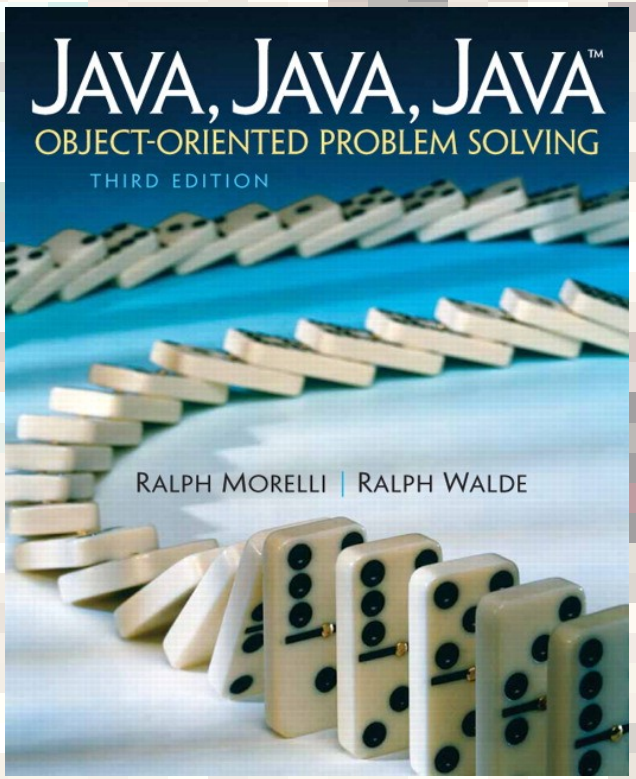


presentation slides for

Object-Oriented Problem Solving

JAVA, JAVA, JAVA

Third Edition

Ralph Morelli | Ralph Walde
Trinity College
Hartford, CT

published by Prentice Hall

Java, Java, Java

Object Oriented Problem Solving

Lecture 05: Java Swing and Event-Driven Programming

Objectives

- Gain more experience with the Swing component set.
- Understand the relationship between the AWT and Swing.
- Learn more about Java's event-driven programing model.
- Be able to design and build simple Graphical User Interfaces (GUI)s.
- Appreciate how object-oriented design principles were used to extend Java's GUI capabilities.

Outline

- Introduction to Javax Swing
  - Object-Oriented Design: Model-View-Controller Architecture
  - The Swing Component Set
  - Containers and Layout Managers
- Java Event-Driven Programming Model
- Case Study: Designing a Basic GUI
- Swing Controls
  - Checkboxes, Radio Buttons, and Borders
  - Menus and Scroll Panes

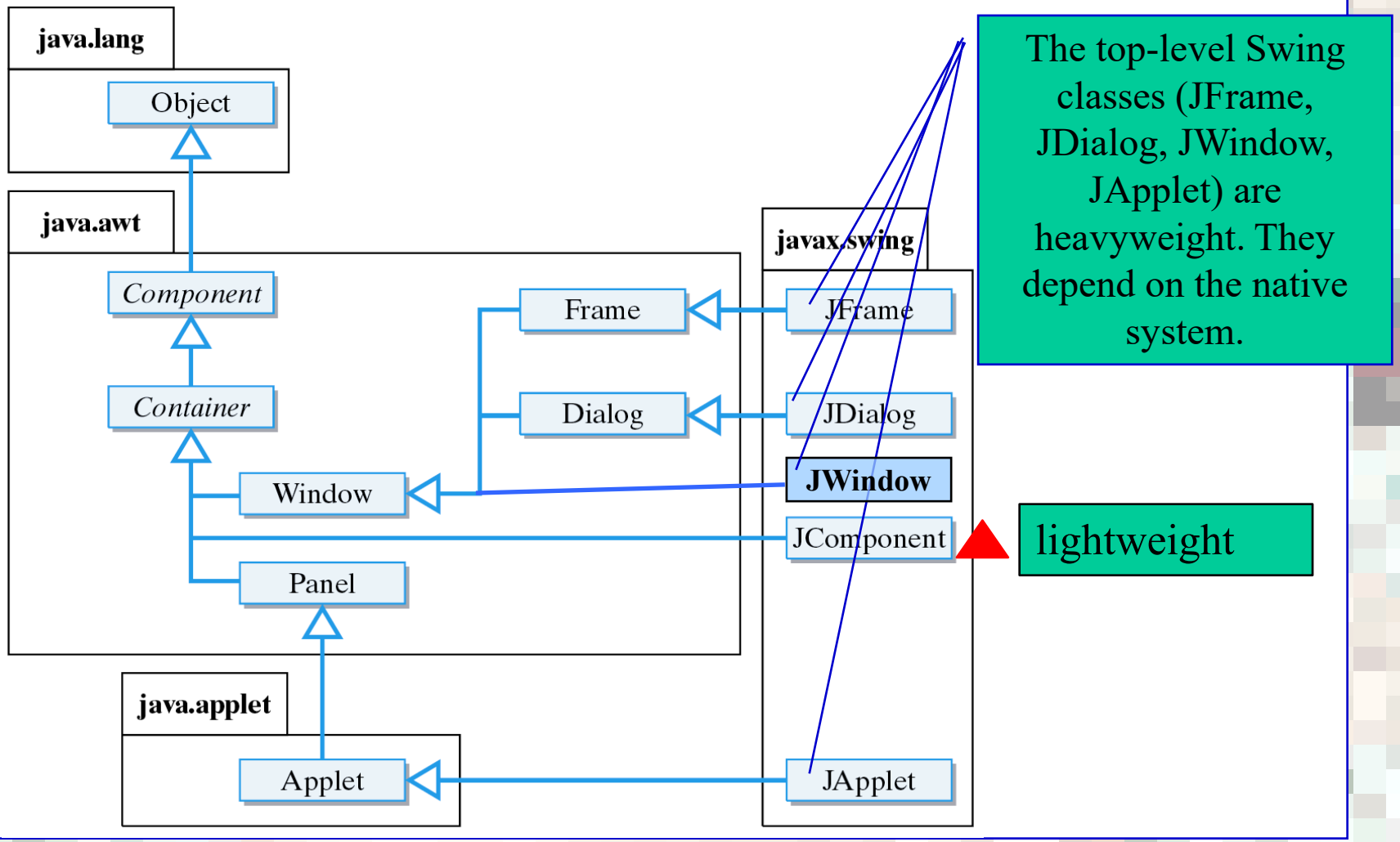
Introduction

- A *Graphical User Interface (GUI)* creates a certain way of interacting with a program.
- It is what gives a program its *look and feel*.
- A GUI uses a set of basic components, such as buttons, text fields, labels, and text areas.
- Java's GUI classes provide an excellent example of object-oriented design.

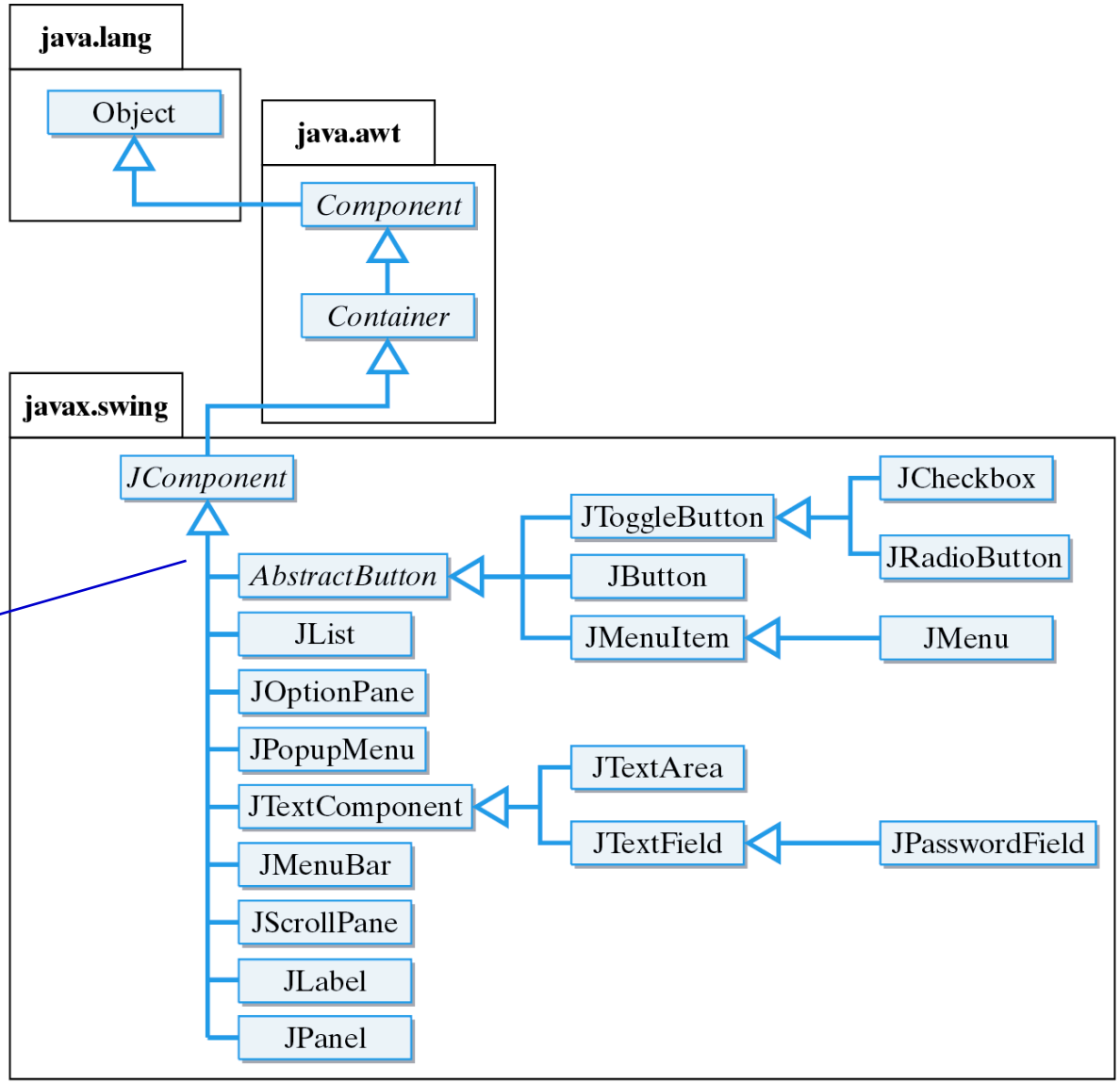
A Brief History

- The original AWT was suitable for Java applets but not for full-fledged application development.
- AWT 1.1 (JDK 1.1) had better event handling but did not have enough GUI components and was too dependent on native code (nonportable) .
- In 1997 Netscape and Sun developed a set of GUI classes written entirely in Java. The *Java Foundation Classes (JFC)*, including the Swing component set, were released with JDK 2.0.
- A Swing program can have the same look and feel on a Mac, Windows, or Unix platform.

Swing Hierarchy



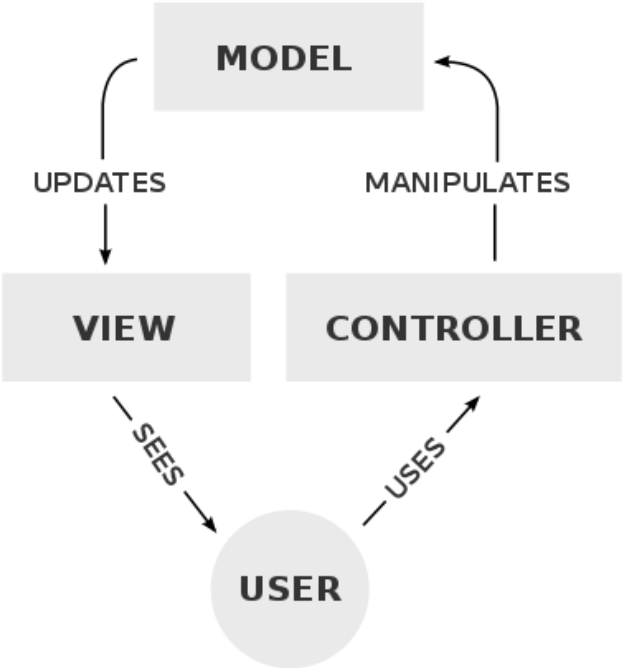
Swing Hierarchy
(Part II)



Swing components names start with 'J'. They are lightweight components as they are derived from the JComponent class.

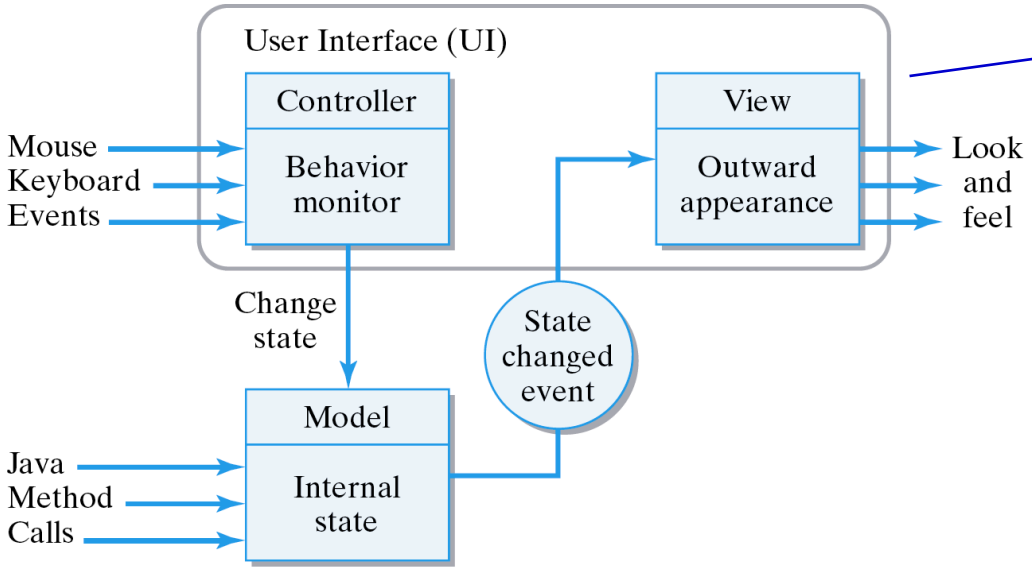
Model-View-Controller Architecture

- **Model-View-Controller (MVC)** : is a software design pattern commonly used to develop graphical user interfaces that divides the related program logic into three interconnected elements.



Model-View-Controller Architecture

- **Model-View-Controller (MVC)** : Swing components composed of 3 aspects: state (*model*), look (*view*), and behavior (*controller*).



For some Swing components (e.g., text components), 3 separate classes are used for its MVC. For others (e.g., Jbutton), the VC are combined into a single BasicButtonUI class.

- JButton click: the *controller* (e.g, mouse click event) puts *model* in pressed state, which changes its outward appearance (e.g., background color) *(view)*.

Swing Components

- A *component* is an independent visual control, such as push button, slider, etc.
- **Swing packages**
  - **javax.swing.event.*** contains Swing events and listeners; similar to java.awt.event.*
  - **javax.swing.plaf.*** contains Swing's *look-and-feel* classes. (plaf: pluggable look and feel)
  - **javax.swing.text.*** contains the classes for JTextField and JTextArea, the Swing classes that replace the AWT's TextField and TextArea classes.
- **Platform independent look and feel.**
  - **javax.swing.plaf.windows** looks like Windows
  - **javax.swing.plaf.motif** looks like Unix (XWindows)

Containers

- A *container* is a component that contains other components -- e.g., JPanel, JFrame, JApplet.
- A *top-level container* is the top of the container hierarchy and cannot be contained within any other container – e.g., JFrame, Japplet, JWindow, and JDialog.
- Container methods:

Container
+ add(in c : Component) : Component + remove(in index : int) + remove(in c : Component) + removeAll()

Layout Managers

- A *layout manager* is an object that manages the layout and organization of a container, including:
  - Size of container.
  - Size of each element in the container.
  - Position and spacing between elements.

Layout Managers

- Set layout for JApplet, JDialog, JFrame, and JWindow, e.g.,

```
getContentPane().setLayout(new FlowLayout());
```

- Set layout for JPanel, e.g.,

```
setLayout(new GridLayout(4,3,1,1));
```

- Note: Top-level containers, such as JFrame, are the only ones that use a content pane. For other containers, such as JPanel, components are added directly to the container itself.

Types of Layout Managers

Manager	Description
java.awt.FlowLayout	Arranges elements left to right across the container.
java.awt.BorderLayout	Arranges elements along the north, south, east, west, and in the center of the container.
java.swing.BoxLayout	Arranges elements in a single row or single column.
java.awt.CardLayout	Arranges elements like a stack of cards, with one visible at a time.
java.awt.GridLayout	Arranges elements into a two-dimensional grid of equally sized cells.
java.swing.OverlayLayout	Arranges elements on top of each other.
java.awt.GridBagLayout	Arranges elements in a grid of variable sized cells (complicated).

Default Layout for Swing Containers

In AWT, the default layout for Applet was FlowLayout. In Swing, it's BorderLayout

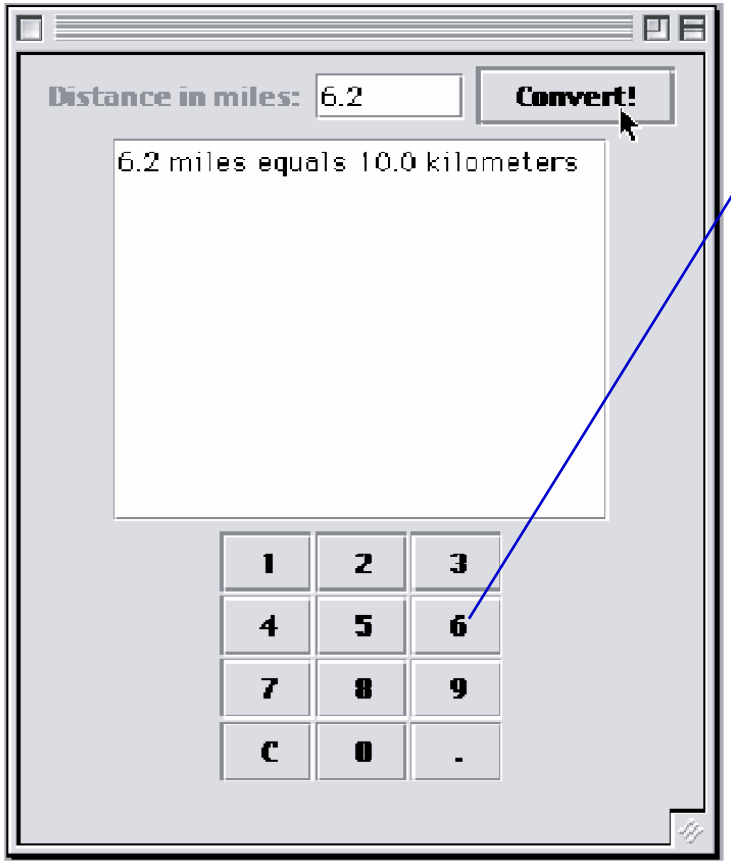
Container	Layout Manager
JApplet	BorderLayout (on its content pane)
JBox	BoxLayout
JDialog	BorderLayout (on its content pane)
JFrame	BorderLayout (on its content pane)
JPanel	FlowLayout
JWindow	BorderLayout (on its content pane)

JPanel uses FlowLayout by default.

Top-level windows (JApplet, JDialog, JFrame, JWindow) use BorderLayout.

The GridLayout Manager

- A GridLayout arranges components in a two-dimensional grid.




```
keypadPanel.setLayout(
   new GridLayout(4,3,1,1));
```

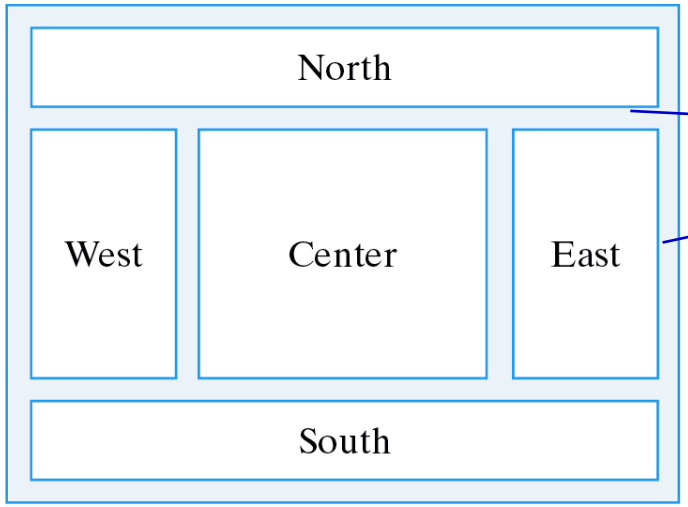
4 rows and 3 columns

1 space between each row and 1 space between each column

- **Design Critique:** We should use BorderLayout for top-level window, i.e., JFrame.

The BorderLayout Manager

- A BorderLayout divides the container into five areas: North, South, East, West, and Center.

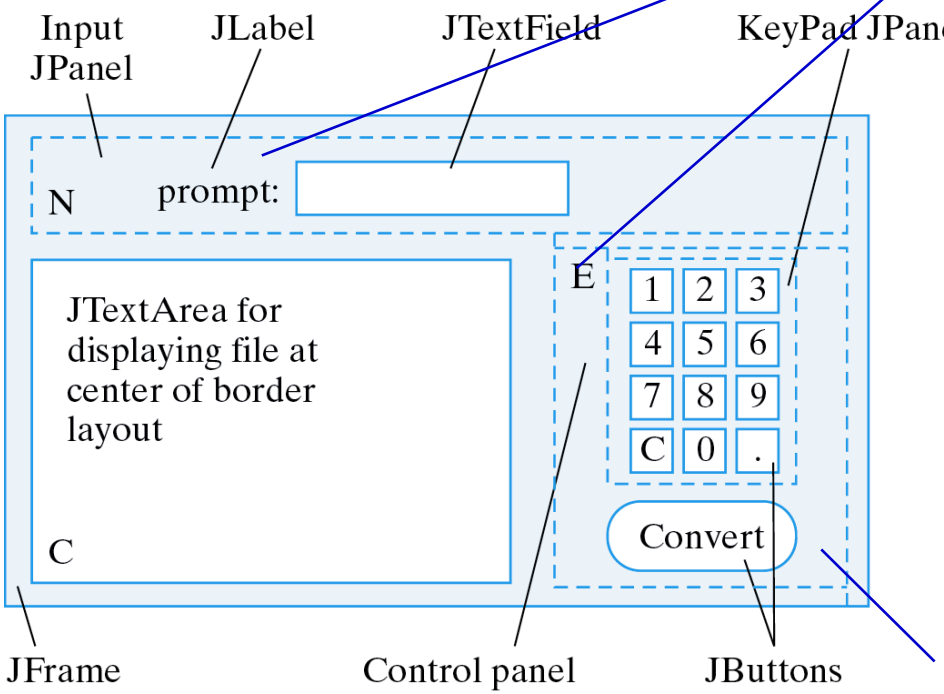


Horizontal and vertical gaps.

- Use add(Component, String) method to add components to a border layout :

```
getContentPane().setLayout(new BorderLayout(2, 2));
getContentPane().add(keypadPanel, "East"); //capitalized
```


Converter: BorderLayout Design



Panels are used to group components by function.

Containment Hierarchy

- JFrame(Border)
  - Input JPanel(Flow)
    - Prompt JLabel
    - Input JTextField
  - Control JPanel(Border)
    - KeyPad JPanel(Grid)
      - 12 Button
    - Convert JButton
  - Display JTextArea

All the controls are grouped together.

Converter: BorderLayout Implementation

```
public Converter() {
    getContentPane().setLayout(new BorderLayout());
    initKeyPad();
    JPanel inputPanel = new JPanel();          // Input panel
    inputPanel.add(prompt);
    inputPanel.add(input);
    getContentPane().add(inputPanel,"North");
    JPanel controlPanel = new JPanel(new BorderLayout(0, 0));// Controls
    controlPanel.add(keypadPanel, "Center");
    controlPanel.add(convert, "South");
    getContentPane().add(controlPanel, "East");
    getContentPane().add(display, "Center");   // Output display
    display.setLineWrap(true);
    display.setEditable(false);
    convert.addActionListener(this);
    input.addActionListener(this);
} // Converter()
```

The GUI is layed out in the constructor.

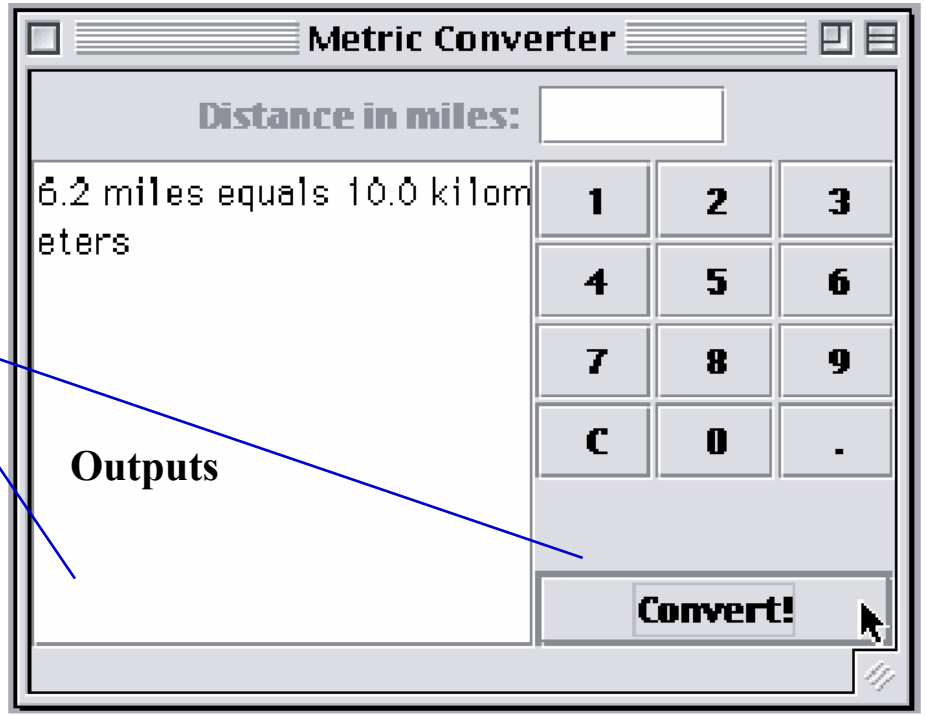
By default, the JPanel has FlowLayout.

Use BorderLayout for JPanel in ‘controlPanel’.

Converter: Final View

- In BorderLayout, when one or more areas is not used, then one or more of the other areas fills its space, except for the center, which would be left blank if unused.

The unused south area is filled by center and east.

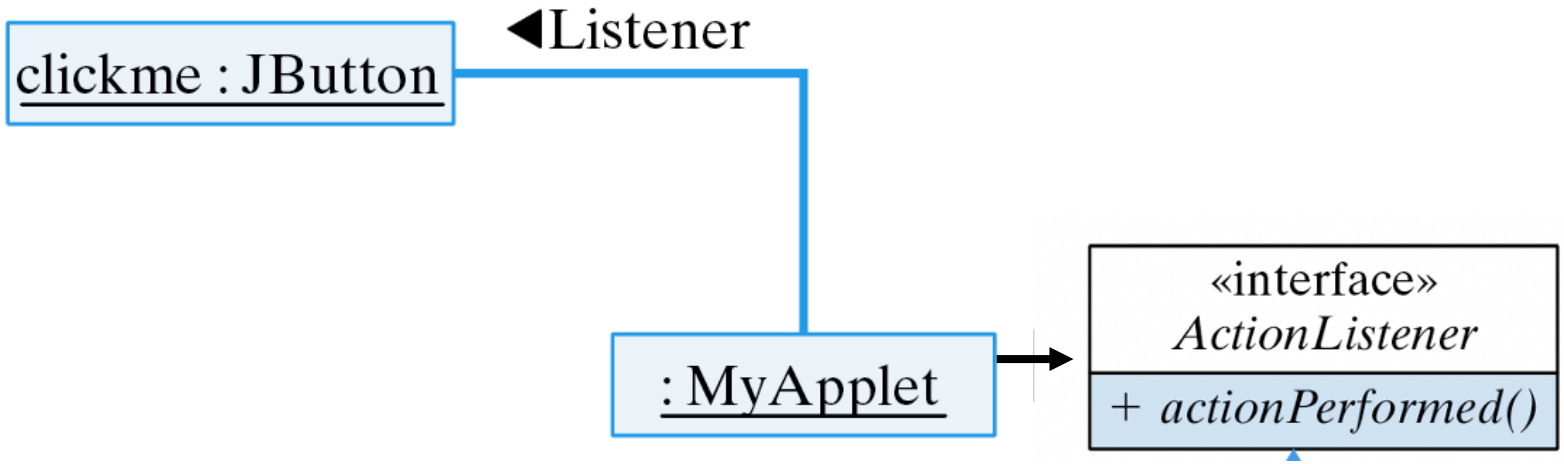


Part 2: Event-Driven Programming Model

- ***Event-driven programming***: the execution flow of a program is dictated by events
  - Program waits for user input events
  - For each event, program runs the specific code to respond
  - The execution flow of the program is determined by the series of events
- Event: An object that represents a user's interaction with a GUI component; can be "handled" to create interactive components.
- All events are handled by objects called ***listeners -*** An object that waits for events and responds to them.
  - To handle an event, attach a listener to a component.
  - listener will be notified when the event occurs (e.g., button click).

Part 2: Event-Driven Programming Model

- GUI events examples:
  - Mouse button press/release, mouse move/click/drag
  - Keyboard: key press/release + Shift, ALT, CTRL, etc.
  - Touchscreen finger tap, drawing tablet, joystick, etc.
  - Window resize/minimize/restore/close
  - Timer (for animations)



Part 2: Event-Driven Programming Model

```
import java.awt.event.*;
    EventObject
        - AWTEvent (AWT)
        • ActionEvent
        • TextEvent
        • ComponentEvent
            • FocusEvent
            • WindowEvent
            • InputEvent
                • KeyEvent
                • MouseEvent
```

An action that has occurred on a GUI component

Button / Menu / Checkbox / Text box …

Keyboard events

Mouse click / move / drag, wheel

Event objects contain information about the event
– UI object that triggered the event
– Other information depending on event. Examples:

`ActionEvent` – text string from a button

`MouseEvent` – mouse coordinates

Creating an ActionListener

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

public class MyGUI extends JFrame implements ActionListener {
    private JButton clickme = new JButton("ClickMe");

    public void MyGUI() {
        getContentPane().add(clickme);   // Add clickme to the applet
        clickme.addActionListener(this); //  and assign it a listener
        setSize(200,200);
        setVisible(true);
    } // MyGUI()

    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == clickme) {
            clickme.setText(clickme.getText() + "*");
        }
    } // actionPerformed()

    public static void main(String args[]) {
        MyGUI gui = new MyGUI();
    }
} // MyGUI
```

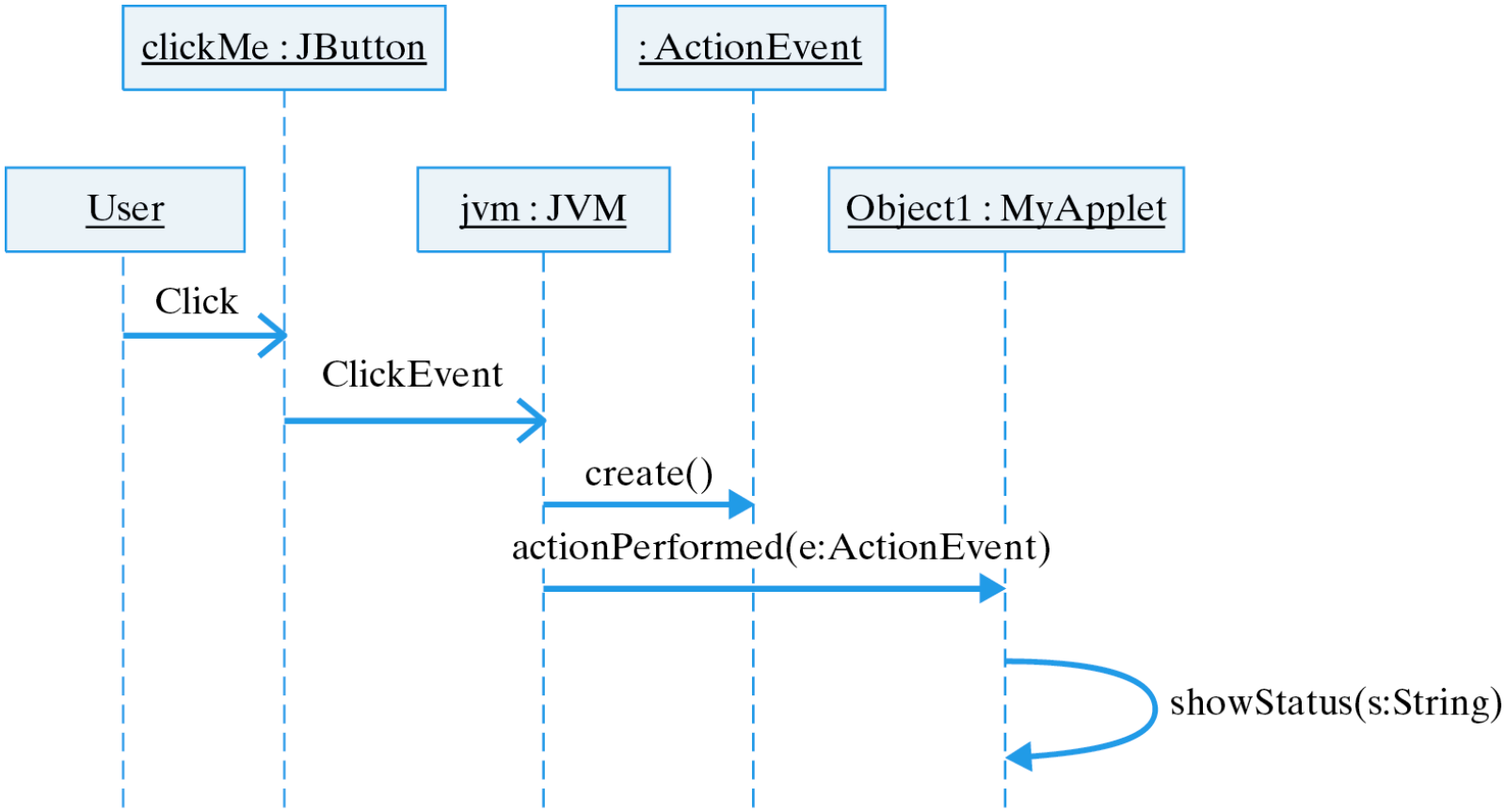
Button click events are handled by ActionListener

The window frame (i.e., MyGUI instance) is the listener. Use addActionListener in constructor

The actionPerformed() method contains code that handles the button click event.

Handling an ActionEvent

- Sequence of actions and events.



The EventObject Class

- The ActionEvent is derived from EventObject class (java.util.EventObject).

- In EventObject, the getSource() method is used to get the Object that caused the event.

EventObject
+ EventObject(in src : Object) + getSource() : Object + toString() : String

Event Classes

- AWT events for each type of component.

Components	Events	Description
Button, JButton	ActionEvent	User clicked button
CheckBox, JCheckBox	ItemEvent	User toggled a checkbox
CheckboxMenuItem, JCheckboxMenuItem	ItemEvent	User toggled a checkbox
Choice, JPopupMenu	ItemEvent	User selected a choice
Component, JComponent	ComponentEvent	Component was moved or resized
	FocusEvent	Component acquired or lost focus
	KeyEvent	User typed a key
	MouseEvent	User manipulated the mouse
Container, JContainer	ContainerEvent	Component added/removed from container
List, JList	ActionEvent	User double-clicked a list item
	ItemEvent	User clicked a list item
Menu, JMenu	ActionEvent	User selected menu item
Scrollbar, JScrollbar	AdjustmentEvent	User moved scrollbar
TextComponent, JTextComponent	TextEvent	User edited text
TextField, JTextField	ActionEvent	User typed Enter key
Window, JWindow	WindowEvent	User manipulated window

New Swing Event Classes

- Newly defined Swing events.

Component	Events	Description
JPopupMenu	PopupMenuEvent	User selected a choice
JComponent	AncestorEvent	An event occurred in an ancestor
JList	ListSelectionEvent	User double-clicked a list item
	ListDataEvent	List's contents were changed
JMenu	MenuEvent	User selected menu item
JTextComponent	CaretEvent	Mouse clicked in text
	UndoableEditEvent	An undoable edit has occurred
JTable	TableModelEvent	Items added/removed from table
	TableColumnModelEvent	A table column was moved
JTree	TreeModelEvent	Items added/removed from tree
	TreeSelectionEvent	User selected a tree node
	TreeExpansionEvent	User changed tree node
JWindow	WindowEvent	User manipulated window

Part 3: Case Study of a Basic GUI

- Basic User Interface Tasks:
  - Provide help/guidance to the user.
  - Allow input of information.
  - Allow output of information.
  - Control interaction between the user and device.

The MetricConverter Class

- Problem Description: Design a GUI for a Java application that converts miles to kilometers. The class that performs the conversions is defined as:

MetricConverter
+ milesToKm(in mi : double) : double

```
public class MetricConverter {

    public static double milesToKm(double miles) {
        return miles / 0.62;
    }
}
```


GUI Design: Choosing Components

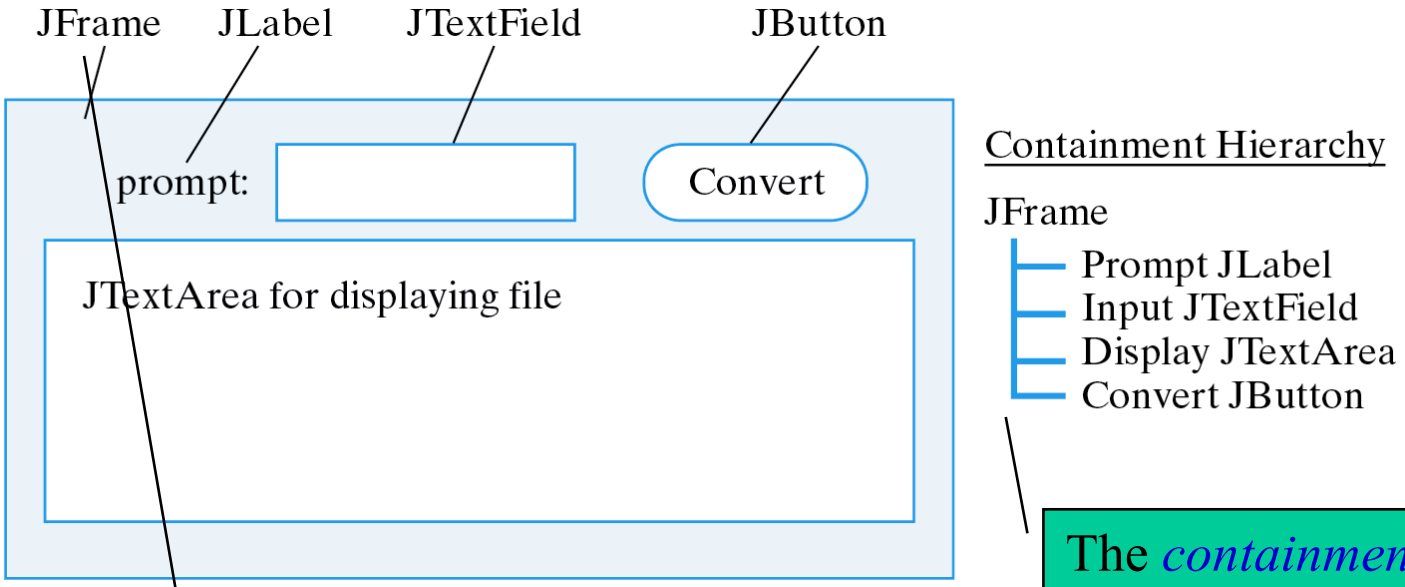
- Swing objects for input, output, control, guidance:
  - **Guidance**: A JLabel displays a short string of text or an image. It can serve as a prompt.
  - **Input**: A JTextField allows editing of a single line of text. It can get the user's input.
  - **Output**: A JTextArea allows editing of multiple lines of text. We'll use it to display results.
  - **Control**: A JButton is an action control. By implementing the ActionListener interface we will handle the user's action events.

GUI Design: Choosing the Top-Level Window

- For applets, top-level window is JApplet.
- For applications, a JFrame is used.
- Both JApplet and JFrame are subclasses of Container and are suitable for holding the interface components.
- Both JApplet and JFrame are *heavyweight* components.

GUI Design: Designing a Layout

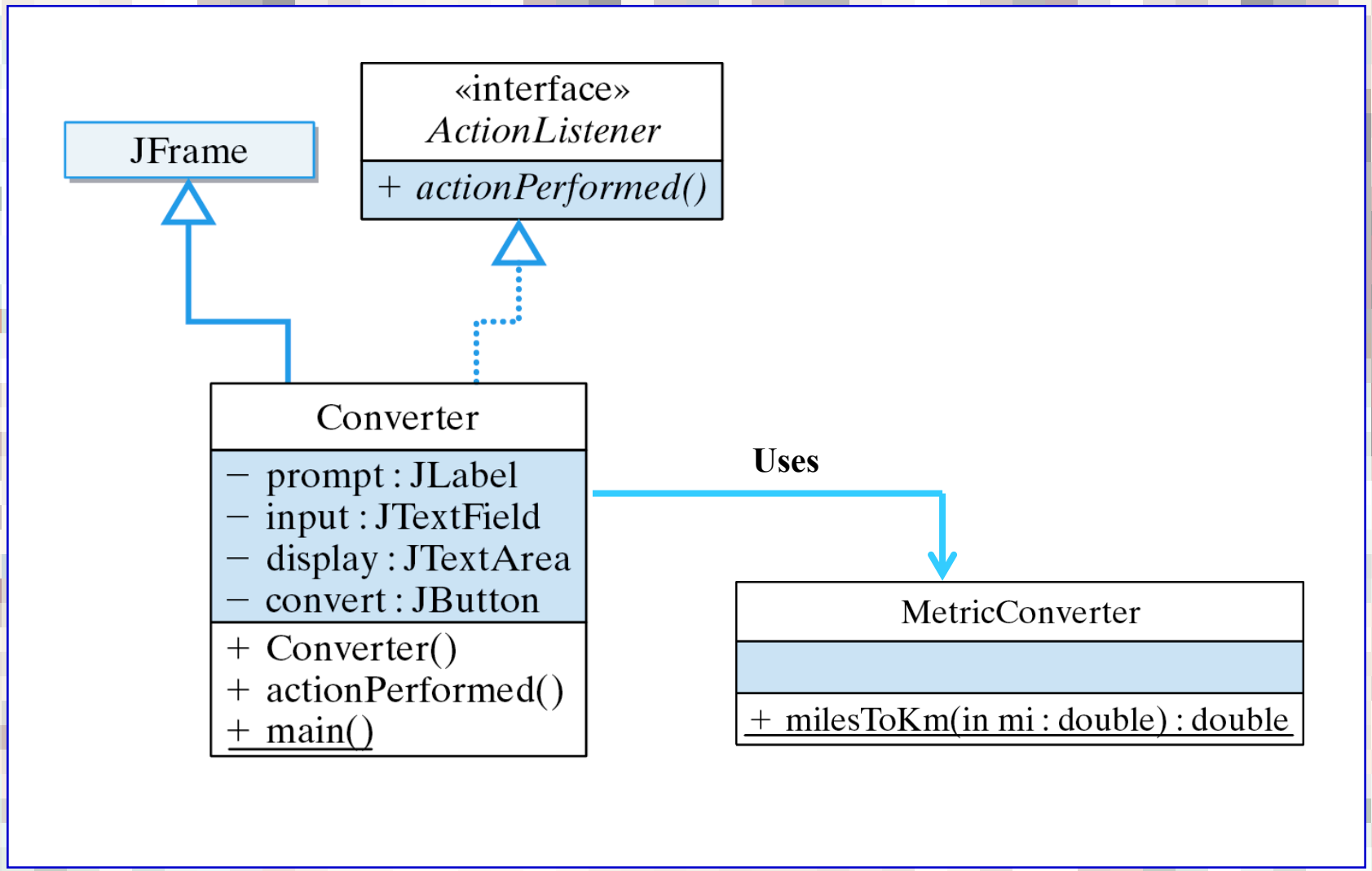
- In a FlowLayout components are arranged left to right in rows within the container.



The top-level window is a JFrame.

The *containment hierarchy* shows which components are contained in containers.

Class Design



Implementing the Converter Class

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

public class Converter extends JFrame implements ActionListener{
    private JLabel prompt = new JLabel("Distance in miles: ");
    private JTextField input = new JTextField(6);
    private JTextArea display = new JTextArea(10,20);
    private JButton convert = new JButton("Convert!");

    public Converter() {
        getContentPane().setLayout(new FlowLayout());
        getContentPane().add(prompt);
        getContentPane().add(input);
        getContentPane().add(convert);
        getContentPane().add(display);
        display.setLineWrap(true);
        display.setEditable(false);
        convert.addActionListener(this);
    } // Converter()

    public void actionPerformed( ActionEvent e )
        double miles =
                Double.valueOf(input.getText()).doubleValue();
        double km = MetricConverter.milesToKm(miles);
        display.append(miles + " miles equals " + km +
                " kilometers\n");
    } // actionPerformed()
} // Converter
```

Declare the components.
JTextField(int columns)
JTextArea(int rows, int columns)

Set FlowLayout

For top-level Swing windows, components are added to the *content pane*.

Invoke MetricConverter when the JButton is clicked.

Convert String to double.

Instantiating the Top-Level JFrame

```
public static void main(String args[])
      Converter f = new Converter();
      f.setSize(400, 300);
      f.setVisible(true);
                        // Quit the application
      f.addWindowListener(new WindowAdapter() {
          public void windowClosing(WindowEvent e) {
              System.exit(0);
          }
      });
} // main()
```

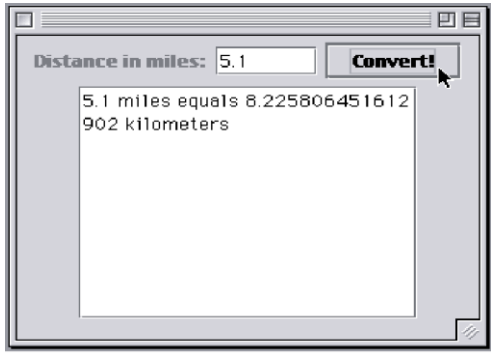
It is necessary to set the size and visibility.

An *anonymous inner class* is used to create an *adapter* to listen for window close events. (See Appendix F)

See demo.

GUI Design Critique

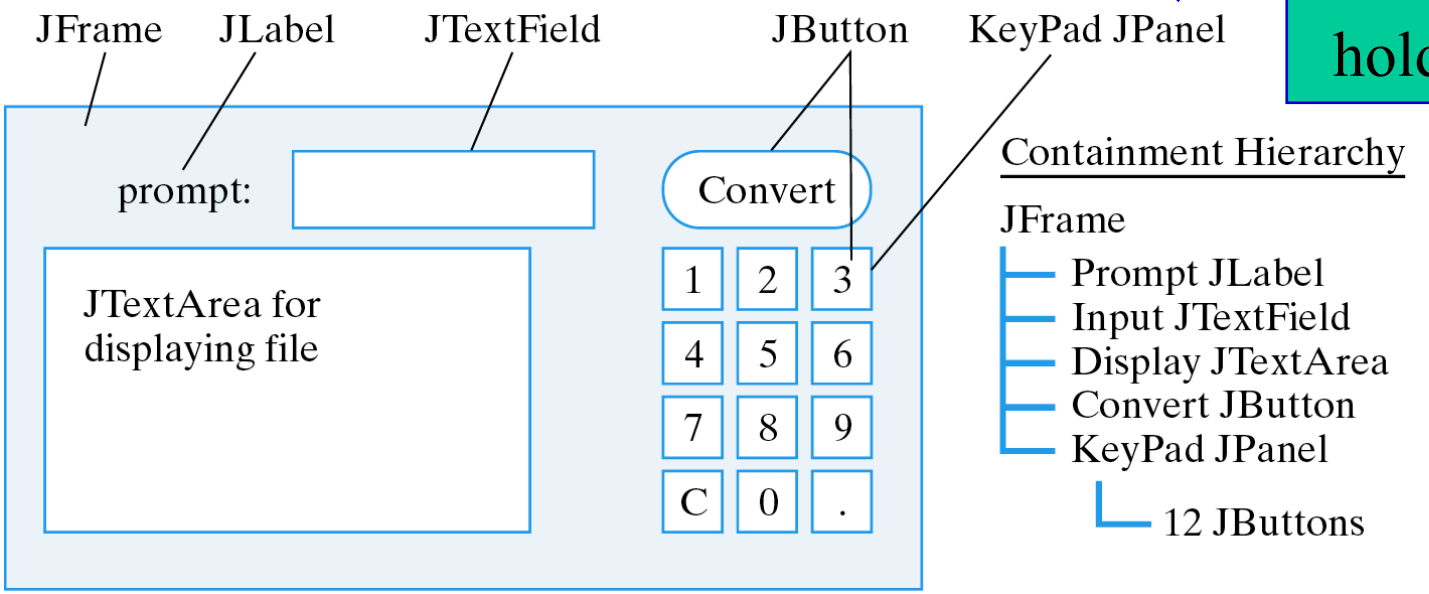
- The user has to manually clear the input field.
  - **Effective Design:** Minimize the user's burden.
  - Clear JTextField after button is clicked.
- The user has to use both keyboard and mouse.
  - **Effective Design:** Minimize the number of input devices needed to perform a single task.
  - Make the JTextField a control so the user doesn't have to use the mouse to perform conversions.



Extending the GUI: Button Array

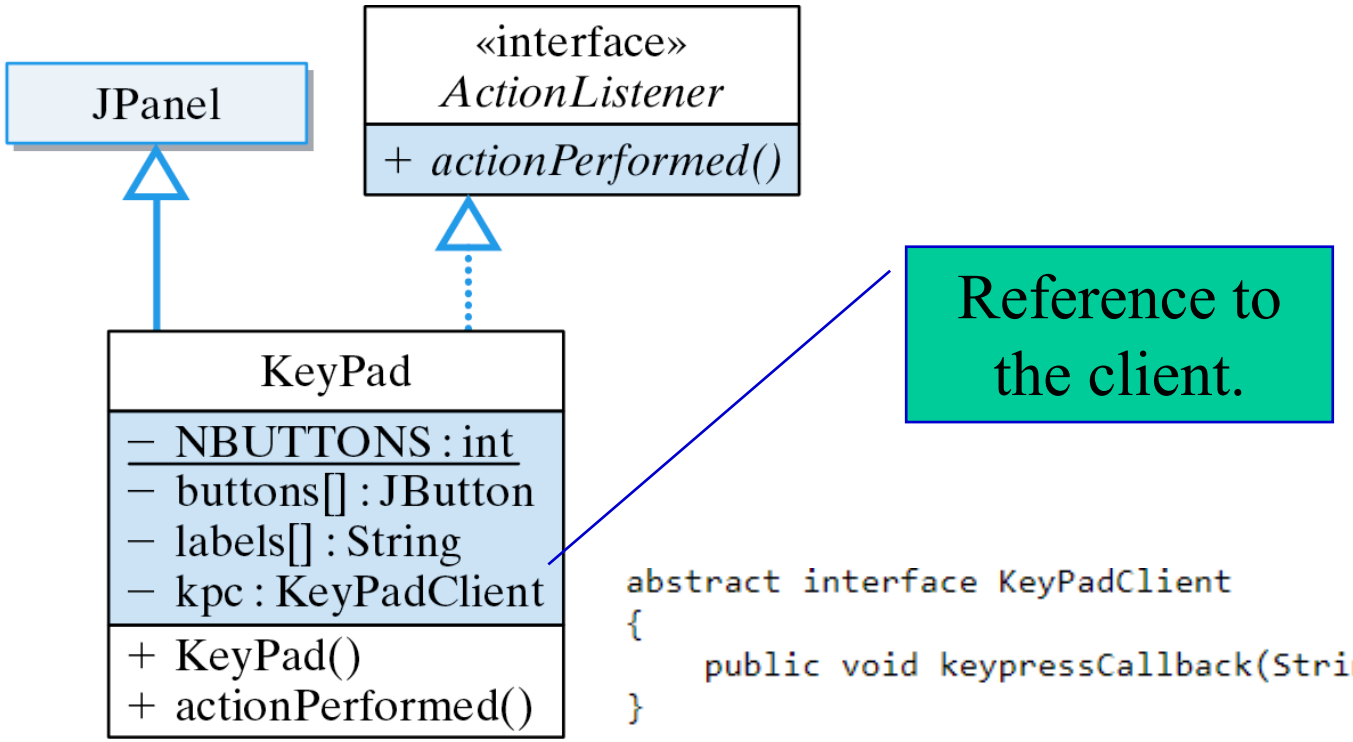
- Implement a 10-key pad so a conversion can be done with just the mouse control.
- **Effective Design:** Redundant controls.

Note the use of a JPanel to hold buttons.



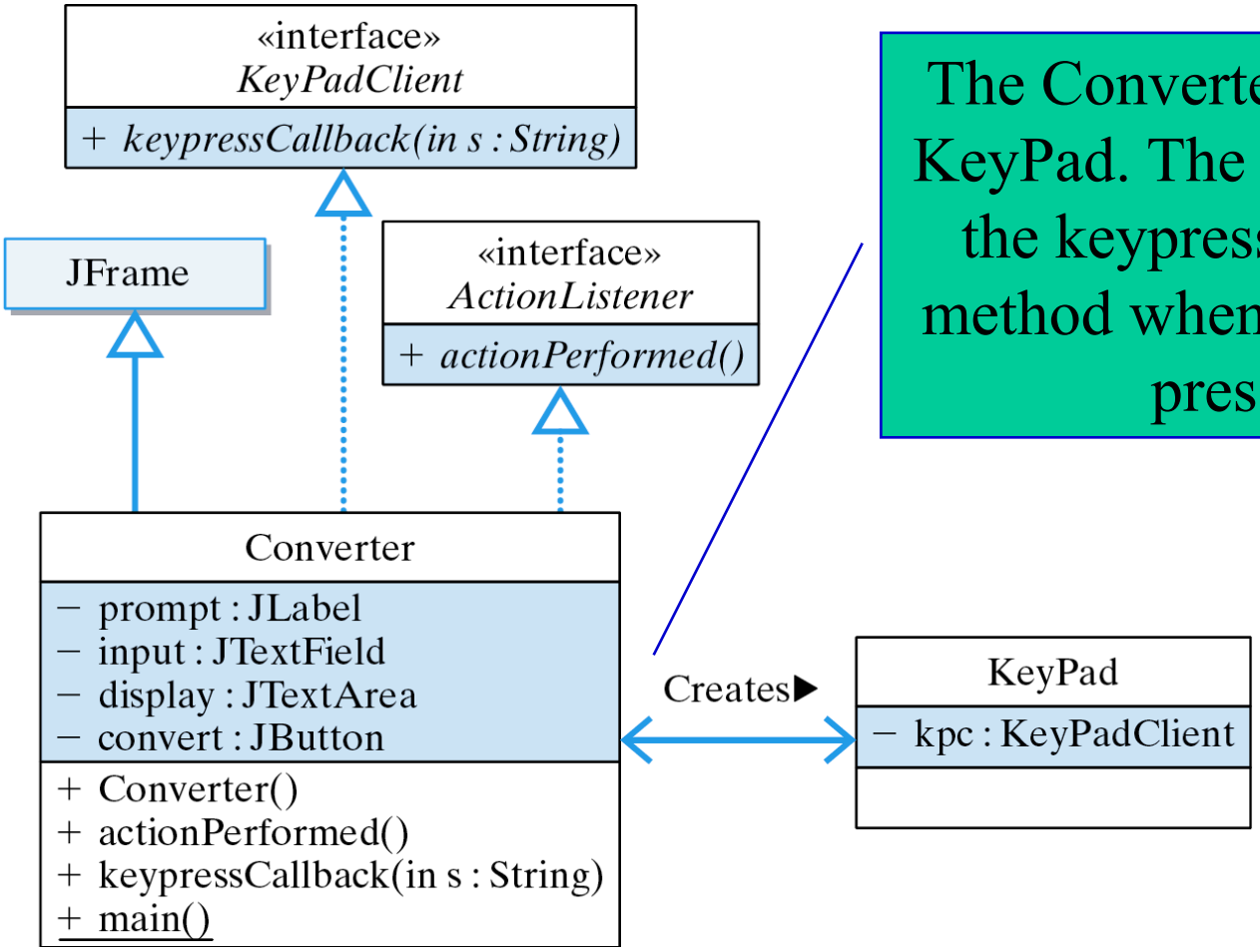
The KeyPad JPanel

- The KeyPad JPanel handles its own actions.




```
abstract interface KeyPadClient
{
    public void keypressCallback(String s);
}
```


The Callback Method Design



The Converter creates the KeyPad. The KeyPad calls the keypressCallback() method whenever a key is pressed.

Implementation: The KeyPad Class

```
public class KeyPad extends JPanel implements ActionListener {
    private final static int NBUTTONS = 12;
    private KeyPadClient kpc;     // Owner of the KeyPad
    private JButton buttons[];    // An array of buttons
    private String labels[] =     // And their labels
              { "1","2","3",
                "4","5","6",
                "7","8","9",
                "C","0","." };
    public KeyPad(KeyPadClient kpc) {
        this.kpc = kpc;
        buttons = new JButton[NBUTTONS];       // Create the array
        for(int k = 0; k < keyPad.length; k++) { // For each button
           buttons[k] = new JButton(labels[k]);  // Create it w/label
           buttons[k].addActionListener(this);   // and a listener
           add(buttons[k]);                      // and add to panel
        } // for
    }
    public void actionPerformed(ActionEvent e) {
        String keylabel = ((JButton)e.getSource()).getText();
        kpc.keypressCallback(keylabel);
    }
}
```

Buttons and labels stored in arrays.

Callback.

Implemention: The Callback Method

- KeyPad's actionPerformed() calls the client's keypressCallback() method, passing it the key's label.

```
public void keypressCallback(String s) {
     if (s.equals("C"))
          input.setText("");  // Clear the input
     else
          input.setText(input.getText() + s); // Type the key
}
```

Clear the input

Keypad Input: Concatenate button's label with the contents of the input field.

Checkboxes

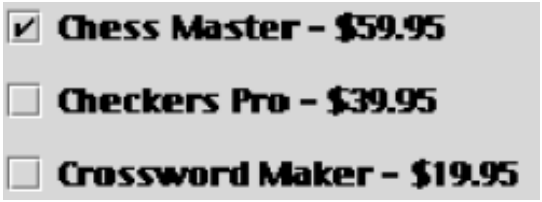
- A JCheckBox is a button which always displays its current state (selected or not).
- Used to select one or more options from a set.

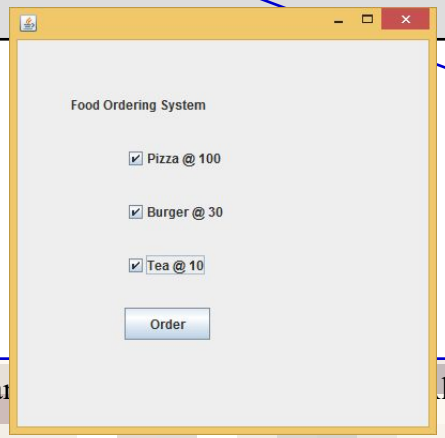
```
private JCheckBox titles[] = new JCheckBox[NTITLES];
private String titleLabels[] =
    {"Chess Master - $59.95", "Checkers Pro - $39.95",
        "Crossword Maker - $19.95"};

for(int k = 0; k < titles.length; k++) {
      titles[k] = new JCheckBox(titleLabels[k]);
      titles[k].addItemListener(this);
      choicePanel.add(titles[k]);
}
```

An array of checkboxes.

An array of checkbox labels.





Each checkbox is given a listener (ItemListener) and added to a JPanel.

Radio Buttons

- A JRadioButton is a button that belongs to a ButtonGroup of mutually exclusive alternatives. Only one button from the group may be selected at a time.

```
private ButtonGroup optGroup = new ButtonGroup();
private JRadioButton options[] = new JRadioButton[NOPTIONS];
private String optionLabels[] = {"Credit Card", "Debit Card",
                                                 "E-cash"};
for(int k = 0; k < options.length; k++) {
      options[k] = new JRadioButton(optionLabels[k]);
      options[k].addItemListener(this);
      optionPanel.add(options[k]);
      optGroup.add(options[k] );
}
options[0].setSelected(true); // Set the first button on default
```

Add each button to the button group.

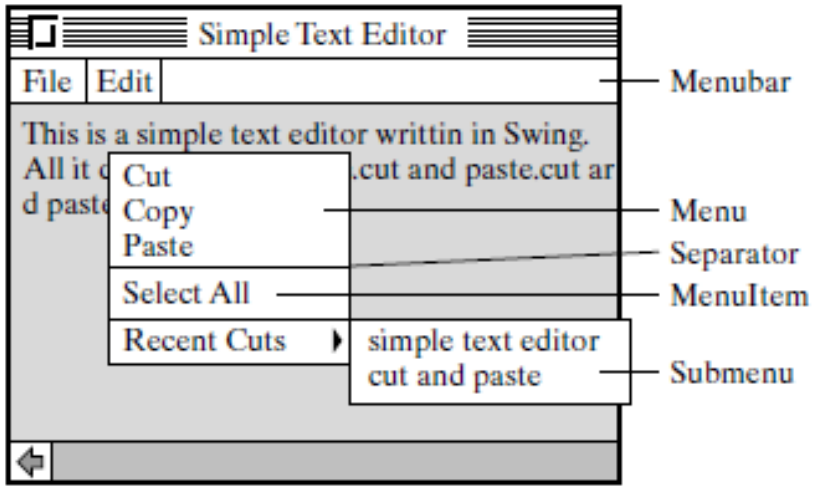
Menus

- Pop-up and pull-down menus allow an application or applet to grow in complexity and functionality without cluttering its interface
- 3 steps to create menus:
  - Create the individual JMenuItems.
  - Create a JMenu and add the JMenuItems to it.
  - Create a JMenuBar and add the JMenus to it.

Menus Example

```
JMenuBar mBar = new JMenuBar ( ) ; // Create menu bar
this.setMenuBar (mBar ) ; // Add it  to this window

fileMenu = new JMenu( "File" ) ; // Create menu
mBar.add ( fileMenu ) ; // Add it to menu bar
openItem = new JMenuItem( "Open" ) ; // Open tem
openItem . addActionListener ( this ) ;
openItem . setEnabled ( false ) ;
fileMenu . add ( openItem ) ;
saveItem = new JMenuItem( "Save" ) ; // Save item
saveItem . addActionListener ( this ) ;
saveItem . setEnabled ( false ) ;
fileMenu . add ( saveItem ) ;
fileMenu . addSeparator ( ) ; // Logical separator
quitItem = new JMenuItem( "Quit " ) ; // Quit item
quitItem.addActionListener ( this ) ;
fileMenu.add ( quitItem ) ;
```



Handling Menu Actions

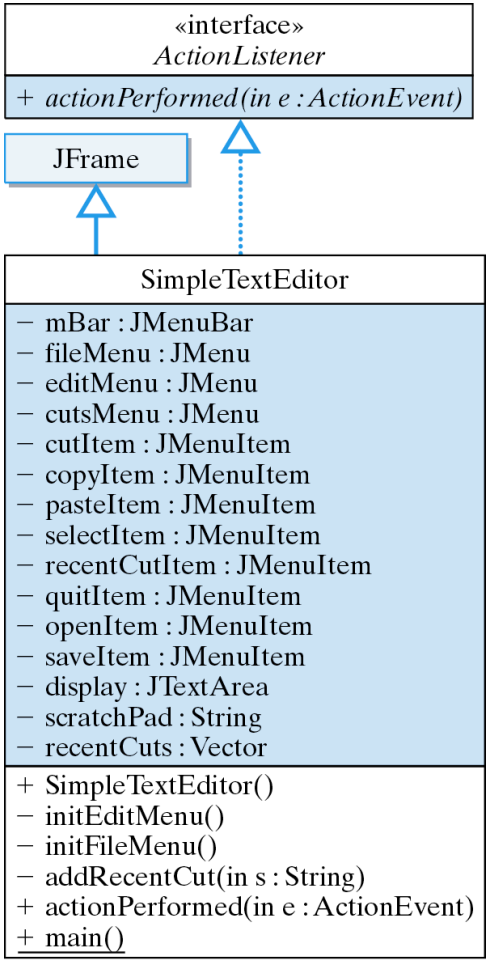
- Menu item selections generate ActionEvents.
- **Algorithm**: *Multiway selection*. Test for each menu item and take appropriate action.

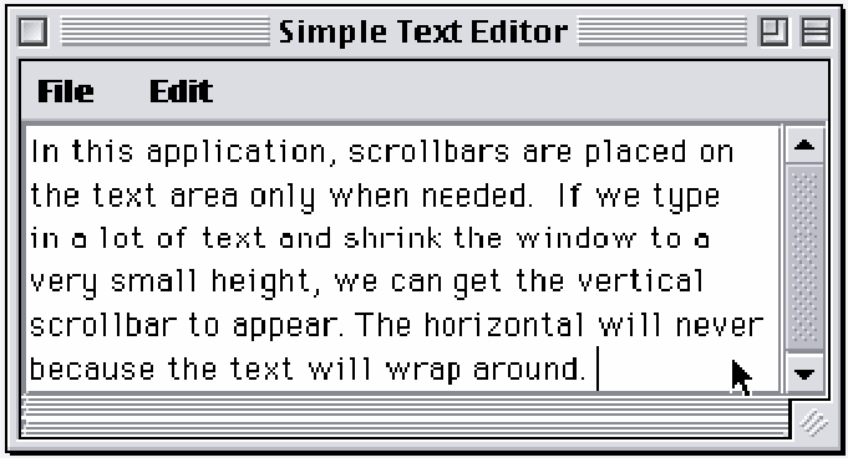
```
public void actionPerformed(ActionEvent e) {
    JMenuItem m  = (JMenuItem)e.getSource();
    if ( m == quitItem ) {                          // Quit
        dispose();
    } else if (m == copyItem)                       // Copy
        scratchPad = display.getSelectedText();
    } else if (m == pasteItem) {                    // Paste
        display.insert(scratchPad, display.getCaretPosition());
    } else if ( m == selectItem ) {
        display.selectAll();            // Select entire document
    }
} // actionPerformed()
```

Need to cast source object.

A scratchpad (String) is used to store text.

JScrollPane



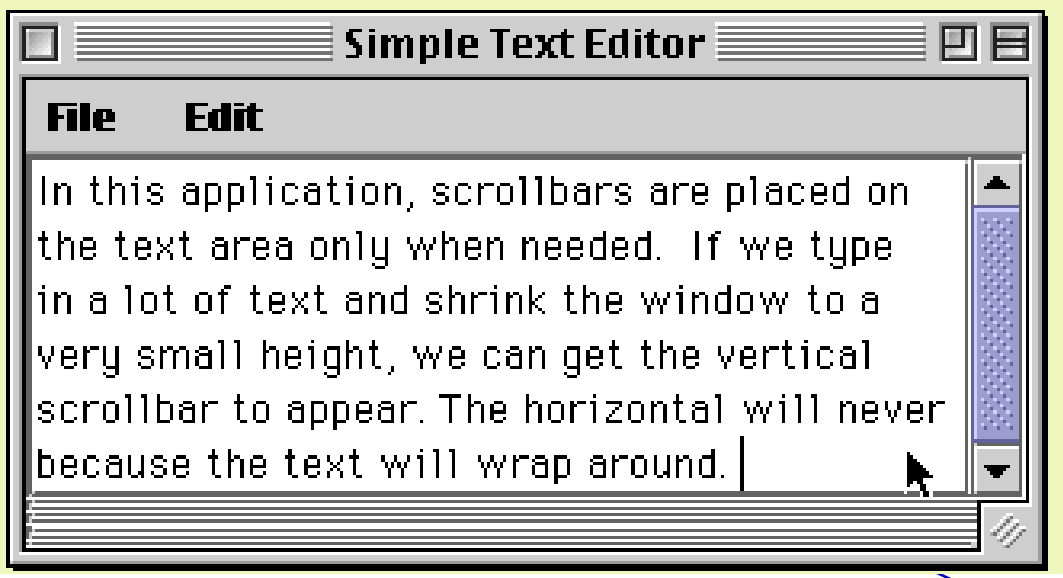


- A JScrollPane is an object that manages scrolling within a window or JTextArea.

```
  this.getContentPane().add(new
JScrollPane(display));
```

This parameter refers to the scrollable component.

Scroll Pane Example



Scroll bars appear on the text area only when they are needed.

Horizontal scroll bar is unnecessary.

Technical Terms

- callback design
- content pane
- containment hierarchy
- controller
- event model
- layout manager
- lightweight component
- listener
- model
- model-view-controller (MVC)
- peer model
- pluggable look and feel
- view
- widget hierarchy

Summary Of Important Points

- Graphical User Interface (GUI) components:
  - Abstract Windowing Toolkit (AWT): JDK 1.0 - 1.2.
  - Swing: the GUI part of Java Foundation Classes (JFC) JDK 1.1, 1.2 (Java 2).
    - Written entirely in Java (pure Java).
    - Platform-independent look and feel.
    - *Pluggable look-and-feel*:
      - Windows 95 style
      - Unix-like Motif style
      - Java Metal style

Summary Of Important Points (cont)

- *Model-View-Controller (MVC):* Components divided into three separate objects: look (*view*), state (model), and behavior (*controller*).
- *User interface* classes -- combine look behavior, for a *pluggable look-and-feel*.
- *Peer model* : every AWT component has a peer in the native windowing system. This model is less efficient and more platform dependent than the MVC model.

Summary Of Important Points (cont)

- *Event model* is based on *event listeners* -- objects responsibile for handling the component's events.
- User interface design: *guidance* of the user, *input*, *output* and *control*.
- *Containment hierarchy:* A hierarchy of the containers and their contents in a GUI (JPanels and other Containers).
- *Content pane*: Used by top-level Swing classes --- JApplet, JDialog, JFrame, and JWindow --- as their component container.

Summary Of Important Points (cont)

- A GUI should minimize the number of input devices that users need to manipulate.
- Redundancy: some forms -- e.g., two independent but complete sets of controls --- are desirable.
- *Layout manager:* manages the arrangement of the components in a container (flow, border, grid and box layouts).
- A *radio button* belongs to a group such that only one button may be selected at a time. A *checkbox* is a toggle button that always displays its state.
- Interface Design: reduce chance of user error and make it easy to recover from errors when they do occur.

