Java GUI. Part II

Swing components

Once you have a JFrame -

you can add Swing **JComponent**s to its pane (***getContentPane()***):

J

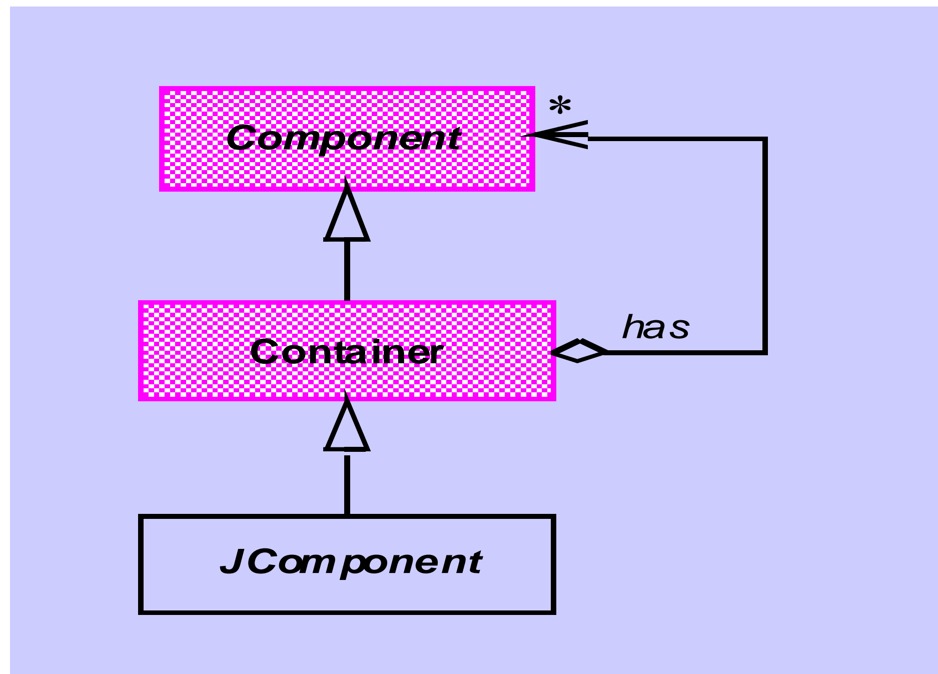
- Button
- RadioButton
- CheckBox
- Label
- List
- ScrollPane
- Slider
- TextArea
- TextField
- Table

Components can be nested

- Virtually all JComponents are capable to hold inside other components, you can stick just anything into anything else
- Most of the time we add **interactive components** (Buttons, Menus, Text boxes) into a **background component** (Panels, Scrollable panels)
- But even a JPanel can be used as an interactive component

Container

Container is a Component that can contain other components and containers.



Swing Component Hierarchy

JComponent

JLabel

JComboBox

JScrollbar

JList

JPanel

JMenuBar

JTextComponent

JTextArea

JTextfield

AbstractButton

JButton

JMenuItem

JMenu

JToggleButton

JCheckbox

JRadioButton

Intermediate containers

Used to organize and position other components.

- **JPanel** used for collecting other components.
- **JScrollPane** provides view with scroll bars.
- **JSplitPane** divides two components graphically.
- **JTabbedPane** lets the user switch between a group of components by clicking on a labeled tab.

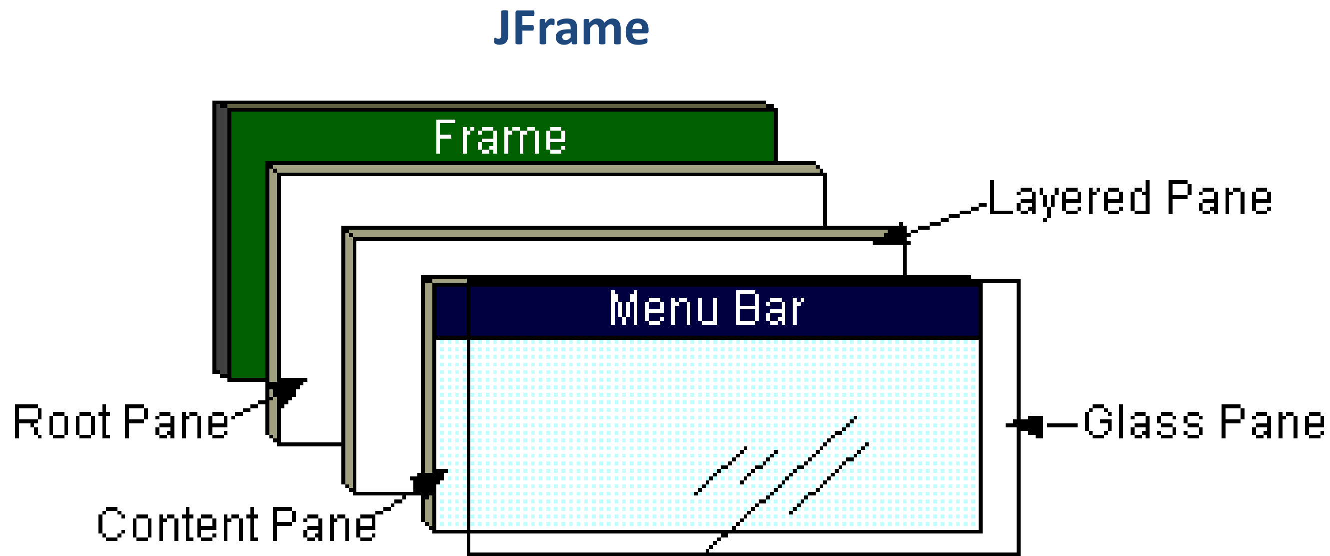
JFrame

is a window with title, border, (optional) menu bar and user-specified components.

It can be moved, resized, iconified.

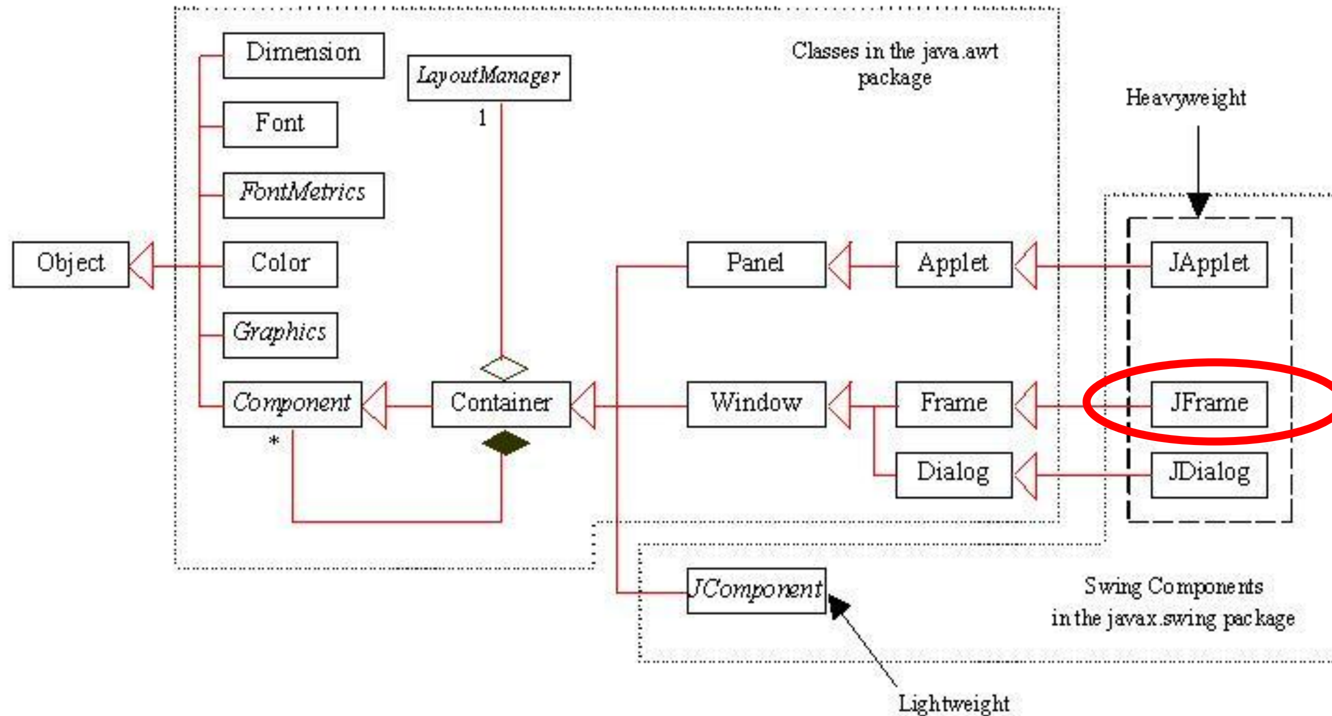
JFrame internal structure

A Swing Frame has a fairly complicated structure, with several panes. Some of these are used to implement pluggable look-and-feel.



JFrame

is not a subclass of ***JComponent***



JFrame

- JFrame delegates responsibility of managing user-specified components to a ***content pane***, an instance of JPanel.
- To add a component to a *JFrame*, add it to its content pane:

```
JFrame f = new JFrame("A Frame");
JButton b = new JButton("Press");
Container cp = f.getContentPane();
cp.add(b)
```

Heavyweight and lightweight components

- *Heavyweight* components
  - Instances of classes *JApplet*, *JDialog*, *JFrame*, and *JWindow*.
  - Created by association to a *native* GUI component part of the native windowing system.
  - Their look and feel depends on the native GUI component.

- *Lightweight* components
  - Any other Swing component.
  - They are completely implemented in Java.

Sequential/Concurrent programming (1/2)

- A *thread* is a sequence of instructions being executed by the processor.

- *Sequential programming*: So far programs consisted of a single thread, which executes the sequence of actions in the **main** method (*main thread*).

- *Concurrent programming*: A program can contain several threads each executing independent sequences of actions.

Sequential/Concurrent programming (2/2)

- *Event-dispatching thread*: executes all the code that involves repainting components and handling events.

- After the *JFrame* has been made visible, the ***main*** thread should not perform actions that affect or depend on the state of the user interface.

LayoutManager

- Responsible for positioning and sizing components added to a container.
- Each container is associated with a *LayoutManager*.
- Setting and accessing *Container*'s layout manager:

```
public LayoutManager getLayout();
public void setLayout (LayoutManager manager);
```

LayoutManager classes (1/2)

- *FlowLayout* lays out components left to right, top to bottom.
- *BorderLayout* lays out up to five components, positioned “north,” “south,” “east,” “west,” and “center.”
- *GridLayout* lays out components in a two-dimensional grid.
- *CardLayout* displays components one at a time from a preset deck of components.

LayoutManager classes (2/2)

- *GridBagLayout* lays out components vertically and horizontally according to a specified set of constraints.
- *BoxLayout* lays out components in either a single horizontal row or single vertical column.
- *OverlayLayout* components are laid out on top of each other.

Understanding Layout manager policies

- Each background container may have its own layout manager
- To avoid frustration, it is useful to understand how each layout manager follows its own policy on determining the position and the size of components it contains

Example: how the layout manager decides

A layout scenario: make a panel and add 3 buttons to it

- The panel’s layout manager asks each button how big it prefers to be: **getPreferredSize()**
- The layout manager of JPanel uses its policies to decide whether it should respect all, part, or none of its buttons’ preferences

- Add the panel to JFrame:
- JFrame’s layout manager asks the panel about its preferred size, and then decides according to its layout manager whether to respect the panel’s preferences or ignore them.

Three main layout managers

- Border
- Flow
- Box

Border layout

- Divides a background container into 5 regions
- You can add only one component per region to a background controlled by BorderLayout manager
- Components don’t get to have their preferred size

JFrame’s content pane default layout manager: ***BorderLayout***.

Flow layout

- Each component is the size it wants to be
- The components are laid out left-to-right in the order they are added
- When the next component would not fit horizontally, it drops to the next “line”

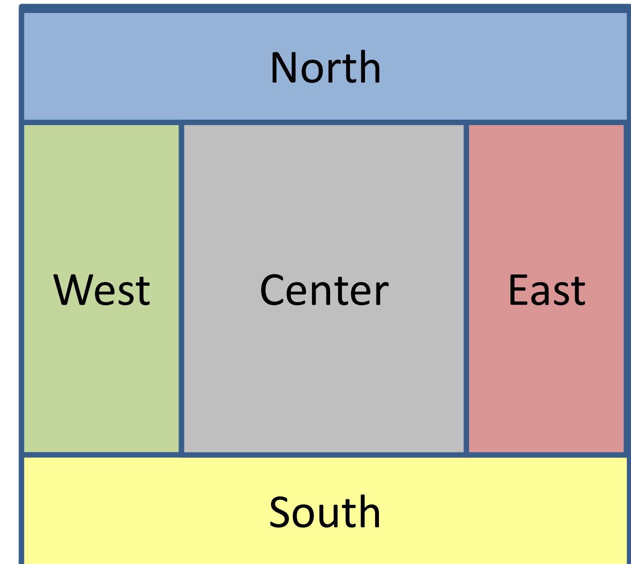
JPanel’s default layout manager: ***FlowLayout***.

Box layout

- Each component gets to have its preferred size
- The components are stack vertically (or horizontally) one above the other
- Each new component is forced to start a new "line"

Border layout

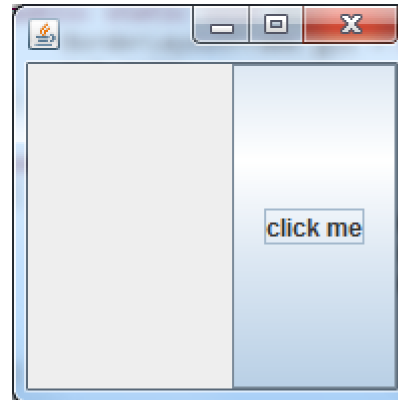
The background container is divided into 5 regions




```
JFrame frame = new JFrame();
JButton button = new JButton("click me");
frame.getContentPane().add(BorderLayout.EAST, button);
frame.setSize(200,200);
frame.setVisible(true);
```

Border layout example 1

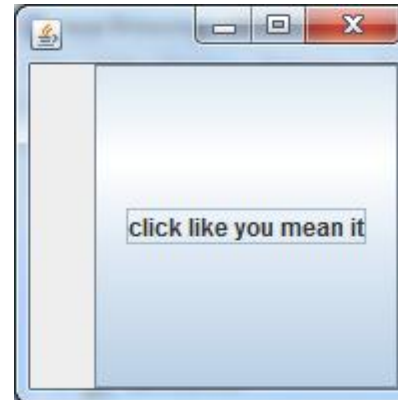
```
JFrame frame = new JFrame();
JButton button = new JButton(“click me”);
frame.getContentPane().add(BorderLayout.EAST, button);
frame.setSize(200,200);
frame.setVisible(true);
```



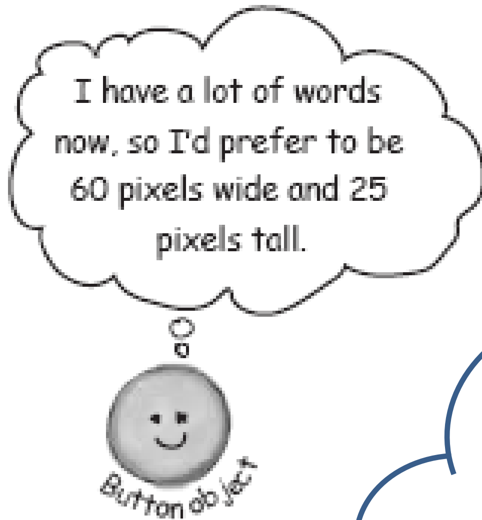
How the layout manager come up with these dimensions for the button?

Border layout example 2

```
JFrame frame = new JFrame();
JButton button = new JButton("click like you mean it");
frame.getContentPane().add(BorderLayout.EAST, button);
frame.setSize(200,200);
frame.setVisible(true);
```



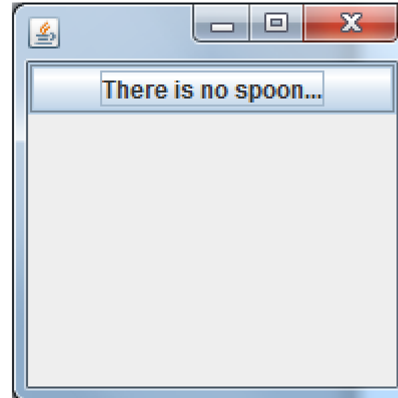
Border layout policy



Border layout manager

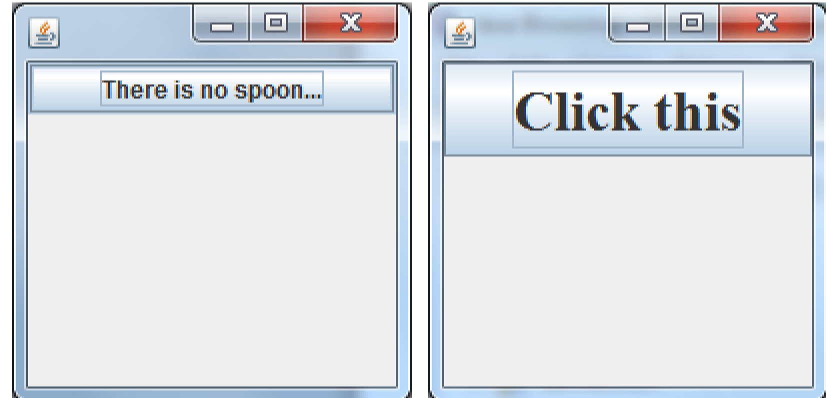
Border layout example 3

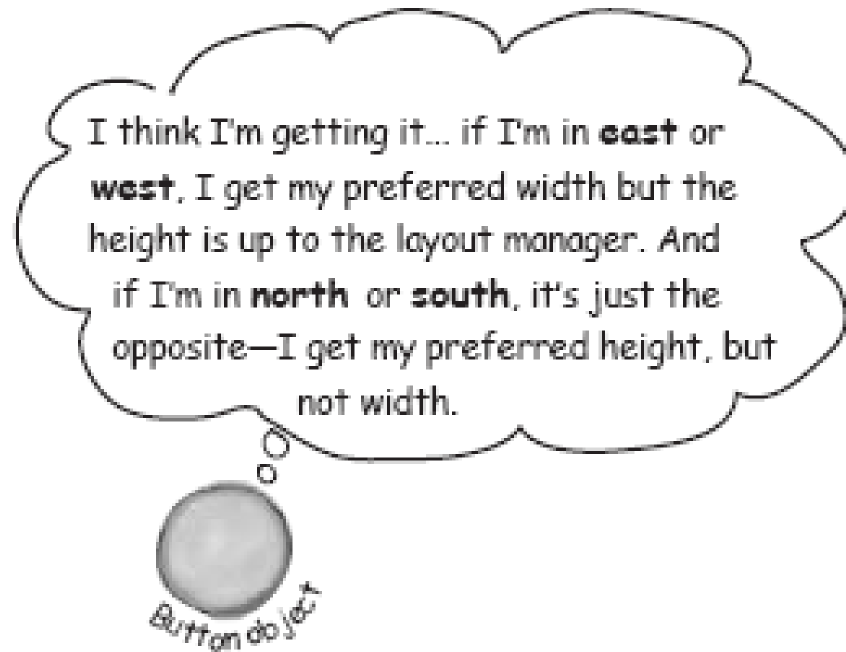
```
JFrame frame = new JFrame();
JButton button = new JButton("There is no spoon...");
frame.getContentPane().add(BorderLayout.NORTH, button);
frame.setSize(200,200);
frame.setVisible(true);
```



Border layout example 4

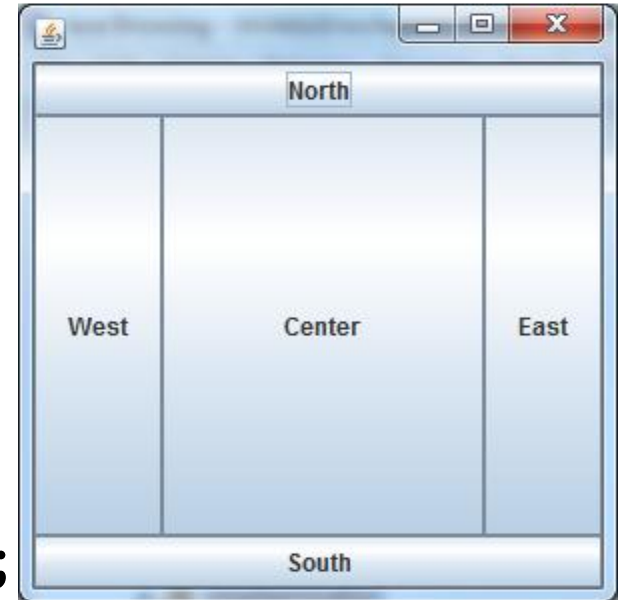
```
JFrame frame = new JFrame();
JButton button = new JButton("Click This!");
Font bigFont = new Font("serif", Font.BOLD, 28);
button.setFont(bigFont);
frame.getContentPane().add(BorderLayout.NORTH, button);
frame.setSize(200,200);
frame.setVisible(true);
```





the center gets whatever is left

Border layout summary example

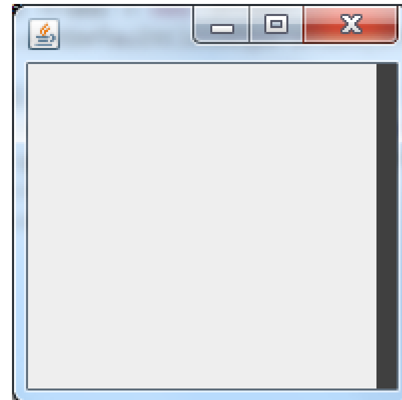



```
JFrame frame = new JFrame();
JButton east = new JButton("East");
JButton west = new JButton("West");
JButton north = new JButton("North");
JButton south = new JButton("South");
JButton center = new JButton("Center");
frame.getContentPane().add(BorderLayout.EAST, east);
frame.getContentPane().add(BorderLayout.WEST, west);
frame.getContentPane().add(BorderLayout.NORTH, north);
frame.getContentPane().add(BorderLayout.SOUTH, south);
frame.getContentPane().add(BorderLayout.CENTER, center);
frame.setSize(300,300);
frame.setVisible(true);
```

Flow layout example: panel with 2 buttons (1/3)

Adding a panel to the East region of JFrame

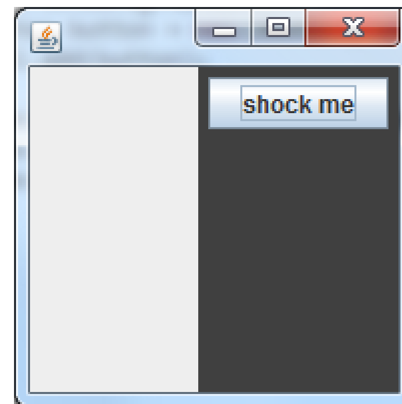
```
JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
frame.getContentPane().add(BorderLayout.EAST, panel);
frame.setSize(200,200);
frame.setVisible(true);
```



Flow layout example: panel with 2 buttons (2/3)

Adding a button to the panel

```
JButton button = new JButton(“shock me”);
panel.add(button);
frame.getContentPane().add(BorderLayout.EAST, panel);
```



Flow layout example: panel with 2 buttons (3/3)

Adding **two** buttons to the panel

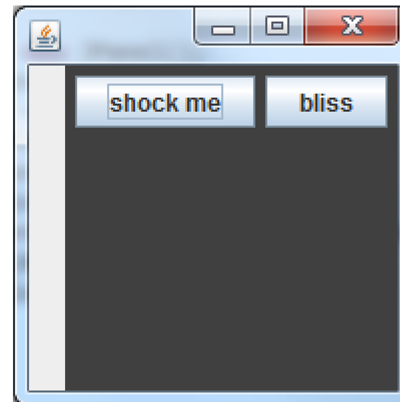
```
JButton button = new JButton("shock me");
panel.add(button);
JButton buttonTwo = new JButton("bliss");
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.EAST, panel);
```

What do you expect to happen?

Flow layout example: panel with 2 buttons (3/3)

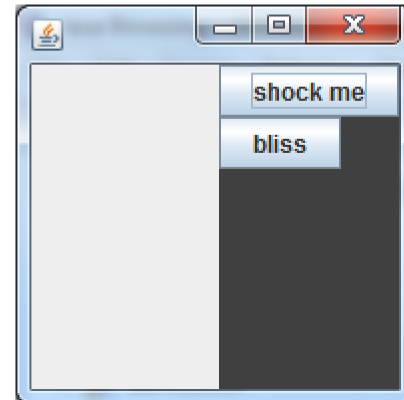
Adding two buttons to the panel

```
JButton button = new JButton("shock me");
panel.add(button);
JButton buttonTwo = new JButton("bliss");
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.EAST, panel);
```

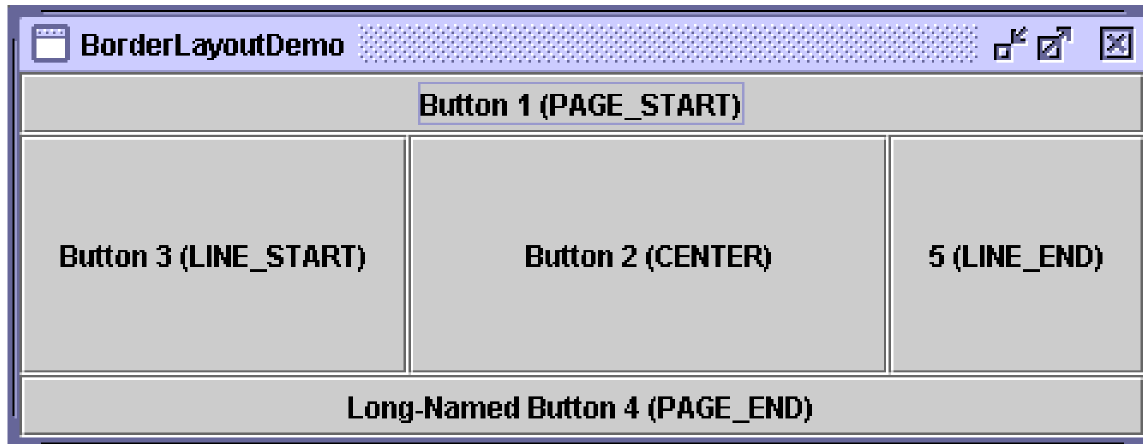


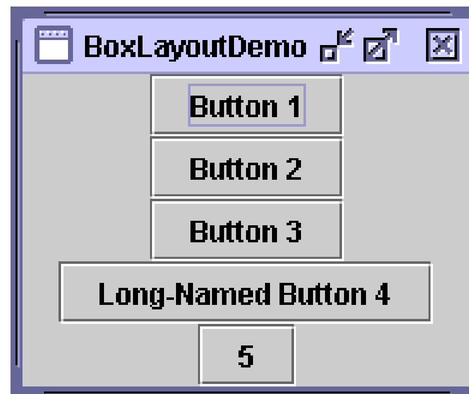
Box layout example: panel with two buttons

```
JPanel panel = new JPanel();
panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
JButton button = new JButton("shock me");
panel.add(button);
JButton buttonTwo = new JButton("bliss");
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.EAST, panel);
```



Component layout summary





Event-driven programs

Two main interaction patterns between the program and the environment:

- **Algorithm-driven**: the program determines what information it needs and when to get it (text-based interfaces). Active application

- **Event-driven**: the application waits for something to happen in the environment - it waits for an ***event***, responds to this event and then waits for the next event (Graphical User Interfaces). Passive application.

Events and components

- Events are objects.
- Events: subclasses of abstract class *java.awt.AWTEvent*.
- Components ***generate*** events.
- An event object knows event source and other relevant information about the event.
- Given an event, to query for its component's source:

```
public Object getSource();
```

Listener or Event handler

- ***Listener***: An object interested in *being notified* when an event occurs in a given component.
- A Listener object **registers** with a component to be notified of events generated by it.
- Listener must implement the ***event listener interface*** associated with events for which it registered.
- Programming a handler for an event consists of implementing the interface associated with the event type.

General approach to GUI design

Program an application that displays a button. When the button is pressed, its foreground and background colors are swapped.

- Design: extend the class *JFrame* with *OnOffSwitch*, and its constructor builds the frame containing the button.

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
class OnOffSwitch extends JFrame {

  public OnOffSwitch () {
       super("On/Off Switch"); // frame title
       JButton button = new JButton("On/Off");
       button.setForeground(Color.black);
       button.setBackground(Color.white);
       this.getContentPane().add(button,
                             BorderLayout.CENTER);
   }
}//end of OnOffSwitch

public class OnOffTest {
       public static void main (String[] args) {
              OnOffSwitch frame = new OnOffSwitch();
              frame.setSize(300,200);
              frame.setVisible(true);
       }
}
```

Program does not work

- Pressing the button has no effect at all.

- When the button is pressed, it generates an *ActionEvent*.

- We need to program the response to that event.

Programming an ActionListener for JButton

- Implement a listener to handle event generated by JButton instance.

- If user presses button, it generates an *ActionEvent*.

- To do:
  - Define a class, *Switcher*, that implements **ActionEvent**.
  - Register an instance of *Switcher* with the **JButton** instance.

Revision of *OnOffSwitch* to create a *Switcher* listener and register it with *JButton*

```
public OnOffSwitch () {
	super("On/Off Switch"); // frame title

	// create button and set its colors
	JButton button = new JButton("On/Off");
	button.setForeground(Color.black);
	button.setBackground(Color.white);

	// create and register button’s listener:
	button.addActionListener(new Switcher());

	// add button to JFrame’s content pane:
	this.getContentPane().add(
		button, BorderLayout.CENTER);
}
```

Switcher ActionListener

```
class Switcher implements ActionListener {

      public void actionPerformed (ActionEvent e) {
         Component source = (Component)e.getSource();
         Color oldForeground = source.getForeground();
         source.setForeground(source.getBackground());
         source.setBackground(oldForeground);
      }
}
```

JFrame close event

- To terminate the program need to program a window listener to close the window.
- A window listener must implement the 7 methods in ***WindowListener*** interface.
- We only want to implement 2 of those methods:

  void windowClosed (WindowEvent e)

  void windowClosing (WindowEvent e)

Adapter classes: WindowAdapter

- Java provides a collection of abstract *event adapter classes*.
- These adapter classes implement listener interfaces with empty, do-nothing methods.
- To implement a listener class, we extend an adapter class and override **only** methods needed.

Terminator class

```
//implements window events to close a window
class Terminator extends WindowAdapter {

      public void windowClosing(WindowEvent e) {
            Window w = e.getWindow();
            w.dispose();
      }

      public void windowClosed(WindowEvent e) {
            System.exit(0);
      }
}
```

Create a new instance of Terminator and register with JFrame

```
public OnOffSwitch () {
        super("On/Off Switch"); // frame title
        this.addWindowListener(new Terminator());

        // create button and set its colors
        JButton button = new JButton("On/Off");
        button.setForeground(Color.black);
        button.setBackground(Color.white);

        // create and register button’s listener:
        button.addActionListener(new Switcher());

        // add button to JFrame’s content pane:
        this.getContentPane().add(
                button, BorderLayout.CENTER);
}
```

Reminder: Timer action listener

```
   BouncingBallAnimationListener(SimpleShape shape,
                                   int step, JFrame window ) {}

ic void actionPerformed(ActionEvent e) {
   if(shape.getX()+shape.getWidth()+this.step >= this.maxBoundX)
      signX=-1; //reached the end of X axis, reverse X direction
   else if (shape.getX() -this.step<=0)
      signX=1; //reached the beginning of X axis, reverse X direct
   if(shape.getY()+shape.getHeight()+this.step >= this.maxBoundY)
      signY=-1; //reached the end of Y axis, reverse Y direction
   else if (shape.getY() -this.step<=0)
      signY=1; //reached the beginning of Y axis, reverse Y direct

   shape.changeLocation(signX*step, signY*step);

   window.repaint();
```

Inner classes for action listeners have access to the members of the outer class

```
public class StartAndStopButton extends JFrame {
    Timer timer;
    boolean animated=false;
    JButton button;

    class StartStopActionListener implements ActionListener{
        public void actionPerformed(ActionEvent e){
            if(animated){
                timer.stop();
                button.setText("Start");
                animated=false;
            }
            else{
                timer.start();
                button.setText("Stop");
                animated=true;
            }
        }
    }
}
```

Design choice for action listeners

- If they are inner classes, you cannot reuse them without creating an instance of an outer class

```
class Foo {
      public static void main (String[] args) {
            MyOuter outerObj = new MyOuter();
            MyOuter.MyInner innerObj
                  = outerObj.new MyInner();
      }
}
```

- If they are standalone classes, you may need to pass to them references to every class member which needs to be affected by an action

Basic GUI programming - summary

- To a JFrame instance
  - Add components comprising the interface
  - Use Layout Managers to position components on the screen
  - Program a WindowListener class to perform actions on window closing event.

- For every GUI component that generates events for which your application needs to react to:
  - Define a class that implements the Listener interface for desired events.
  - Instantiate and register Listener class with the component that generates desired events.

Building GUIs

- Use JPanel as a decomposition tool for complex views.
  - A standard technique.
  - Provides more flexibility;
  - *JPanel* can be added to other structures to expand or modify application.

- Build app view on a JPanel and add to a JFrame content pane.
- Good practice: replace default window content pane with the top-level Jpanel:

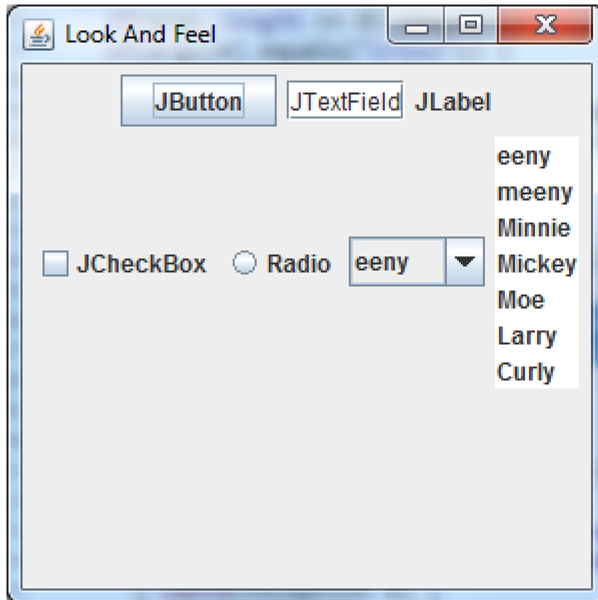
```
myFrame.setContentPane(myPanel);
```

Look and feel

- “Pluggable Look & Feel” allows your program to emulate the look and feel of various operating environments.
- You can even dynamically change the look and feel while the program is executing.
- Usually selection of one of two things:
  - the “cross platform” (Swing’s “metal”),
  - look and feel for the system you are currently on
- You must execute the LookAndFeel setup *before* you create any visual components

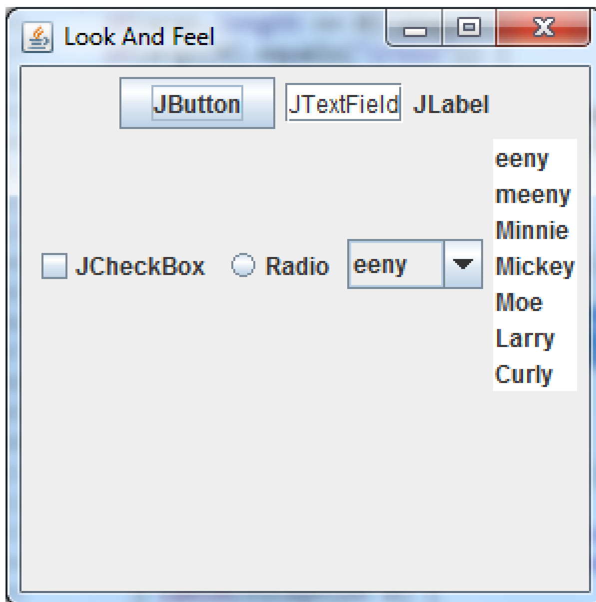
Look and feel examples (1/3)

```
UIManager.setLookAndFeel(UIManager.
      getCrossPlatformLookAndFeelClassName());
```

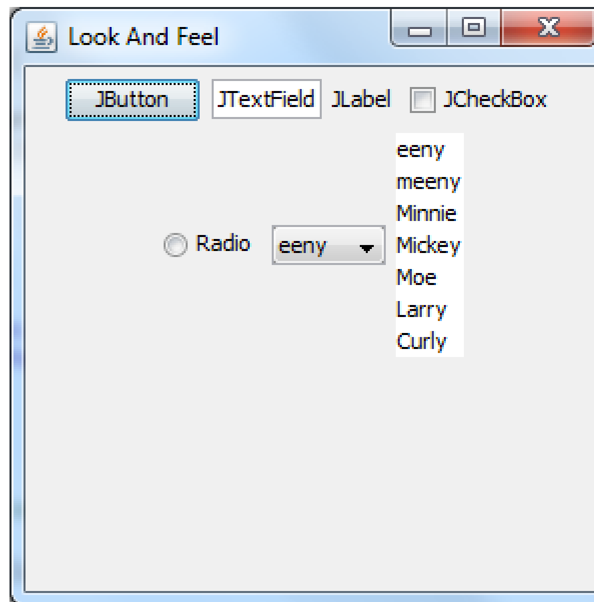


Look and feel examples (2/3)

```
UIManager.setLookAndFeel(UIManager.
                getSystemLookAndFeelClassName());
```



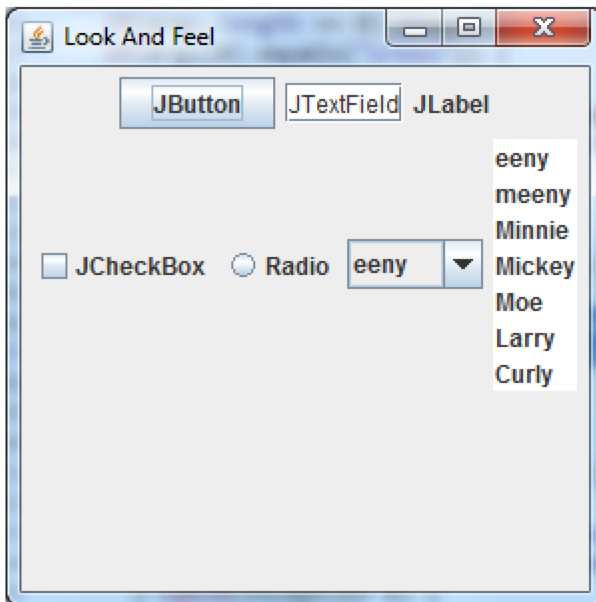
Cross-platform



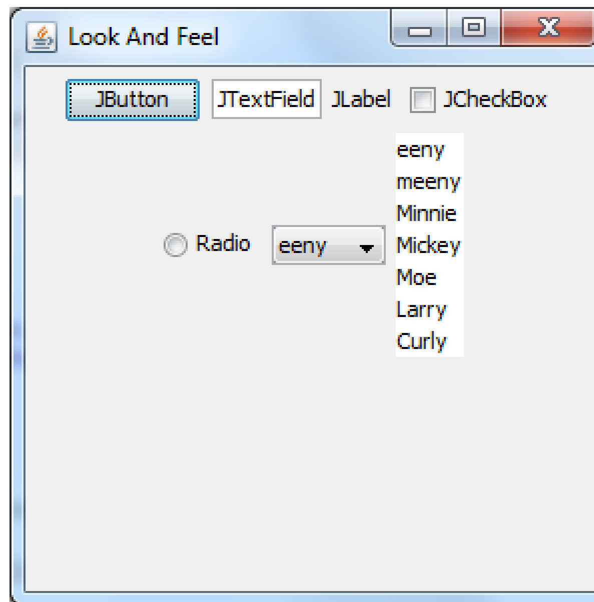
System (Windows)

Look and feel examples (3/3)

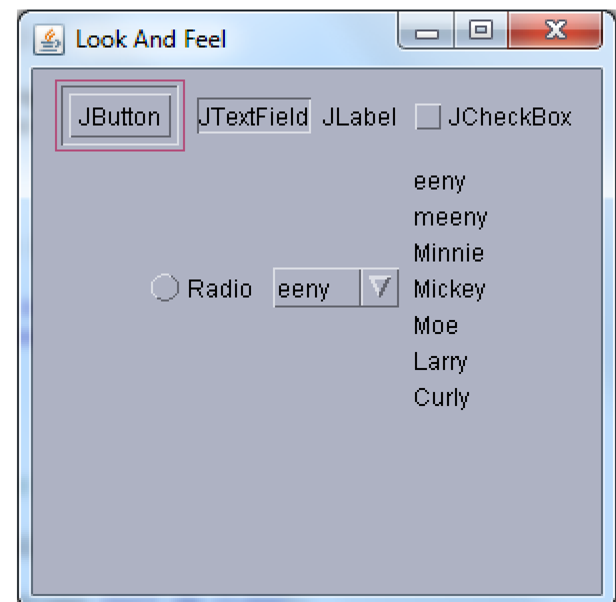
```
UIManager.setLookAndFeel("com.sun.java."+
        "swing.plaf.motif.MotifLookAndFeel");
```



Cross-platform



System (Windows 7)



Motif

Building GUIs

- Components can have borders to give them desired looks.
- The *JComponent* method adds a border to a component:

```
public void setBorder (Border border)
```

- Standard borders are obtained from the class *javax.swing.BorderFactory*.

MenuBar and Menu

- A *menu* offers options to user.
- Menus are not generally added to user interface.
- Menu usually appears either in a *menu bar* or as a *popup menu*.
- A *JFrame* often has a menu bar containing many menus; and each menu can contain many choices.

MenuBar and Menu

- Menu bar can be added to a *JFrame* with the method **setJMenuBar:**

```
JFrame window = new JFrame("Some Application");
JMenuBar menuBar = new JMenuBar();
window.setJMenuBar(menuBar);
```

Menu

- Menus are *JMenu* instances and added to menu bar:

```
JMenu batter = new JMenu("Batter");
menuBar.add(batter);
```

- Menu choices are *JMenuItem* instances, and are added to menu:

```
JMenuItem swing = new JMenuItem("Swing");
JMenuItem take = new JMenuItem("Take");
JMenuItem bunt = new JMenuItem("Bunt");
batter.add(swing);
batter.add(take);
batter.add(bunt);
```

Menubar and Menu

JMenuItem listener

- When the user selects an item, the *JMenuItem* selected generates an *ActionEvent*.

- Implement an *ActionListener* for each *JMenuItem* to program menu events.

Java code examples for widgets

- Visit:
- http://java.sun.com/docs/books/tutorial/uiswing/components/
- And choose choice: How to …

Dialog

- A window to present information or gather input from user.
- For standard dialogs use:*JOptionPane*, *JFileChooser*, and *JColorChooser*
- For custom dialogs use *JDialog*.

Dialog

- Every dialog
  - Has *owner,* a frame.
  - It’s destroyed if owner is destroyed,
  - disappears from the screen while owner is iconified.

- Two kinds of dialogs
  - *modal :* User input to all other windows is blocked when a modal dialog is visible.
  - *non-modal :* dialogs for which you must use *JDialog*.

JOptionPane `showMessageDialog`

- Used to create simple, standard dialogues.

```
public static void showMessageDialog (
   Component parentComponent,
   Object message,
   String title,
   int messageType,
   Icon icon
);
```

Frame owner

String message to be displayed

Window's title

`int` value indicating style of message

icon to be displayed in dialog

JOptionPane showInputDialog

- Used to get input from user. It gets a *String* from user, using either a text field or a combo box.

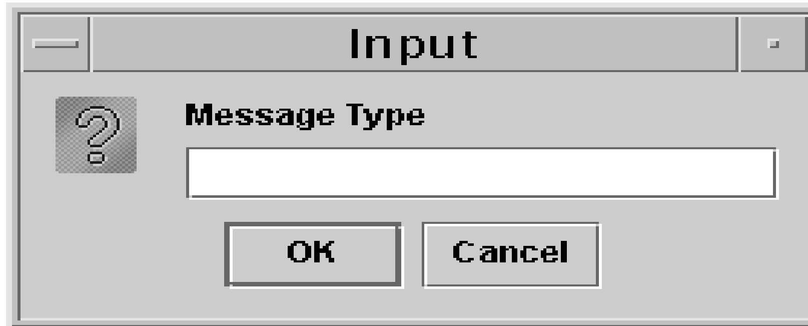
- Parameters are the same as in **showMessageDialog**.

- A simpler variants of method is specified as

```
public static String showInputDialog (
        Component parentComponent, Object message)
```

- When user presses “OK” button:
  - contents of text field is returned or null if user presses “Cancel” or closes window.
  - Contents is *String*. Requesting a number from user, you must validate and convert *String* to appropriate type of value.

showInputDialog



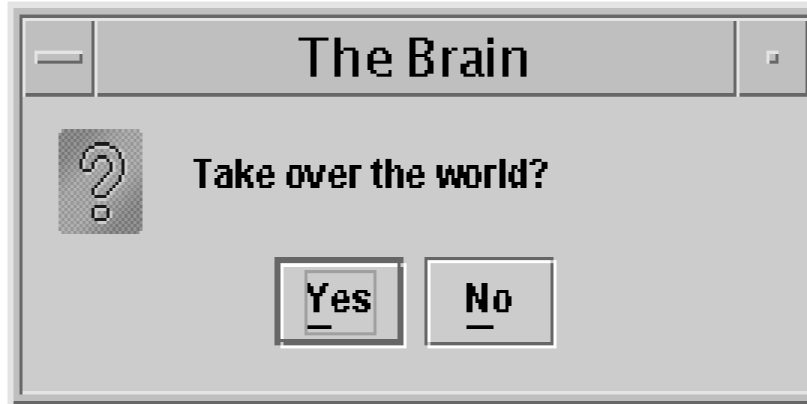

```
String response =
      JOptionPane.showInputDialog(frame, “Message Type”);
int value = convertToInt(response);
```

JOptionPane method showConfirmDialog

- The **showConfirmDialog** generates a two or three button window.
- The two button provides “Yes” and “No” or ‘OK” and “Cancel” buttons.
- The three button, “Yes,” “No,” and “Cancel” buttons.
- The method returns an **int** indicating the user’s response. Possible return values include

```
JOptionPane.YES_OPTION,
JOptionPane.OK_OPTION,
JOptionPane.NO_OPTION, JOptionPane.CANCEL_OPTION,
  and if user closes window,
JOptionPane.CLOSED_OPTION.
```

Show confirm dialog

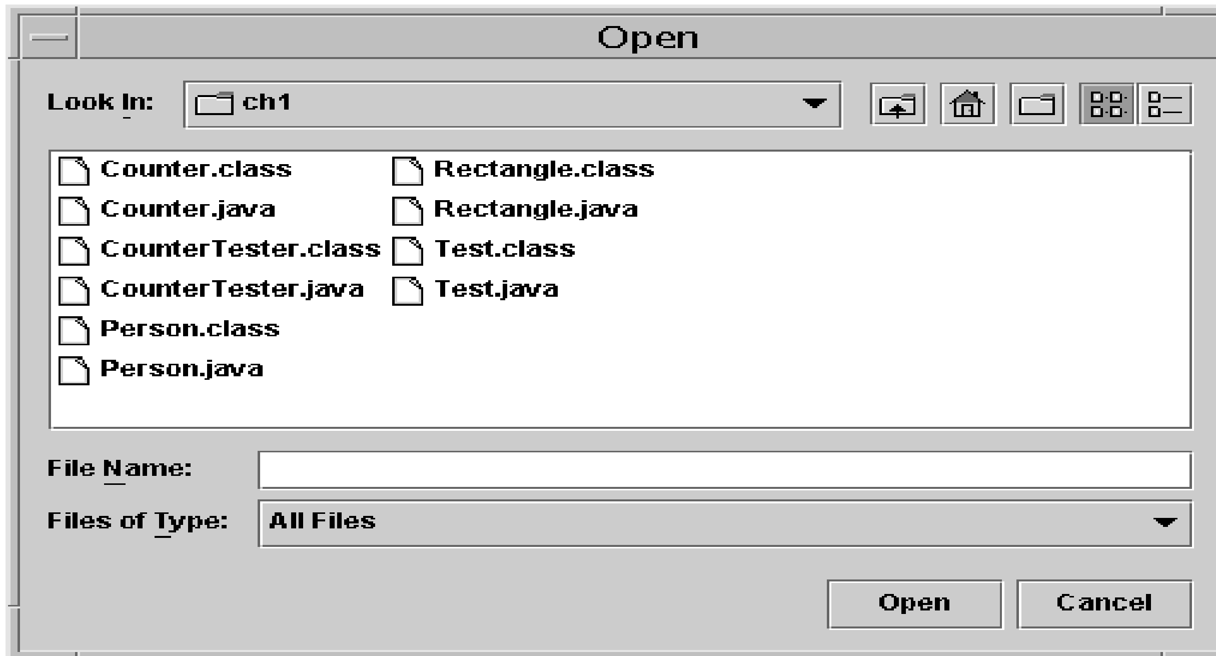



```
int response =
JOptionPane.showConfirmDialog(frame,
    “Take over the world?”,
    “The Brain”, JOptionPane.YES_NO_OPTION);
if (response == YES_OPTION) …
```

FileChooser and JColorChooser dialogs

- *JFileChooser* : mechanism for user to select a file.

```
JFileChooser directory = new JFileChooser();
directory.setCurrentDirectory(new File(“.”));
directory.showOpenDialog(this); //open dialog.
File file = directory.getSelectedFile();
```



FileChooser and JColorChooser dialogs

- *JColorChooser* presents a pane of controls that allow a user to select and manipulate a color.

JDialog

- Used to create custom dialog windows.

- A *Jdialog*
  - a top-level window.
  - has an owner, generally a frame.
  - It delegates component management to a content pane, to which components are added.
  - It’s displayed by invoking its **setVisible** method with an argument of **true**, and is hidden by invoking its **setVisible** method with an argument of **false**

JDialog

- A typical constructor is specified as follows:

```
public JDialog (Frame owner, String title,
                boolean modal)
```

- Provides an object to create custom views to get or present data.