CS5001 Object Oriented Modelling Design and Programming
Week 7

Graphical User Interfaces (GUIs)

Fahrurrozi Rahman & Xu Zhu
School of Computer Science
University of St Andrews

GUI DESIGN PATTERNS

MVC and MD

The Model–View–Controller Pattern (MVC)

- Many applications need some kind of user interface
  - a graphical user interface
  - A textual interface
  - An interface containing physical controls like buttons and switches
  - Some hybrid of the above

- Some tools present different interfaces depending on circumstances
- File System has two interfaces:
  - A command line interface
  - A graphical user interface

How Do We Engineer the Interface?

- Clearly it is possible to *bodge* user interface code into the middle of classes, for example:

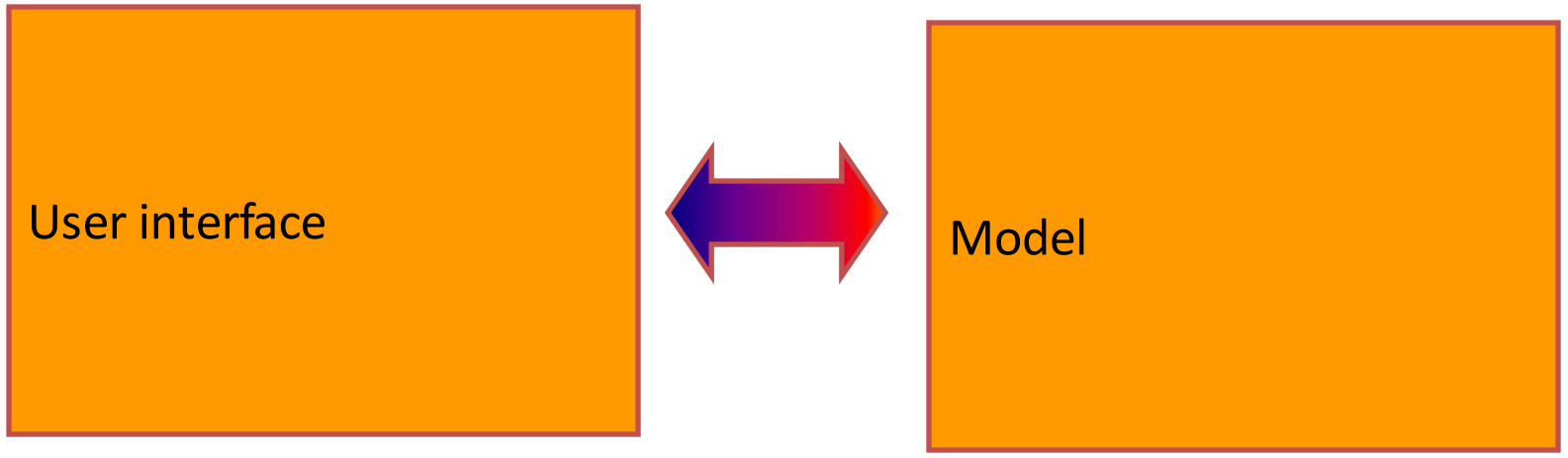
```
public class Frog {
    private String colour;
    private int length;
    private BufferedReader br = new BufferedReader(
                                    new InputStreamReader(System.in));
    public Frog() {
        System.out.println( "what colour is your frog?" );
         try{
            colour = br.readLine();
        } catch (IOException e){ System.err.println(e.getMessage()); }
        System.out.println( "how long is your frog?" );
        try {
            length = Integer. parseInt(br.readLine());
        } catch (IOException e){ System.err.println(e.getMessage()); }
    }
}
```

Using the Frog Class

- The problem with putting I/O code into a class like Frog is that **we do not know** where the Frog code is going to be used
  - On a Unix machine with only a textual interface
  - From a Graphical User Interface with buttons
  - From a Web page
  - In an embedded application – such as a environmental frog monitoring station with no I/O
  - On a phone (or handheld device with a tiny screen)
  - On a physical device such as a child's toy with physical (real) buttons

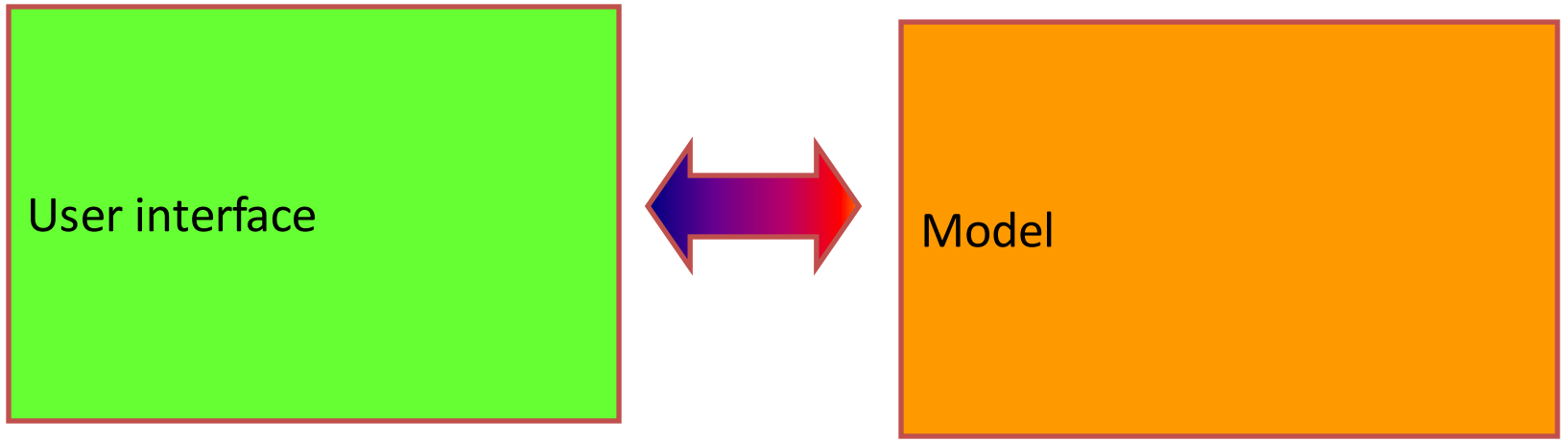
A Better Model

- A better way of dealing with the issue of I/O is to separate the model from the user interface
- So we can have the idea of a model and a user interface for the model



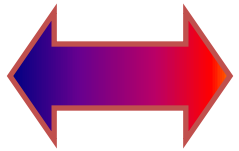
The Model–View Paradigm

- A better way of dealing with the issue of I/O is to separate the model from the user interface
- So we can have the idea of a model and a user interface for the model – this is often called the **model–view paradigm** or **model–view pattern**



The Model–View Paradigm

- A better way of dealing with the issue of I/O is to separate the model from the user interface
- So we can have the idea of a model and a user interface for the model – this is often called the **model–view paradigm** or **model–view pattern**



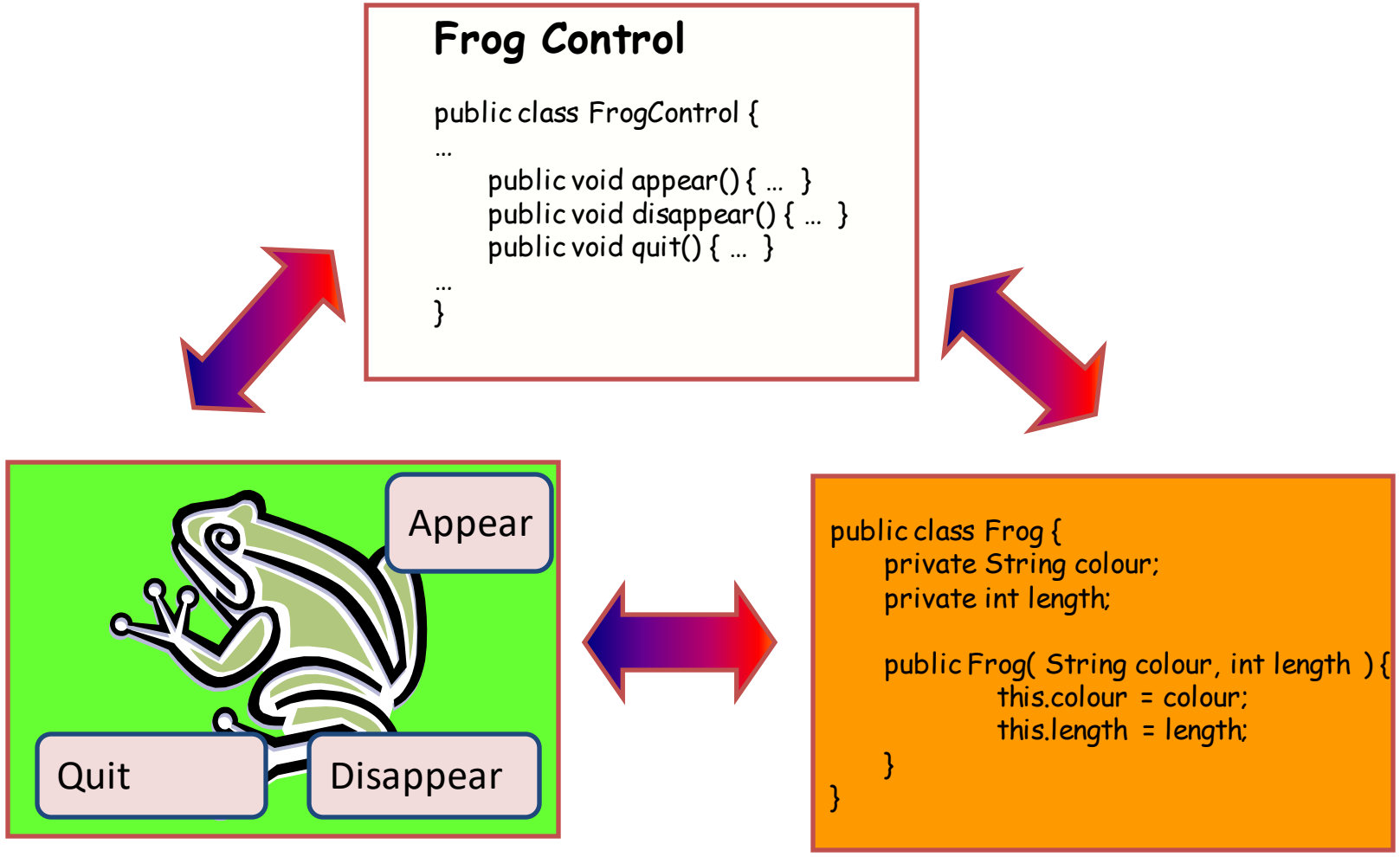
```
public class Frog {
    private String colour;
    private int length;

    public Frog( String colour, int length ){
            this.colour = colour;
            this.length = length;
    }
}
```

The Controller Element

- Just as it is possible to separate the **viewing** of a model from the actual model, it may be useful to separate out the **control**

- So we end up with three separate elements:
  - The **model** – the real world entities being modelled
  - The **view** – how we see the model
  - The **controller**
    - link between user action and model manipulation
      - specifies logical action to perform on model given UI button press, etc.
      - then manipulates the model – making ships, frogs, people etc.

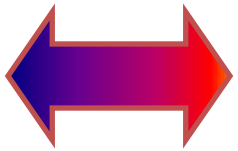
The Model–View–Controller Paradigm

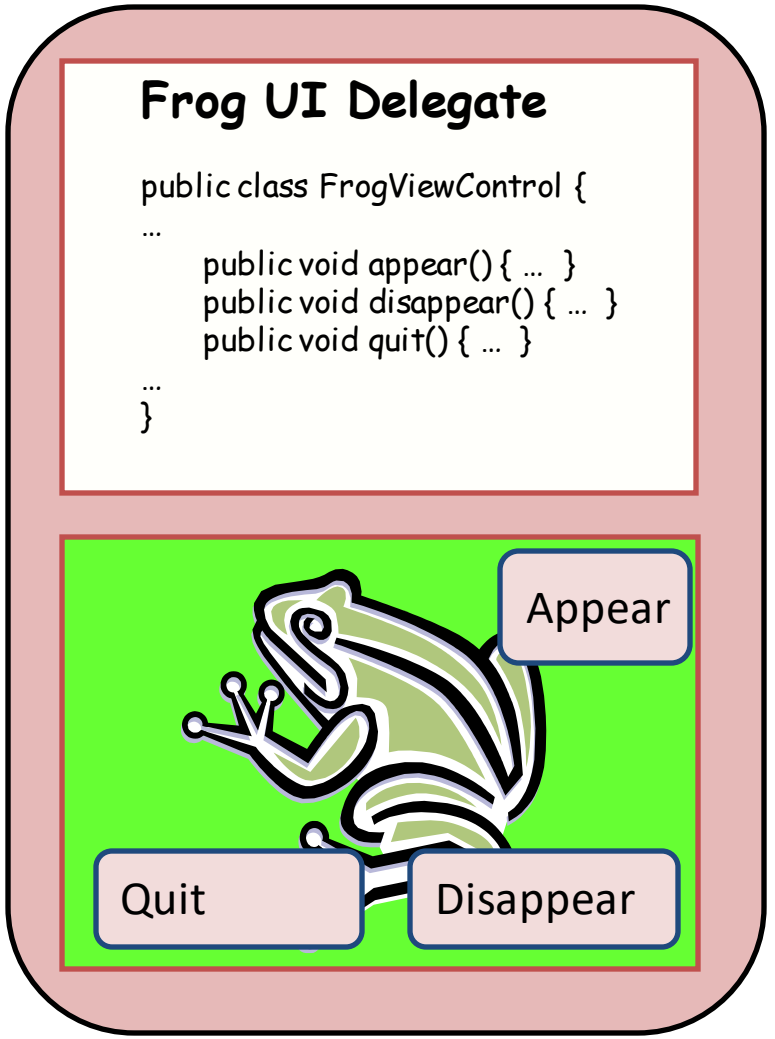


The Model–Delegate Paradigm

- Simplification of MVC
  - Model–View paradigm where view contains controller
  - the Controller and View are merged into a single User Interface (UI) **Delegate** component

```
public class Frog {
      private String colour;
      private int length;

      public Frog( String colour, int length ){
                this.colour = colour;
                this.length = length;
      }
}
```





LINKING MODEL AND VIEW

Linking the Model and the View

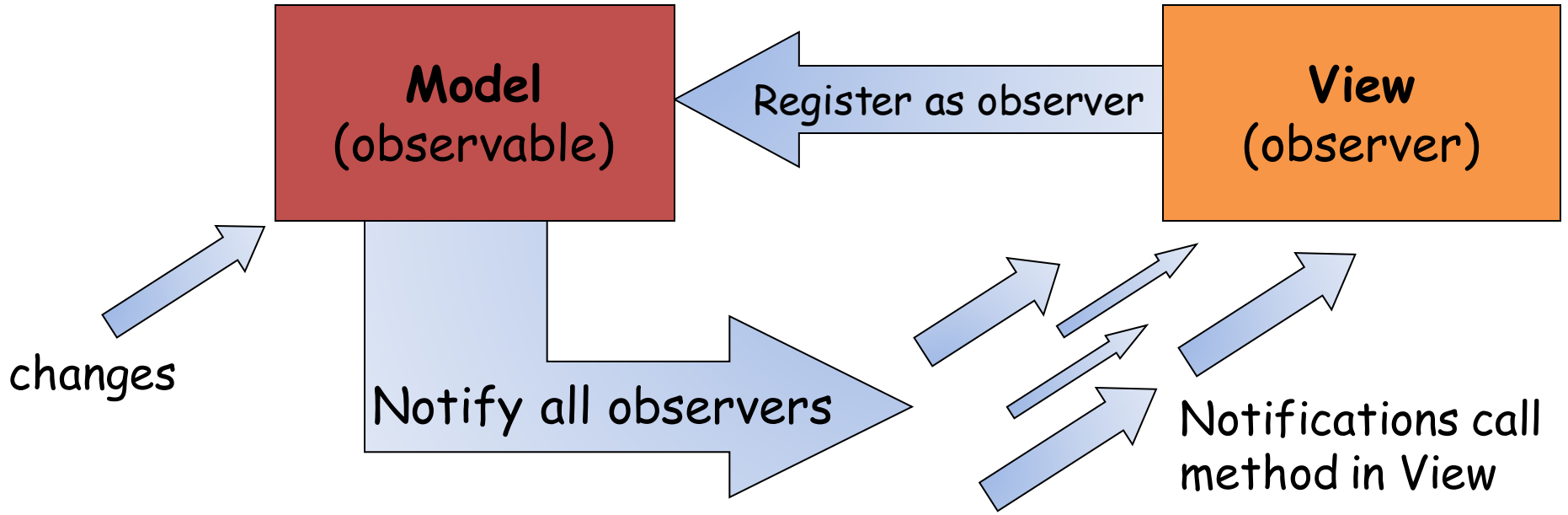
- The difficult part of linking the Model and the View is keeping them separate
- Here's an example where the model tries to update the view:

```
public class Frog {
    FrogGUI gui;
    ...
    public void setLength(int length) {
        this.length = length;
        this.gui.setLengthLabel(length);
    }
}
```

- This is bad because:
  - The model has to know details about how the view works
  - The model only works with one view (so it can't be reused with a different GUI)
- We want a model that knows **nothing** about the user interface

Observers

- The **observer** pattern is a common design pattern for linking model to view
  - Also known as the **listener** pattern
- View acts as an *observer* (aka *listener*) and registers itself with the model
- Model keeps a list of observers and notifies them of any changes



Observer pattern in Java

- No special library required
  - Implement it yourself!
- Interface for observers (aka listeners)
  - View implements this interface
- Model keeps a list of observers
  - has a method to add a new one
- Model notifies all observers when a change is made by calling their `update()` method
  - Different observers handle things differently, but they must all implement `update()` because of the interface

```
public class Frog {

    public interface Listener {
        // View must implement this
        void update();
    }

    private List<Listener> listeners;

    public void addListener(Listener listener) {
        listeners.add(listener);
    }

    // Called when something changes
    private void changed() {
        for (Listener listener : listeners) {
            listener.update();
        }
    }
    ...
}
```

Sending information to observers

- In the last example, we called `listener.update();` to announce changes
- We might want to include more information:

```
listener.update();  // Something changed. Look and get the new value.
listener.update(17);  // The new value is 17
listener.update("size", 17);  // size changed to 17
listener.update("size", 13, 17);  // size changed from 13 to 17
```

- Think about what's right for your project
  - and write the appropriate method in the interface:

```
void update();
void update(int newValue);
void update(String name, int newValue);
void update(String name, int oldValue, int newValue);
```

Observer: useful Java features

- **PropertyChangeListener** interface can be used to make this easier.
  - https://docs.oracle.com/en/java/javase/17/docs/api/java.desktop/java/beans/PropertyChangeListener.html
- Observers can implement PropertyChangeListener
  - must implement *propertyChange* method, which takes a PropertyChangeEvent parameter
- Model can have a **PropertyChangeSupport** object to handle the listeners
  - pre-defined methods for *add/removePropertyChangeListener* and *firePropertyChange*
    - https://docs.oracle.com/en/java/javase/17/docs/api/java.desktop/java/beans/PropertyChangeSupport.html
- See MVCGuiExample on StudRes:
  - https://studres.cs.st-andrews.ac.uk/CS5001/Examples/W07_GUIs/4_MVCGuiExample/

- Note: Java's 'Observer' and 'Observable' features are *deprecated*
  - Don't use them, as they may stop working soon!

SWING

A Java windowing toolkit

Java Windowing Toolkits

- No need to create your GUI from scratch, Java has Windowing Toolkits which provide
  - Widgets (*Window Gadgets*) such as Buttons, Toolbars, Menus, etc.
  - Event Notification system to allow user programs to act
    - Events for button presses, mouse movements etc.
- We will focus on the *Swing* toolkit
  - There are alternatives: *JavaFX* and *Google Web Toolkit (GWT)*
  - Many principles are the same
- GWT and JavaFX simplify GUI impl. for web applications

Swing Components

- GUIs are composed of components
- Top level swing Component
  - JFrame (Desktop window)
  - Lots of components all starting with **J**
    - JMenuBar, JPanel, JButton, JLabel, JTextField, JScrollPane, JOptionPane, etc.
    - Check the javax.swing API

Hello World

```
public class HelloWorld extends JFrame {

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

        public HelloWorld() {
                JLabel jlbHelloWorld = new JLabel("Hello World");
                getContentPane().add(jlbHelloWorld);
                this.setSize(100, 100);
                setVisible(true);
                setDefaultCloseOperation(EXIT_ON_CLOSE);
        }
}
```

1 2 3 4 5 6

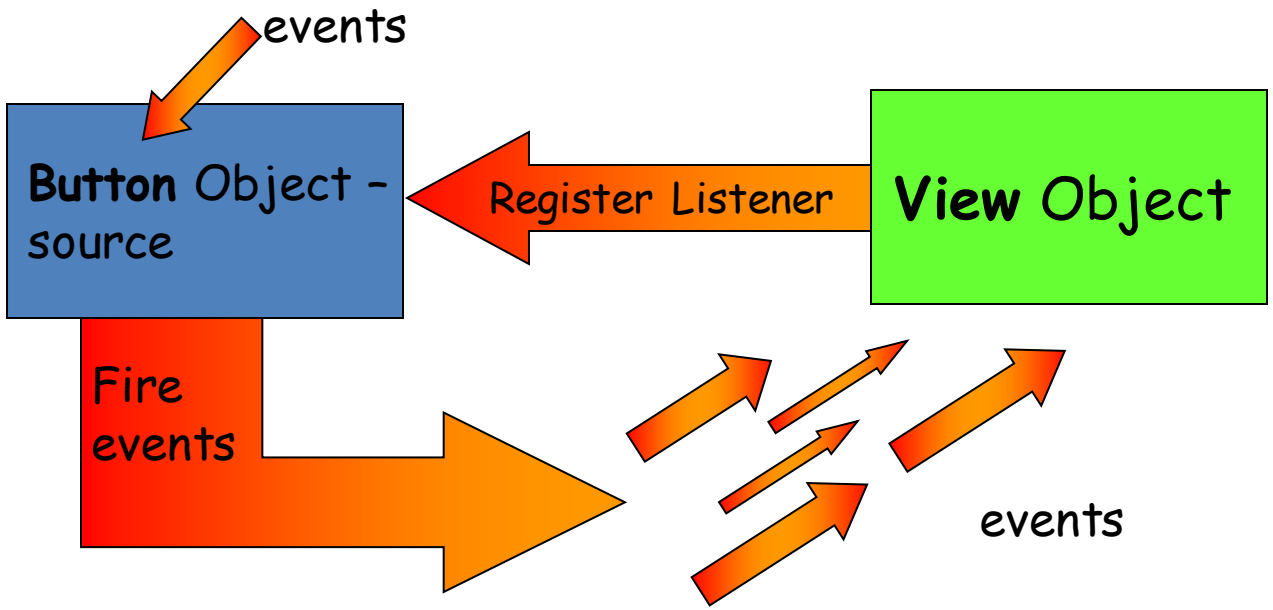
Hello World explanation

- Our object extends JFrame so it is a top level Component i.e. a window (1)
  - Could have used a separate JFrame object
- Create a label (2)
- Add the label to the JFrame's content pane (window) using the default layout manager (3)
- Set the size of the JFrame (4)
- Show the JFrame (5)
- Set the default action on closing the window (6)

EVENTS IN SWING

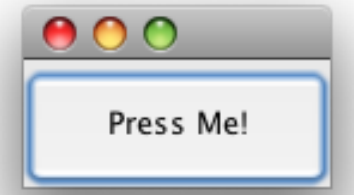
Handling Events

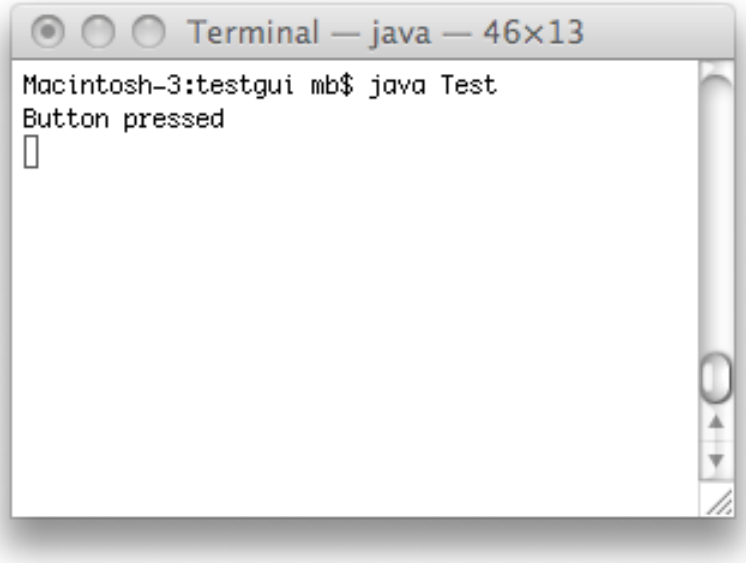
- Java GUI components use an event notification system based on the *observer* pattern
- The **View** registers *Listeners* (event handlers) with a *Source* (e.g. a Button, the main JFrame, a JPanel)
- *Listeners* are objects (complying with a suitable Interface) containing your own methods that handle UI events
  - the methods are called when e.g. a Button is pressed, the Mouse is moved …



ActionListener

```
public class EgListener implements ActionListener {
    public void actionPerformed (ActionEvent e) {
        System.out.println ("Button pressed");
    }
}

public class Test extends JFrame {
    public Test() {
        JButton button = new JButton("Press Me!");
        button.addActionListener(new EgListener());
        getContentPane().add(button);
        setSize(75, 75);
        setVisible (true);
    }
}
```

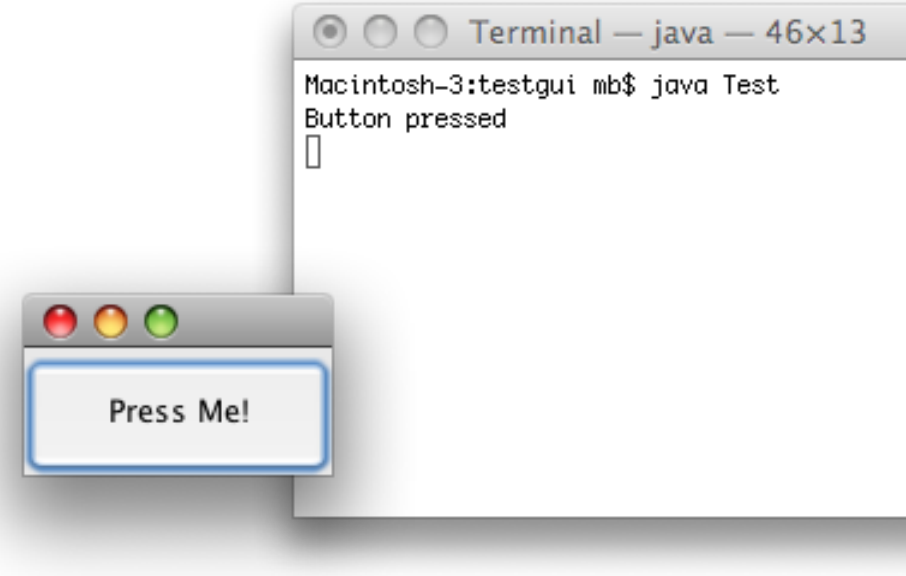




Anonymous inner class Example

```
public class Test extends JFrame {
    public Test() {
        JButton button = new JButton ("Press Me!");
        button.addActionListener (new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    System.out.println ("Button pressed");
                }
        });
        getContentPane ().add (button);
        setSize (75, 75);
        setVisible (true);
    }
    public static void main (String[] args) {
        new Test ();
    }
}
```

Anonymous inner class!



Listener Interfaces

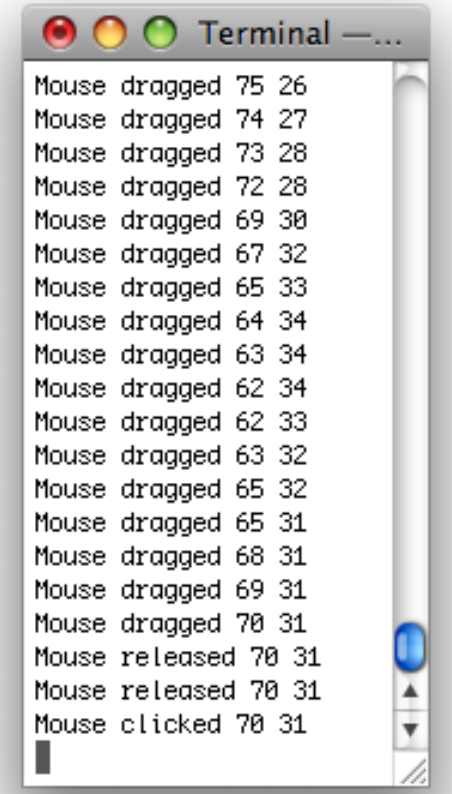
- All Swing components allow the following listeners to be registered
  - KeyListener, MouseListener, MouseMotionListener, MouseWheelListener, FocusListener
- Some Components allow other Listeners, commonly used ones are
  - ActionListener, ChangeListener, ListSelectionListener, WindowListener
- There are many others

Mouse Events

- Three listeners of interest
- MouseListener - mouse buttons
  - void mouseClicked (MouseEvent e)
  - void mouseEntered (MouseEvent e)
  - void mouseExited (MouseEvent e)
  - void mousePressed (MouseEvent e)
  - void mouseReleased (MouseEvent e)
- MouseMotionListener - mouse moved
  - void mouseDragged (MouseEvent e)
  - void mouseMoved (MouseEvent e)
- MouseWheelListener
  - void mouseWheelMoved (MouseWheelEvent e)

Mouse Events Example

```
public class EgMouseListener extends JFrame {
    public EgMouseListener() {
        addMouseListener(new MouseListener () {
            public void mouseClicked(MouseEvent e) {
                System.out.println ("Mouse clicked " +
                                    e.getX() + " " + e.getY());
            }
            public void mouseReleased(MouseEvent e) {
                System.out.println ("Mouse released " +
                                    e.getX() + " " + e.getY());
            }
            public void mouseEntered(MouseEvent e) {}
            public void mouseExited(MouseEvent e) {}
            public void mousePressed(MouseEvent e) {}
        });
        addMouseMotionListener(new MouseMotionListener(){
            public void mouseDragged(MouseEvent e) {
                System.out.println ("Mouse dragged " +
                                    e.getX() + " " + e.getY());
            }
            public void mouseMoved(MouseEvent e) {}
        });
        setVisible(true);
        setSize(500, 350); } }
```



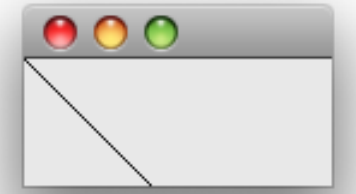
DRAWING

Drawing Shapes

- Every Swing component allows you to draw on it - extend it and override paint (Graphics g)
- Graphics allows you to draw lots of different shapes easily (circle, rectangle, arcs, ovals, polygons)

- Extend a JPanel and override paint method

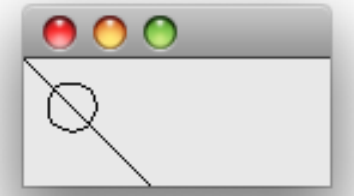
```
public void paint (Graphics g) {
  g.drawLine (0, 0, 75, 75);
}
```



- All Graphics objects in Swing are really Graphics2D objects
  - Graphics was the AWT object

Drawing

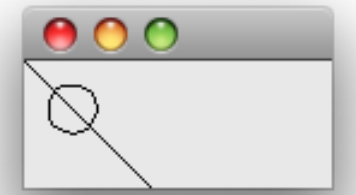
```
public class ExPanel extends JPanel {
    public void paint (Graphics g) {
        g.drawLine (0, 0, 75, 75);
        g.drawOval (10, 10, 20, 20);
    }
}
public class TestExPanel extends JFrame {
    public TestExPanel() {
        getContentPane().add(new ExPanel());
        setSize (75, 75);
        setVisible (true);
    }
    public static void main (String argv[]) {
        new TestExPanel();
    }
}
```



Graphics2D

- Part of the Java2D framework
- Has additional methods such as
  - draw (Shape s)
  - Where Shape is an interface implemented by
    - Area, CubicCurve2D, GeneralPath, Line2D, QuadCurve2D, Rectangle, RectangleShape, Ellipse2D
- And also other drawing primitives

Using Graphics2D & Shape



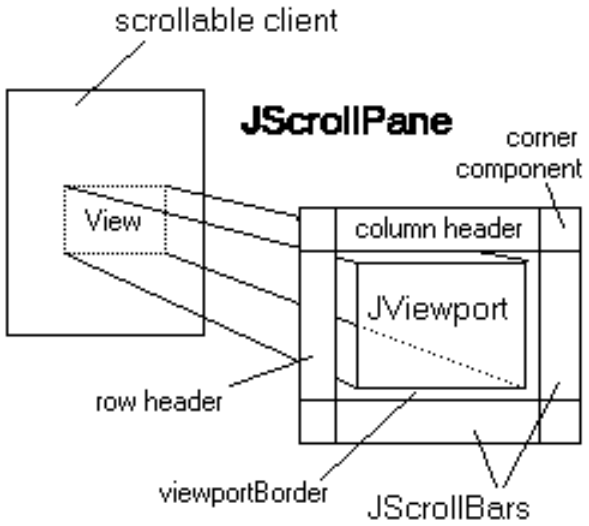
```
public class ExPanel extends JPanel {
    public void paint(Graphics g) {
        Graphics2D g2d = (Graphics2D) g;
        Line2D line = new Line2D.Double(0, 0, 75, 75);
        g2d.draw(line);
        Ellipse2D curve = new Ellipse2D.Double(10, 10, 20, 20);
        g2d.draw(curve);
    }
}
```

See more examples at https://docs.oracle.com/javase/tutorial/2d/geometry/primitives.html

SWING COMPONENTS

JScrollPane

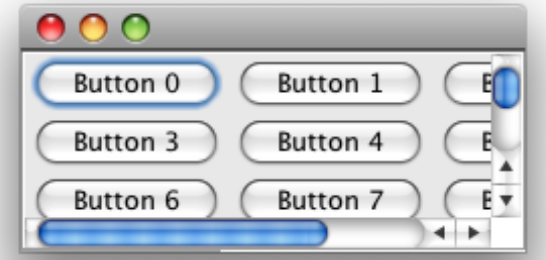
- Provides scrollable view of a component
- Use when space is limited or the component size changes



See
http://docs.oracle.com/javase/tutorial/uiswing/components/scrollpane.html

JScrollPane Example

```
public class TestScrollPane extends JFrame {
    public TestScrollPane() {
        GridButtonPanel gbp = new GridButtonPanel();
        JScrollPane sp = new JScrollPane(gbp);
        getContentPane().add(sp);
        setSize(75, 75);
        setVisible(true);
    }
    public static void main(String[] args) {
        new TestScrollPane();
    }
}
public class GridButtonPanel extends JPanel {
    public GridButtonPanel() {
        setLayout(new GridLayout(10, 3));
        for (int i = 0; i < 30; i++) {
            add(new JButton("Button " + i));
        }
        setVisible(true);
    }
}
```



Dialog

- Several ways to create dialogs
  - JOptionPane
    - Simple dialogs, standard layout
  - JDialog
    - Completely custom essentially same as JFrame
  - JColorChooser and JFileChooser

JOptionPane

- Number of static methods to create dialog boxes e.g.
  - showMessageDialog(parent, message, title, type)
  - showInputDialog(parent, message)
- Five message types

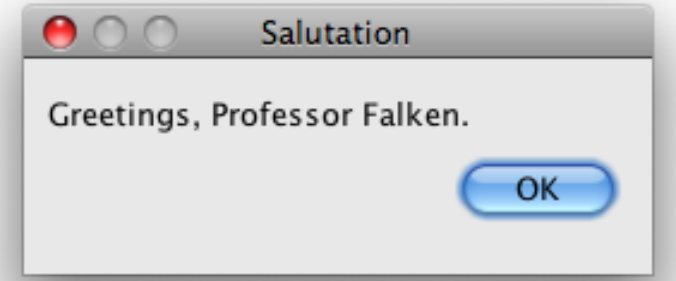

QUESTION_MESSAGE


INFORMATION_MESSAGE


WARNING_MESSAGE


ERROR_MESSAGE

- PLAIN_MESSAGE




```
JOptionPane.showMessageDialog(this,
  "Greetings, Professor Falken.", "Salutation",
  JOptionPane.PLAIN_MESSAGE);
```

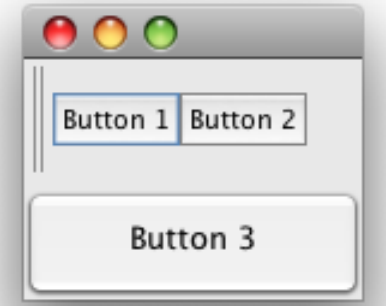
Creating Menus

- JMenuBar - attaches to top level JFrame (**`this`** in example below)
- JMenu - the actual menu - File, Edit etc.
- JMenuItem - selectable menu item - copy cut paste etc
  - Attach an ActionListener to receive *clicked* event

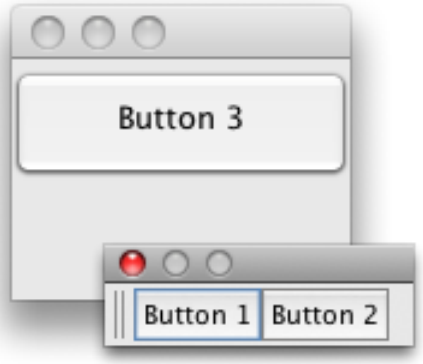
```
JMenuBar menu = new JMenuBar();
JMenu file = new JMenu("File");
JMenu edit = new JMenu("Edit");
JMenuItem load = new JMenuItem("Load");
file.add(load);
menu.add(file);
menu.add(edit);
load.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(null, "Not implemented ;-(");
    }
});
this.setJMenuBar(menu);
```

Creating Toolbars

- JToolBar
  - Provides a detachable toolbar
  - Can be either horizontal or vertical
- JToolBar is just another component

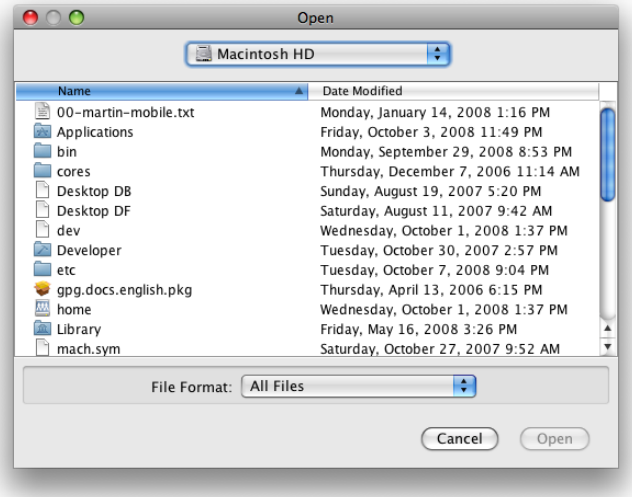



```
public class Test extends JFrame {
    public Test() {
        setLayout(new GridLayout(2, 1));
        JToolBar jtb = new JToolBar();
        getContentPane().add(jtb);
        jtb.add(new JButton("Button 1"));
        jtb.add(new JButton("Button 2"));
        getContentPane().add(new JButton("Button 3"));
        setSize(75, 75);
        setVisible(true);
    }
}
```



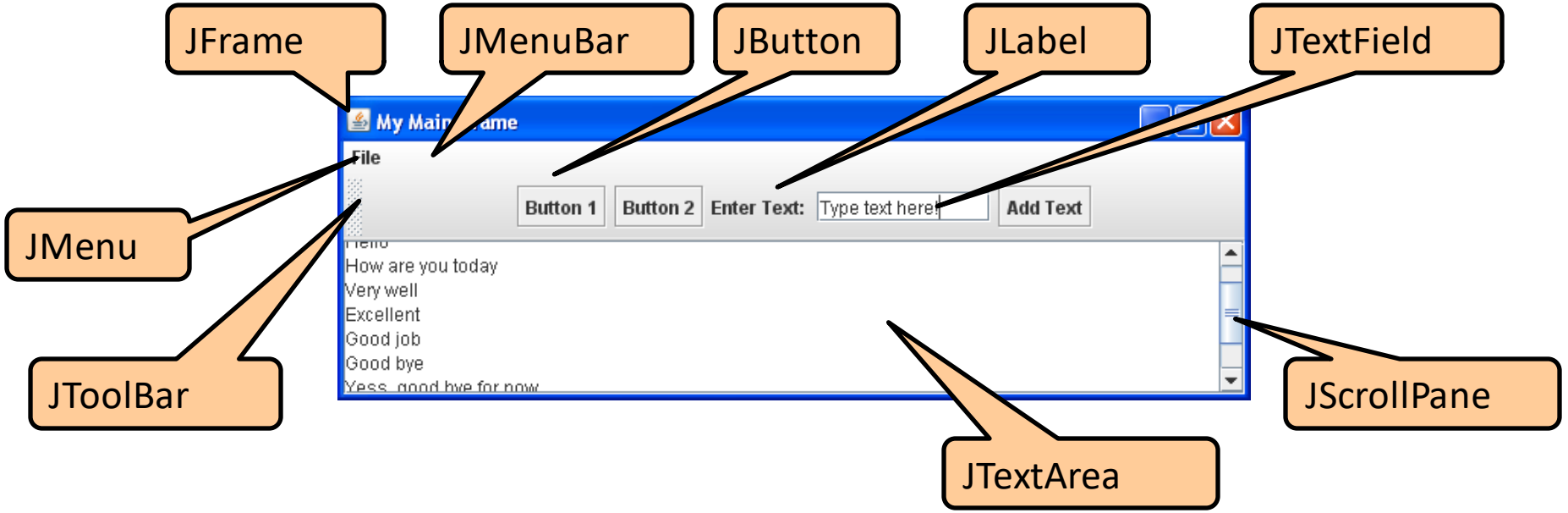
JFileChooser

- Dialog box for loading and saving file
  - Common dialogs
  - Filtering of filenames
  - Custom dialogs




```
JFileChooser fc = new JFileChooser();
int returnVal = fc.showOpenDialog(fc);
if (returnVal == JFileChooser.APPROVE_OPTION) {
    File file = fc.getSelectedFile();
    try {
        System.out.println("File is " + file.toString());
    } catch(Exception e) {}
} else {
  ...
}
```

Component Composition



- Components contain other components
  - JFrame – JMenuBar, JToolBar, JScrollPane
  - JMenuBar – JMenu
  - JToolBar – JButton, JLabel, JTextField
  - JScrollPane – JTextArea
  - JMenu – JMenuItem

Other Common Components

- JTextField - single line text entry
- JTextArea – multiple lines of text
- JPasswordField - single line text entry (invisible)
- JProgressBar - progress bar
- JTabbedPane - allows multiple tabs
- JPopupMenu - context menus
- JList - list
- JTable - table formatted data
- JTree - tree formatted data, expand/collapse

Many examples of these at https://studres.cs.st-andrews.ac.uk/CS5001/Examples/W07_GUIs/2_SwingComponentExamples/

LAYOUT MANAGERS

FlowLayout

BorderLayout

GridLayout

and many more!

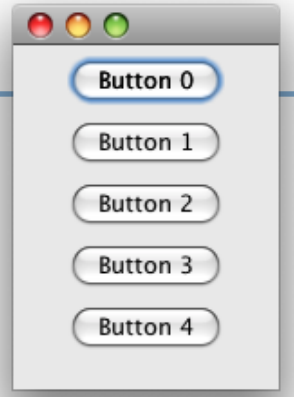
Layout managers

- Control how your GUI will look and behave
- **FlowLayout**
  - Components are added to the right and wrap around
- **BorderLayout**
  - Allows adding components to the north, south, east, west and center
- **GridLayout**
  - x by y grid, components added in order
- There are others
  - **GridBagLayout**, **GroupLayout**, …
- Examples on StudRes!

FlowLayout

- Components behave like a line of text being wrapped

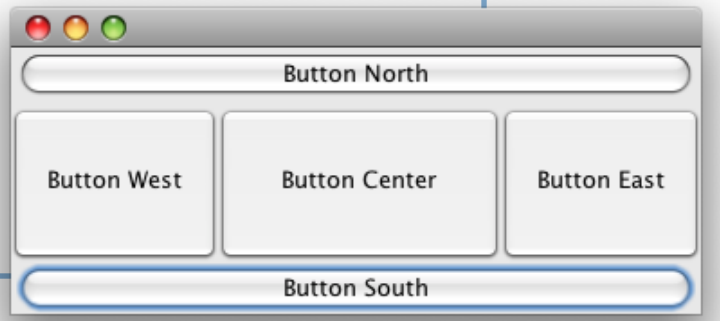
```
public class FlowExample extends JFrame {
    public FlowExample() {
        getContentPane().setLayout(new FlowLayout());
        for (int i = 0; i < 5; i++) {
            getContentPane().add(new JButton("Button " + i));
        }
        setVisible(true);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }
    public static void main(String[] args) {
        new FlowExample();
    }
}
```



BorderLayout

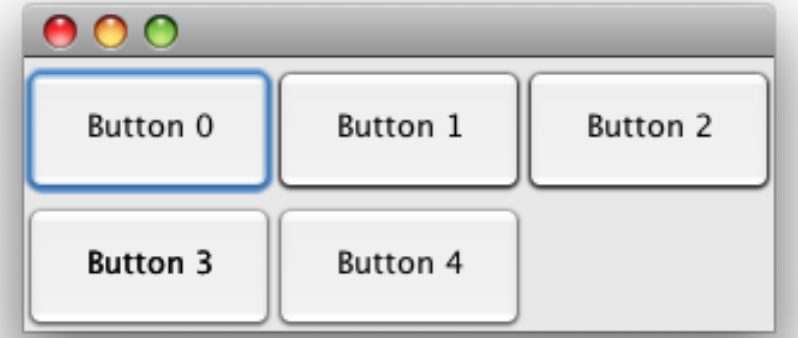
- Components align by north, south, east, west & center

```
public class BorderExample extends JFrame {
    public BorderExample() {
        Container cp = getContentPane();
        cp.setLayout(new BorderLayout());
        cp.add(new JButton("Button North"), BorderLayout.NORTH);
        cp.add(new JButton("Button South"), BorderLayout.SOUTH);
        cp.add(new JButton("Button East"), BorderLayout.EAST);
        cp.add(new JButton("Button West"), BorderLayout.WEST);
        cp.add(new JButton("Button Center"), BorderLayout.CENTER);
        setVisible(true);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

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



GridLayout

```
public class GridExample extends JFrame {
        public GridExample() {
                getContentPane().setLayout(new GridLayout(2,3));
                for (int i = 0; i < 5; i++) {
                        getContentPane().add(new JButton("Button " + i));
                }
                setVisible(true);
                setDefaultCloseOperation(EXIT_ON_CLOSE);
        }

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



What we've covered

- Application Design Patterns (for GUI driven Apps)
  - Model–View–Controller (MVC)
  - Model–Delegate (MD)
- GUI (View) Implementation
  - GUI Components (the building blocks of a GUI)
  - Component Composition (putting it all together)
- Examples

MVC Example

- Please find a simple MVC calculator on StudRes at

https://studres.cs.st-andrews.ac.uk/CS5001/Examples/W07_GUIs/4_MVCGuiExample/

No frogs are harmed while running this application!

Reading

- *Head First Design Patterns (Freeman and Freeman, Bates, Sierra*)
  - Library Classmark: QA76.76D47H4

- There are plenty of GUI component examples on the web, e.g.

http://docs.oracle.com/javase/tutorial/uiswing/examples/components/index.html