Java GUI (intro)

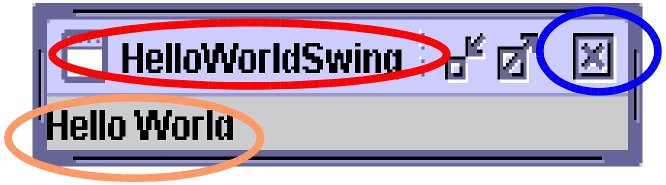
- JFC – Java Foundation Classes
  - encompass a group of features for building Graphical User Interfaces (GUI).
- `javax.swing.*` used for building GUIs.
- Some basic functionality is already there for you to reuse
  - ready made components, buttons, progress bars, text fields etc.
  - drag and drop support
  - internationalization
- There is no change in the way you code, compile and run.

Things to remember ...

- There is always a hierarchy of *components*
  - the hierarchy has a *root*
- Components can be added to the *containers*
  - you need to take care of how to place these components *layout*
- It's all events and actions from then on!
  - every user interaction with the graphical environment causes an event
  - you need to declare *listeners* to capture events and perform actions accordingly
- Finally *pack* and make *visible* your graphical environment.

Hello World!

```
import javax.swing.*;

public class HelloWorldSwing {

  private static void createAndShowGUI() {
    //Make sure we have nice window decorations
    JFrame.setDefaultLookAndFeelDecorated(true);

    //Create and set up the window.
    JFrame frame = new JFrame("HelloWorldSwing");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    //Add the ubiquitous "Hello World" label.
    JLabel label = new JLabel("Hello World");
    frame.getContentPane().add(label);

    //Display the window.
    frame.pack();
    frame.setVisible(true);
    }

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



Hierarchy of Components

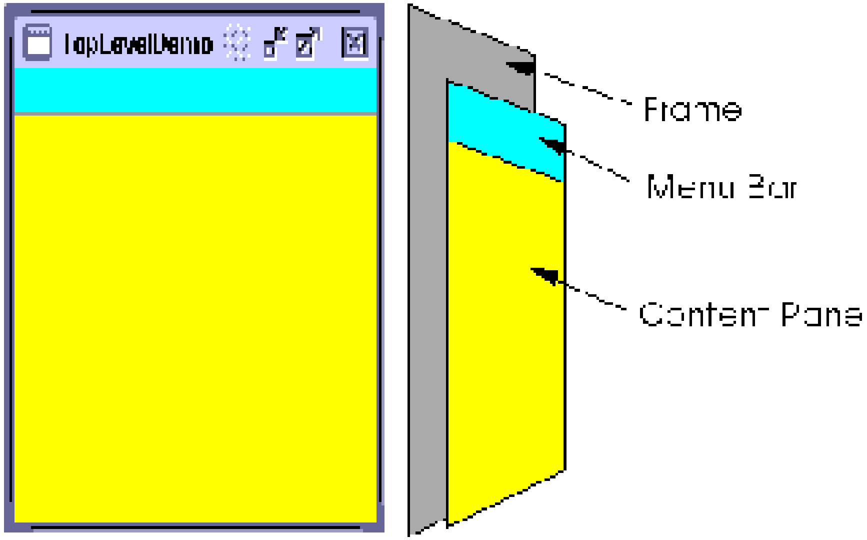
- Top level containers
  - Top of any Swing hierarchy e.g. Applet, Dailog, Frame
- General purpose containers
  - Intermediate containers for multiple uses e.g. tool bar, tabbed pane, scroll pane
- Special purpose containers
  - Intermediate containers with a special role e.g Intenal Frame, Layered Pane
- Basic controls
  - Atomic components get information from the user e.g. slider, list, combo box.

Hierarchy of Components

- Uneditable Information Displays
  - Atomic components solely used to display information to the user e.g. label, progress bar, tool tip
- Interactive Displays
  - Atomic components for displaying specialized formatted ionformation e.g. File Chooser, table, tree

Using Top Level Containers

- Every GUI component must be part of a containment hierarchy.
  - A containment hierarchy is a tree of components that has a top-level container as its root.
- Each GUI component can be contained only once.



- Each top-level container has a content pane that contains (directly or indirectly) the visible components in that top-level container's GUI.

As a rule, a standalone application with a Swing-based GUI has at least one containment hierarchy with a `JFrame` as its root

Adding Components to a Content Pane

- Create Frame
  - set title name and default close operation
- Create a menu bar
  - JMenuBar - set opaque if it is to be used as a pane
- Create a label
- Set menu Bar
  - specialized position
- Add label to content pane

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

public class TopLevelDemo {
  private static void createAndShowGUI() {
    JFrame.setDefaultLookAndFeelDecorated(true);
    JFrame frame = new JFrame("TopLevelDemo");
    frame.setDefaultCloseOperation(
                    JFrame.EXIT_ON_CLOSE);
    JMenuBar cyanMenuBar = new JMenuBar();
    cyanMenuBar.setOpaque(true);
    cyanMenuBar.setBackground(Color.cyan);
    cyanMenuBar.setPreferredSize(
             new Dimension(200, 20));
    JLabel yellowLabel = new JLabel();
    yellowLabel.setOpaque(true);
    yellowLabel.setBackground(Color.yellow);
    yellowLabel.setPreferredSize(
             new Dimension(200, 180));
    frame.setJMenuBar(cyanMenuBar);
    frame.getContentPane().add(yellowLabel,
                         BorderLayout.CENTER);
    frame.pack();
    frame.setVisible(true);
  }
  public static void main(String[] args) {
    createAndShowGUI();
  }
}
```

Adding to Content Panes

- A content pane is a `Container` which provides `add` methods which takes as arguments
  - some other component instance to be added to the container
  - coordinates (with restrictions) about its placement in the container.
- The layout of components inside a container can be customized. Java provides some policies
  - GridBagLayout, GridLayout, SpringLayout, BorderLayout, BoxLayout
  - Depending on your needs
    - full screen, space between components, re-sizable as the main window is being resized, etc

Using Layout Managers

- Layout Manager is an object which determines the size and position of components inside a containers.
- You need to worry about layout managers for
  - `JPanel` (default layout manager `FlowLayout`)
  - and content panes (default layout manager `BorderLayout`)
- Setting your layout manager

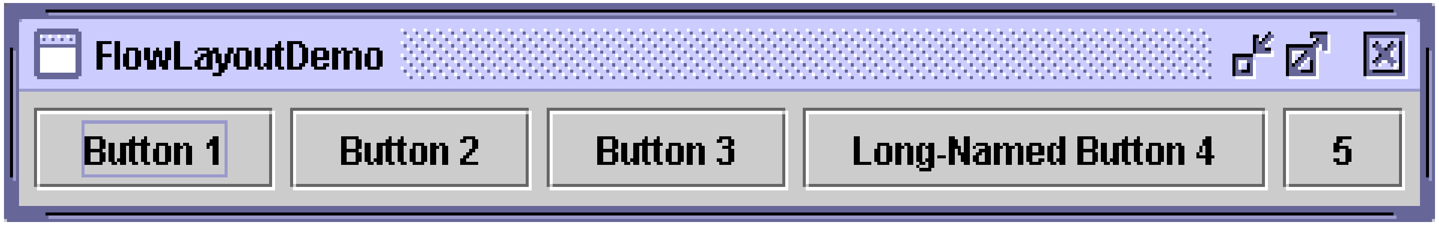
```
JPanel frame = new JPanel(new FlowLayout());
```

  - or after creation of a `JPanel`

```
Conainer contentPane = frame.getConContentPane();
   contentPane.setLayout(new BorderLayout());
```

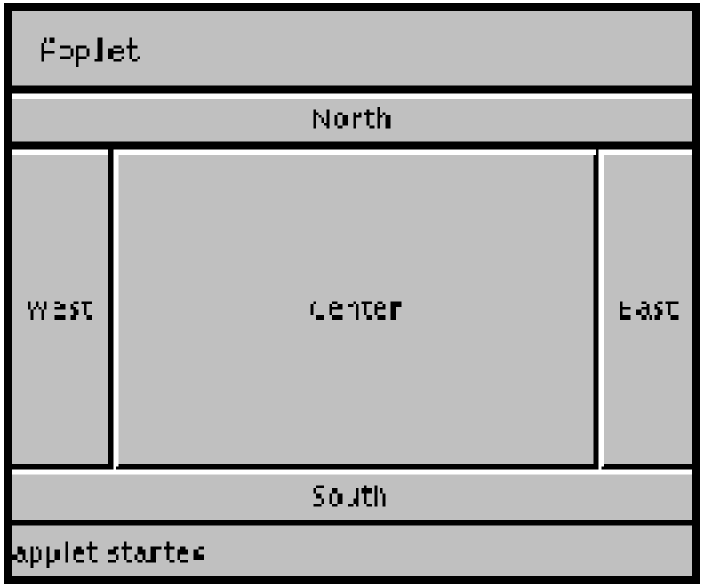
FlowLayout

- Adds components in a row one next to each other.
  - if the total width of the added components is longer than their containers width, successive components are placed in a new row.



Border Layout

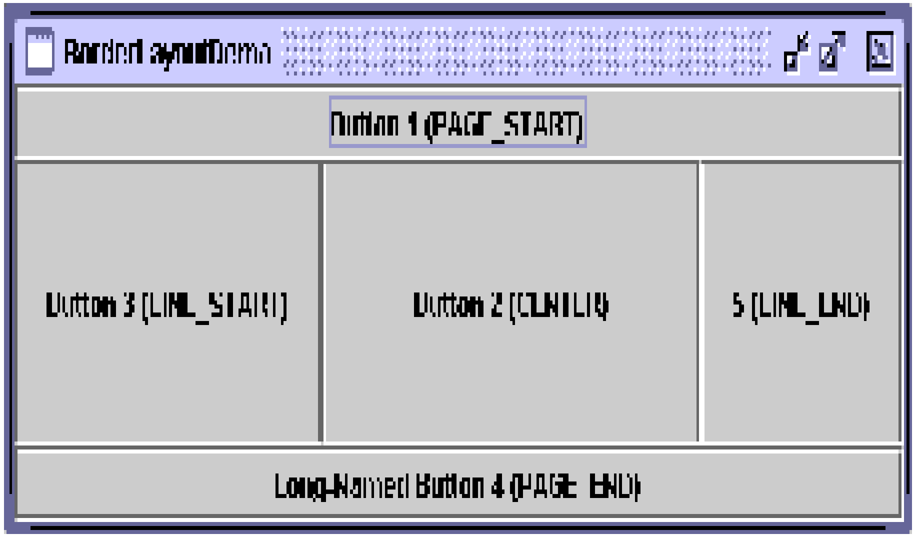
- Adds components filling in the whole window
  - allows positioning usign NORTH, WEST, SOUTH, EAST, CENTER



```
import java.awt.*;
import java.applet.Applet;

public class buttonDir extends Applet {
  public void init() {
    setLayout(new BorderLayout());
    add(new Button("North"), BorderLayout.NORTH);
    add(new Button("South"), BorderLayout.SOUTH);
    add(new Button("East"), BorderLayout.EAST);
    add(new Button("West"), BorderLayout.WEST);
    add(new Button("Center"), BorderLayout.CENTER);
  }
}
```

Border Layout

- Adds components filling in the whole window
  - PAGE_START, CENTER, LINE_START, LINE_END, PAGE_END



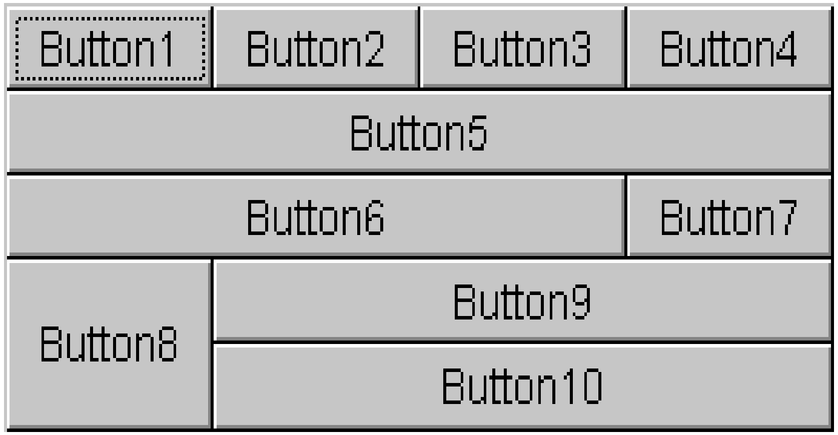
```
import java.awt.*;
import java.applet.Applet;

public class buttonDir extends Applet {
  public void init() {
    JButton button = new Jbutton(
            "Button 1 (PAGE_START)");
    pane.add(button, BorderLayout.PAGE_START);
    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);
   }
}
```

GridBagLayout

- Very flexible
  - allows components of different sizes
  - spanning ceveral rows and/or columns
- Every component has to be added with
  - an instance of `GridBagLayout`
  - an instance of `GridBagConstraints`
- Each `GridBagLayout` object maintains a dynamic, rectangular grid of cells, with each component occupying one or more cells, called its display area.

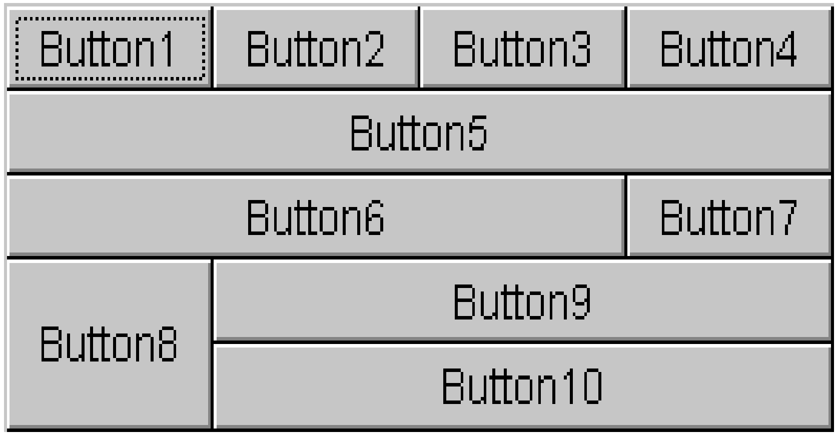
GridBagLayout (cont)

```
import java.awt.*;
import java.util.*;
import java.applet.Applet;
public class GridBagEx1 extends Applet {
  protected void makebutton(String name,
               GridBagLayout gridbag,
               GridBagConstraints c) {
    Button button = new Button(name);
    gridbag.setConstraints(button, c);
    add(button);
  }

  public static void main(String args[]) {
    Frame f = new Frame(
                "GridBag Layout Example");
    GridBagEx1 ex1 = new GridBagEx1();
    ex1.init();
    f.add("Center", ex1);
    f.pack();
    f.setSize(f.getPreferredSize());
    f.show();
    }
```



GridBagLayout (cont)

```java
public void init() {
  GridBagLayout gridbag = new GridBagLayout();
  GridBagConstraints c = new GridBagConstraints();
  setLayout(gridbag);
  c.fill = GridBagConstraints.BOTH;
  c.weightx = 1.0;
  makebutton("Button1", gridbag, c);
  makebutton("Button2", gridbag, c);
  makebutton("Button3", gridbag, c);
  c.gridwidth = GridBagConstraints.REMAINDER;
  makebutton("Button4", gridbag, c);
  c.weightx = 0.0;
  makebutton("Button5", gridbag, c);
  c.gridwidth = GridBagConstraints.RELATIVE;
  makebutton("Button6", gridbag, c);
  c.gridwidth = GridBagConstraints.REMAINDER;
  makebutton("Button7", gridbag, c);
  c.gridwidth = 1;
  c.gridheight = 2;
  c.weighty = 1.0;
  makebutton("Button8", gridbag, c);
  c.weighty = 0.0;
  c.gridwidth = GridBagConstraints.REMAINDER;
  c.gridheight = 1;
  makebutton("Button9", gridbag, c);
  makebutton("Button10", gridbag, c);
  setSize(300, 100);
  }

}
```



Giving life to your GUI components

- Every time the user performs an action on one of the GUI's component it (the action) creates an event
- Objects can be notified of events,
  - the object has to implement the appropriate interface
  - be registered as an event listener on the appropriate event source

User Actions	**Listener Type**
click a button, press Enter while typing	*ActionListener*
Close a window	*WindowListener*
press mouse button	*MouseListener*
moves mouse over	*MouseMotionListener*
keyboard focus	*FocusListener*

Implementing a Listener

- Two steps

(a) the declaration of the event handler class, implement the appropriate interface

(b) register an instance of the listener one (or more) components

```
import javax.swing.*;
import java.awt.Toolkit;
import java.awt.BorderLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class Beeper extends JPanel implements ActionListener {
    JButton button;
    public Beeper() {
        super(new BorderLayout());
        button = new JButton("Click Me");
        button.setPreferredSize(new Dimension(200, 80));
        add(button, BorderLayout.CENTER);
        button.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e) {
        Toolkit.getDefaultToolkit().beep();
    }
```

Multiple Listeners

- You can add more than one Listeners to componets
  - e.g. to a `JButton`, check with the API first
  - one event can cause multiple actions

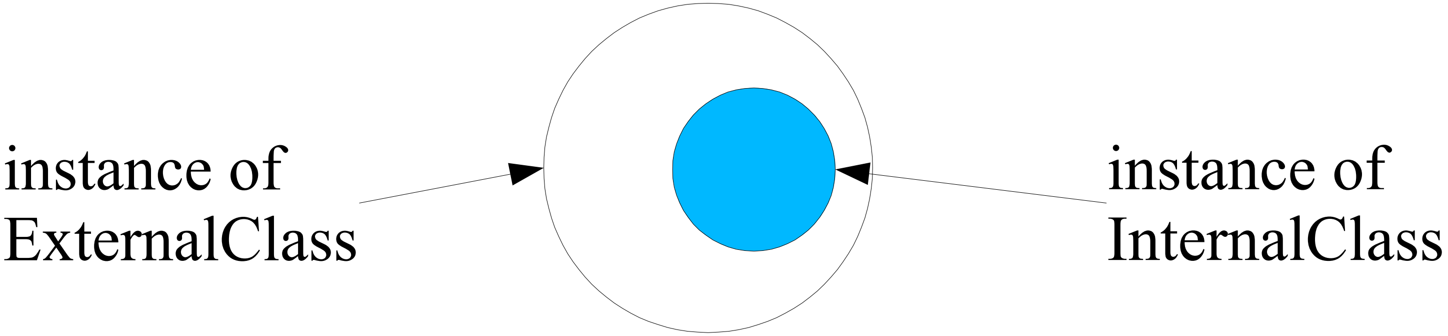
```
public class MultiListener ... implements ActionListener {
    ...
    //where initialization occurs:
        button1.addActionListener(this);
        button2.addActionListener(this);
        button2.addActionListener(new Eavesdropper(bottomTextArea));
    }
    public void actionPerformed(ActionEvent e) {
        topTextArea.append(e.getActionCommand() + newline);
    }
}
class Eavesdropper implements ActionListener {
    ...
    public void actionPerformed(ActionEvent e) {
        myTextArea.append(e.getActionCommand() + newline);
    }
}
```

Inner Classes

- Java allows the definition of a class inside another class

```
public class ExternalClass {
...
  class InternalClass {
  ...
  }
...
}
```

- `InternalClass` exists inside an instance of an `ExternalClass`. `InternalClass` has direct access to members of `ExternalClass`



Using inner classes

- Recall the Stack example
  - we can enumeration capabilities to Stack

```
public class Stack {
    private Vector items;
    ...//code for Stack's methods and constructors not shown...
    public Enumeration enumerator() {
        return new StackEnum();
    }
    class StackEnum implements Enumeration {
        int currentItem = items.size() - 1;
        public boolean hasMoreElements() {
            return (currentItem >= 0);
        }
        public Object nextElement() {
            if (!hasMoreElements())
                throw new NoSuchElementException();
            else
                return items.elementAt(currentItem--);
        }
    }
}
```

Anonymous inner classes

- Inline class definition without giving a name to the class

```
public class SomeGUI extends JFrame{
  //button member declarations ...
  protected void buildGUI(){
   button1 = new JButton();
   button2 = new JButton();
   ...
   button1.addActionListener(
     new java.awt.event.ActionListener(){
      public void actionPerformed(java.awt.event.ActionEvent e){
        // do something
      }
     }
   );
  button2.addActionListener(
     new java.awt.event.ActionListener(){
      public void actionPerformed(java.awt.event.ActionEvent e){
        // do something else
      }
     }
   );
}
```

Compare

```
public class SomeGUI extends JFrame{
  //button member declarations ...
  protected void buildGUI(){
   button1 = new JButton();
   button2 = new JButton();
   ...
   button1.addActionListener(
     new java.awt.event.ActionListener(){
      public void actionPerformed
(java.awt.event.ActionEvent e){
         // do something
      }
     }
   );
  button2.addActionListener(
     new java.awt.event.ActionListener(){
      public void actionPerformed
(java.awt.event.ActionEvent e){
         // do something else
      }
     }
   );
}
```

```
public class SomeGUI extends JFrame{
  //button member declarations ...
  protected void buildGUI(){
   button1 = new JButton();
   button2 = new JButton();
   ...
  class Button1Action implements ActionListener{
   public void actionPerformed(ActionEvent e){
   // do something
   }
  class Button2Action implements ActionListener{
   public void actionPerformed(ActionEvent e){
   // do something else
   }

 button1.addActionListener(new Button1Action());

 button2.addActionListener(new Button2Action());

}
```

Reasons for inner classes

- More readable code
  - all information for how to handle the event is located in one file
  - for a novice this might be difficult to parse at first
- Better encapsulation
  - the inner class can be declared private and thus only accessible to its enclosing class
  - e.g. the connection to a database server can be captured as an inner class limiting the classes that can directly connect to the database, enforcing the connection protocol
  - one point of control!