_\_ (MouseEvent e)

mouse\_\_\_\_\_ (MouseEvent e)



Fill in the gaps to complete the method names.

# Other Event Types: MouseEvents

## Action

- Click
- Press
- Release
- Enter
- Exit
- Move
- Drag

## Source



## Event

`mouseClicked(MouseEvent e)`  
`mousePressed(MouseEvent e)`  
`mouseReleased(MouseEvent e)`  
`mouseEntered(MouseEvent e)`  
`mouseExited(MouseEvent e)`  
`mouseMoved(MouseEvent e)`  
`mouseDragged(MouseEvent e)`



Fill in the gaps to complete the method names.

# Practice Exercise 2

- What is wrong in the following code? Spot 7 problems!

```
import java.awt.*;
import java.swing.*;
public class Test extends JFrame implements ActionListener {
    public Test() {
        JButton jbtOK = new JButton("OK");
        getContentPane().add(jbtOK);
    }
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == jbtOK)
            System.out.println("OK button is clicked");
    }
}
```

**Homework**

Write a simple GUI program with a button that responds to events from a mouse being pressed and released. The *button should display*:

- the message "No action" when no action is taken on the mouse.
- the message "Pressing down" when the mouse is pressed.
- the message "Releasing" when the mouse is released.

# JLabel and JButton Classes

- **JLabel**: component that you can put text into.
  - When **creating a label**, you can **specify the initial value** and the **alignment you wish to use** within the label.  

```
JLabel myLabel = new JLabel("text", JLabel.RIGHT);
```
  - You can use methods **getText()** and **setText()** to **get and change the value of the label**, respectively.
- **JButton**: extends **Component**, displays a string and delivers an **ActionEvent** for each mouse click.
  - Normally **buttons** are **displayed with a border**.
  - In addition to text, **JButtons** can also **display icons**.

```
JButton myButton = new JButton("text");
```

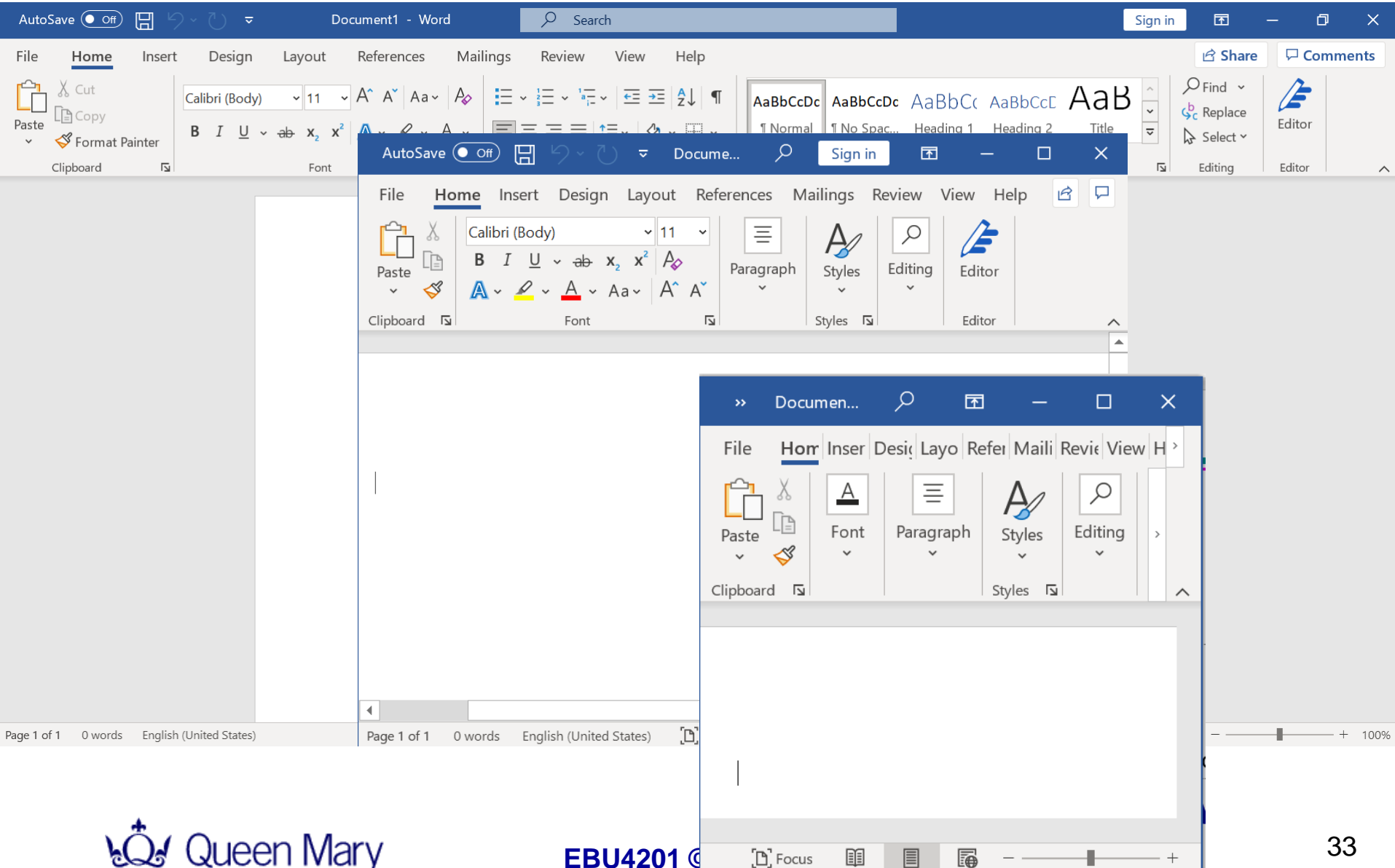
# Components, Containers & Layout Managers

- **Layout Manager**: An interface that defines methods for positioning and sizing objects within a container.
  - Java defines several default implementations of **LayoutManager**.
  - **Geometrical placement in a Container** is controlled by a **LayoutManager** object.
- **Containers** may contain components → so containers can contain containers!
  - All containers come equipped with a *layout manager* which positions and shapes (lays out) the container's components.
  - Much of the **action in the AWT** occurs between *components*, *containers*, and their *layout managers*.

---

# MANAGING GUI LAYOUTS

# Layout is Hard!





# Layout Managers

- **Layouts** allow you to format components on the screen in a platform independent way.
  - **Layout managers** are defined in the `java.awt` package.
  - The standard JDK provides five classes that implement the **LayoutManager** interface:
    - `FlowLayout`
    - `GridLayout`
    - `BorderLayout`
    - `CardLayout`
    - `GridBagLayout`
- } *Not discussed here!*

# Changing the Layout

- **Steps to change the layout** in a container:
  - **Step 1**: Create the layout.
  - **Step 2**: Invoke the `setLayout()` method on the container to use the new layout.

```
JPanel p = new JPanel();  
p.setLayout(new FlowLayout());
```

- **Or both Steps** at once:

```
JPanel p = new JPanel(new FlowLayout());
```



The **layout manager** should be **established before** **any components are added** to the container.

# FlowLayout

- **FlowLayout** is the **default layout** for the **JPanel** class.
  - When you **add components to the screen**, they **flow left to right** (centered), based on the order added and the width of the screen.
  - Very **similar to** *word wrap* and *full justification* on a **word processor**.
  - If the screen is resized, the components' flow will change based on the new width and height.
  - **Constructors:**
    - `FlowLayout()`
    - `FlowLayout(int align)`
    - `FlowLayout(int align, int hgap, int vgap)`

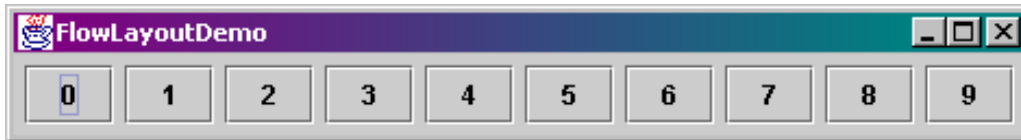


This is the **default layout manager** for a *pane*!

# Example using FlowLayout

```
import javax.swing.*;
import java.awt.FlowLayout;
public class FlowLayoutDemo extends JFrame {
    public FlowLayoutDemo(String title, int num) {
        this.setTitle(title);
        this.getContentPane().setLayout(new FlowLayout());
        for (int i = 0; i < num; i++)
            this.getContentPane().add(new JButton("" + i));
    }
    public static void main(String args[]) {
        FlowLayoutDemo frame = new FlowLayoutDemo("FlowLayoutDemo",10);
        frame.pack();
        frame.setVisible(true);
    }
}
```

Output is ...

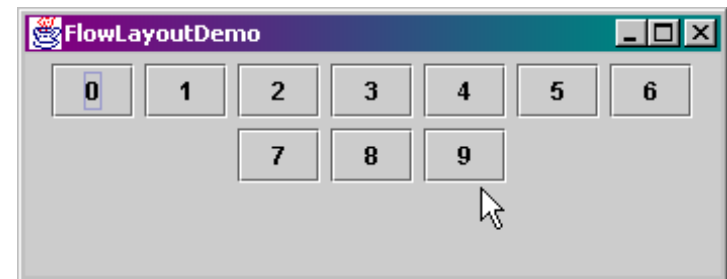


Resizing the window:

(1)



(2)



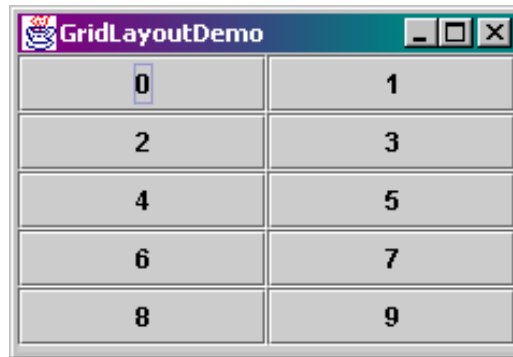
# GridLayout

- **GridLayout** arranges components in **rows** or **columns**:
  - If **number of rows is specified**, the number of columns will be the number of components divided by the rows.
  - If **number of columns is specified**, the number of rows will be the number of components divided by the columns.
  - The **order in which you add components is relevant**.
  - **Constructors**:
    - `GridLayout()` : default of 1 column per component, in a single row.
    - `GridLayout(int rows, int cols)`
    - `GridLayout(int rows, int cols, int hgap, int vgap)`

# Examples using GridLayout



`GridLayout(10,0)`



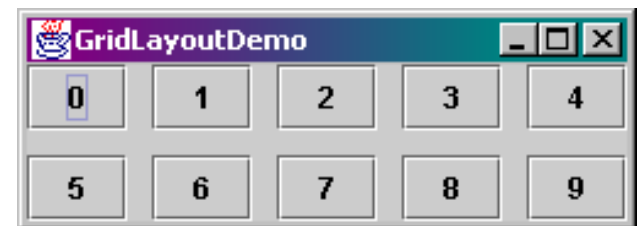
`GridLayout(0,2)`



`GridLayout(0,4)`



`GridLayout(0,2) (Resized)`



`GridLayout(0,5,10,10)`

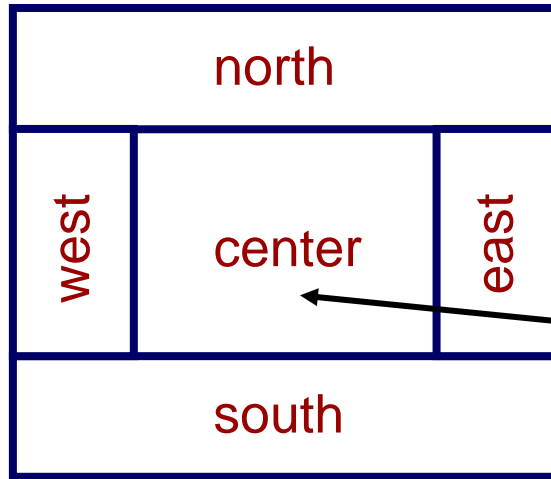
# BorderLayout

- **BorderLayout** provides 5 areas to hold components.
  - These are named after the four different borders of the screen, **North**, **South**, **East**, **West**, and **Center**.
  - To **add a Component**, you must **specify which area to place it in**.
  - The **order** in which components are created **is not important**.
  - The **NORTH** and **SOUTH** components **may be stretched horizontally**.
  - The **EAST** and **WEST** components **may be stretched vertically**.
  - The **CENTER** component may stretch both horizontally and vertically to **fill any space left over**.
  - **Constructors:**
    - `BorderLayout()`: default; provides no gaps between components.
    - `BorderLayout(int hgap, int vgap)`



This is the **default layout manager for a frame!**

# BorderLayout



default region  
to add things to

- How to add a button to a frame:

```
myFrame.getContentPane().add(myButton);
```

**OR**

```
myFrame.getContentPane().add(BorderLayout.CENTER, myButton);
```

recommended way  
of adding a button

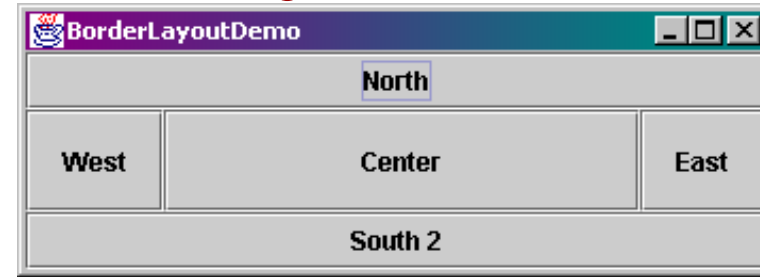


# Example using BorderLayout

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

public class BorderLayoutDemo extends JFrame {
    public BorderLayoutDemo() {
        setTitle("BorderLayoutDemo");
        Container content = getContentPane();
        content.setLayout(new BorderLayout());
        content.add(BorderLayout.NORTH, new JButton("North"));
        content.add(BorderLayout.SOUTH, new JButton("South"));
        content.add(BorderLayout.EAST, new JButton("East"));
        content.add(BorderLayout.WEST, new JButton("West"));
        content.add(BorderLayout.SOUTH, new JButton("South 2"));
        content.add(BorderLayout.CENTER, new JButton("Center"));
    }
    public static void main(String args[]) {
        BorderLayoutDemo frame = new BorderLayoutDemo();
        frame.pack();
        frame.setVisible(true);
    }
}
```

Resizing the window:



Output is ...

