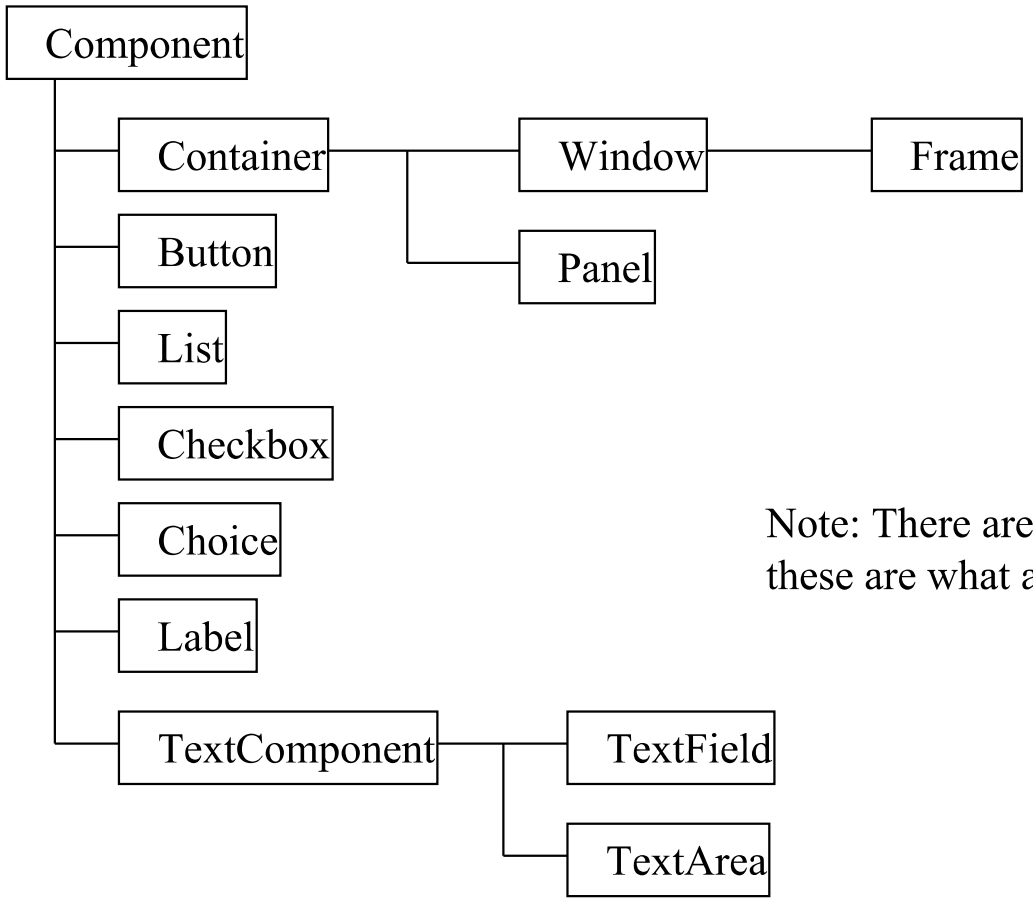
AWT (Abstract Windowing Toolkit)

- The AWT is roughly broken into three categories
  - Components
  - Layout Managers
  - Graphics

- Many AWT components have been replaced by Swing components
- It is generally not considered a good idea to mix Swing components and AWT components. Choose to use one or the other.

AWT □ Class Hierarchy



Note: There are more classes, however, these are what are covered in this chapter

Component

- Component is the superclass of most of the displayable classes defined within the AWT. Note: it is abstract.
- MenuComponent is another class which is similar to Component except it is the superclass for all GUI items which can be displayed within a drop-down menu.
- The Component class defines data and methods which are relevant to all Components

```
setBounds
setSize
setLocation
setFont
setEnabled
setVisible
setForeground       -- colour
setBackground       -- colour
```

Container

- Container is a subclass of Component. (ie. All containers are themselves, Components)
- Containers contain components
- For a component to be placed on the screen, it must be placed within a Container
- The Container class defined all the data and methods necessary for managing groups of Components
  - add
  - getComponent
  - getMaximumSize
  - getMinimumSize
  - getPreferredSize
  - remove
  - removeAll

Windows and Frames

- The Window class defines a top-level Window with no Borders or Menu bar.
  - Usually used for application splash screens
- Frame defines a top-level Window with Borders and a Menu Bar
  - Frames are more commonly used than Windows
- Once defined, a Frame is a Container which can contain Components

```
Frame aFrame = new Frame("Hello World");
aFrame.setSize(100,100);
aFrame.setLocation(10,10);
aFrame.setVisible(true);
```

Panels

- When writing a GUI application, the GUI portion can become quite complex.
- To manage the complexity, GUIs are broken down into groups of components. Each group generally provides a unit of functionality.
- A Panel is a rectangular Container whose sole purpose is to hold and manage components within a GUI.

```
Panel aPanel = new Panel();
aPanel.add(new Button("Ok"));
aPanel.add(new Button("Cancel"));

Frame aFrame = new Frame("Button Test");
aFrame.setSize(100,100);
aFrame.setLocation(10,10);

aFrame.add(aPanel);
```

Buttons

- This class represents a push-button which displays some specified text.
- When a button is pressed, it notifies its Listeners. (More about Listeners in the next chapter).
- To be a Listener for a button, an object must implement the ActionListener Interface.

```
Panel aPanel = new Panel();
Button okButton = new Button("Ok");
Button cancelButton = new Button("Cancel");

aPanel.add(okButton));
aPanel.add(cancelButton));

okButton.addActionListener(controller2);
cancelButton.addActionListener(controller1);
```

Labels

- This class is a Component which displays a single line of text.
- Labels are read-only. That is, the user cannot click on a label to edit the text it displays.
- Text can be aligned within the label

```
Label aLabel = new Label("Enter password:");
aLabel.setAlignment(Label.RIGHT);

aPanel.add(aLabel);
```

List

- This class is a Component which displays a list of Strings.
- The list is scrollable, if necessary.
- Sometimes called Listbox in other languages.
- Lists can be set up to allow single or multiple selections.
- The list will return an array indicating which Strings are selected

```
List aList = new List();
aList.add("Calgary");
aList.add("Edmonton");
aList.add("Regina");
aList.add("Vancouver");

aList.setMultipleMode(true);
```

Checkbox

- This class represents a GUI checkbox with a textual label.
- The Checkbox maintains a boolean state indicating whether it is checked or not.
- If a Checkbox is added to a CheckBoxGroup, it will behave like a radio button.

```
Checkbox creamCheckbox = new CheckBox("Cream");
Checkbox sugarCheckbox = new CheckBox("Sugar");
[□]
if (creamCheckbox.getState())
{
  coffee.addCream();
}
```

Choice

- This class represents a dropdown list of Strings.
- Similar to a list in terms of functionality, but displayed differently.
- Only one item from the list can be selected at one time and the currently selected element is displayed.

```
Choice aChoice = new Choice();
aChoice.add("Calgary");
aChoice.add("Edmonton");
aChoice.add("Alert Bay");
[□]

String selectedDestination= aChoice.getSelectedItem();
```

TextField

- This class displays a single line of optionally editable text.
- This class inherits several methods from TextComponent.
- This is one of the most commonly used Components in the AWT

```
TextField emailTextField = new TextField();
TextField passwordTextField = new TextField();
passwordTextField.setEchoChar("*");
[…]

String userEmail = emailTextField.getText();
String userpassword = passwordTextField.getText();
```

TextArea

- This class displays multiple lines of optionally editable text.
- This class inherits several methods from TextComponent.
- TextArea also provides the methods: appendText(), insertText() and replaceText()

```
// 5 rows, 80 columns
TextArea fullAddressTextArea = new TextArea(5, 80);
[□]

String userFullAddress= fullAddressTextArea.getText();
```

Layout Managers

- Since the Component class defines the setSize() and setLocation() methods, all Components can be sized and positioned with those methods.
- Problem: the parameters provided to those methods are defined in terms of pixels. Pixel sizes may be different (depending on the platform) so the use of those methods tends to produce GUIs which will not display properly on all platforms.
- Solution: Layout Managers. Layout managers are assigned to Containers. When a Component is added to a Container, its Layout Manager is consulted in order to determine the size and placement of the Component.
- NOTE: If you use a Layout Manager, you can no longer change the size and location of a Component through the setSize and setLocation methods.

Layout Managers (cont)

- There are several different LayoutManagers, each of which sizes and positions its Components based on an algorithm:
  - FlowLayout
  - BorderLayout
  - GridLayout
- For Windows and Frames, the default LayoutManager is BorderLayout. For Panels, the default LayoutManager is FlowLayout.

Flow Layout

- The algorithm used by the FlowLayout is to lay out Components like words on a page: Left to right, top to bottom.
- It fits as many Components into a given row before moving to the next row.

```
Panel aPanel = new Panel();
aPanel.add(new Button("Ok"));
aPanel.add(new Button("Add"));
aPanel.add(new Button("Delete"));
aPanel.add(new Button("Cancel"));
```

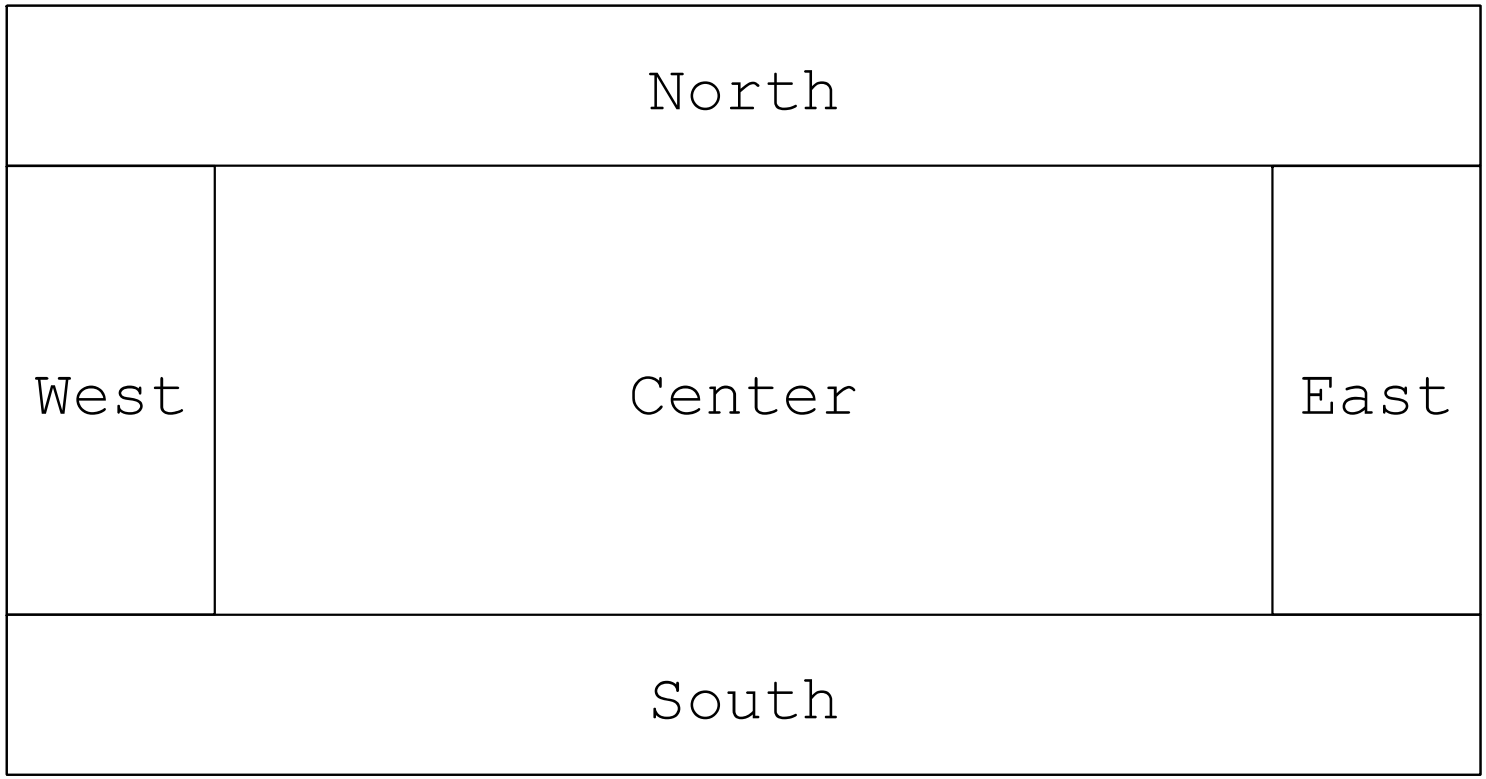
Border Layout

- The BorderLayout Manager breaks the Container up into 5 regions (North, South, East, West, and Center).
- When Components are added, their region is also specified:

```
Frame aFrame = new Frame();
aFrame.add("North", new Button("Ok"));
aFrame.add("South", new Button("Add"));
aFrame.add("East", new Button("Delete"));
aFrame.add("West", new Button("Cancel"));
aFrame.add("Center", new Button("Recalculate"));
```

Border Layout (cont)

- The regions of the BorderLayout are defined as follows:



Grid Layout

- The GridLayout class divides the region into a grid of equally sized rows and columns.
- Components are added left-to-right, top-to-bottom.
- The number of rows and columns is specified in the constructor for the LayoutManager.

```
Panel aPanel = new Panel();
GridLayout theLayout = new GridLayout(2,2);
aPanel.setLayout(theLayout);

aPanel.add(new Button("Ok"));
aPanel.add(new Button("Add"));
aPanel.add(new Button("Delete"));
aPanel.add(new Button("Cancel"));
```

What if I don□t want a LayoutManager?

- LayoutManagers have proved to be difficult and frustrating to deal with.
- The LayoutManager can be removed from a Container by invoking its setLayout method with a null parameter.

```
Panel aPanel = new Panel();
aPanel.setLayout(null);
```

Graphics

- It is possible to draw lines and various shapes within a Panel under the AWT.
- Each Component contains a Graphics object which defines a Graphics Context which can be obtained by a call to getGraphics().
- Common methods used in Graphics include:

drawLine
drawOval
drawPolygon
drawPolyLine
drawRect
drawRoundRect
drawString
draw3DRect
fill3DRect
fillArc

- fillOval
- fillPolygon
- fillRect
- fillRoundRect
- setColor
- setFont
- setPaintMode
- drawImage

AWT (Abstract Window Toolkit)

- Present in all Java implementations
- Described in most Java textbooks
- Adequate for many applications
- Uses the controls defined by your OS
  - therefore it's “least common denominator”
- Difficult to build an attractive GUI
- import java.awt.*;
  import java.awt.event.*;

Swing

- Same concepts as AWT
- Doesn't work in ancient Java implementations (Java 1.1 and earlier)
- Many more controls, and they are more flexible
  - Some controls, but not all, are a lot more complicated
- Gives a choice of "look and feel" packages
- Much easier to build an attractive GUI
- `import javax.swing.*;`

Swing vs. AWT

- Swing is bigger, slower, and more complicated
  - But much faster than it used to be
- Swing is more flexible and better looking
- Swing and AWT are *incompatible*--you can use either, but you can't mix them
  - Actually, you can, but it's tricky and not worth doing
- Learning the AWT is a good start on learning Swing
- Many of the most common controls are just

To build a GUI...

- Make somewhere to display things—usually a Frame or Dialog (for an application), or an Applet
- Create some Components, such as buttons, text areas, panels, etc.
- Add your Components to your display area
- Arrange, or *lay out*, your Components
- Attach Listeners to your Components
  - Interacting with a Component causes an Event to occur
  - A Listener gets a message when an interesting

Containers and Components

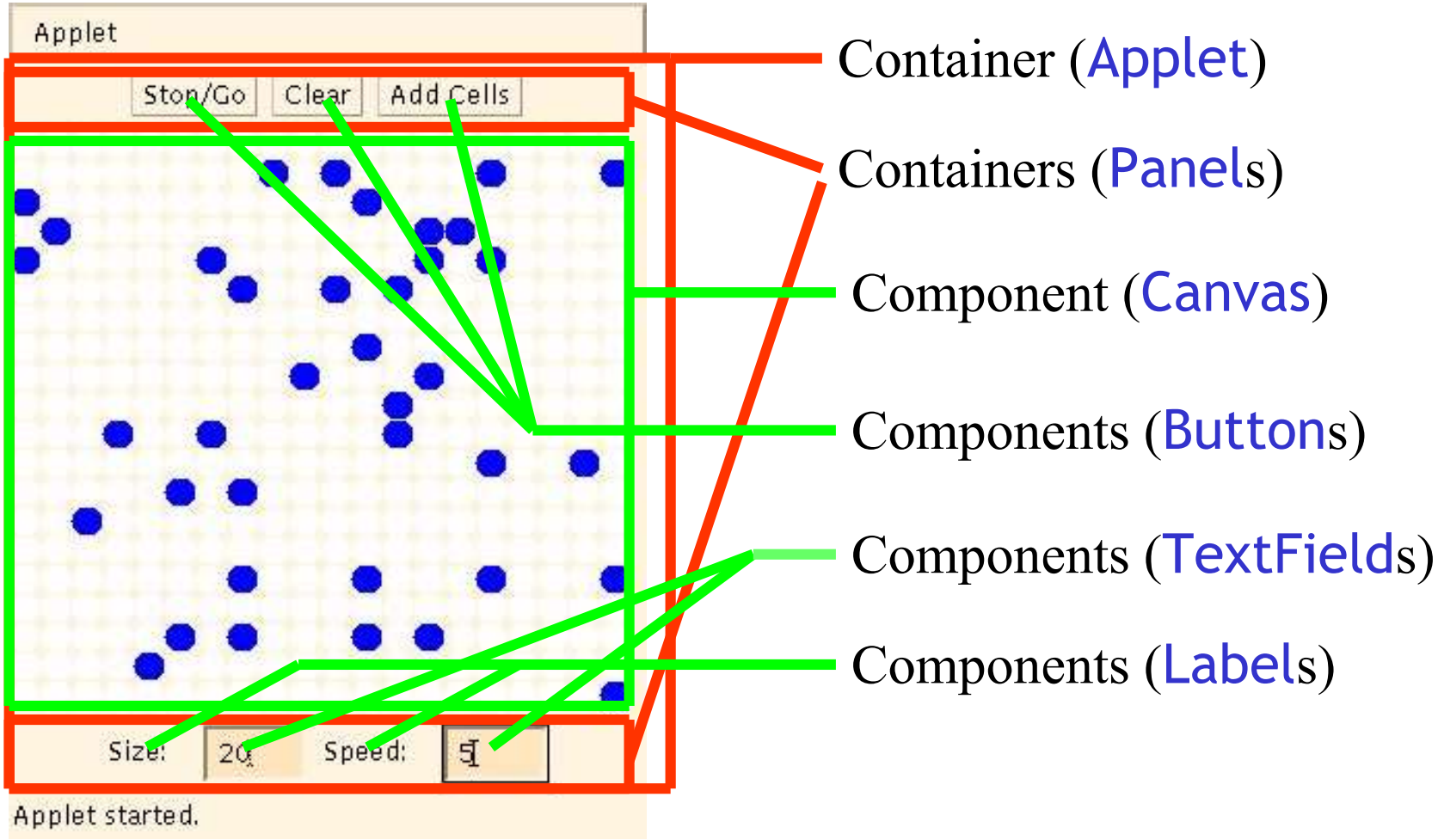
- The job of a Container is to hold and display Components
- Some common subclasses of Component are Button, Checkbox, Label, Scrollbar, TextField, **and** TextArea
- A Container is also a Component
  - This allows Containers to be nested
- Some Container subclasses are Panel (and Applet), Window, and Frame

An Applet is Panel is a Container

```
java.lang.Object
   |
   +----java.awt.Component
          |
          +----java.awt.Container
                 |
                 +----java.awt.Panel
                        |
                        +----java.applet.Applet
```

…so you can display things in an Applet

Example: A "Life" applet



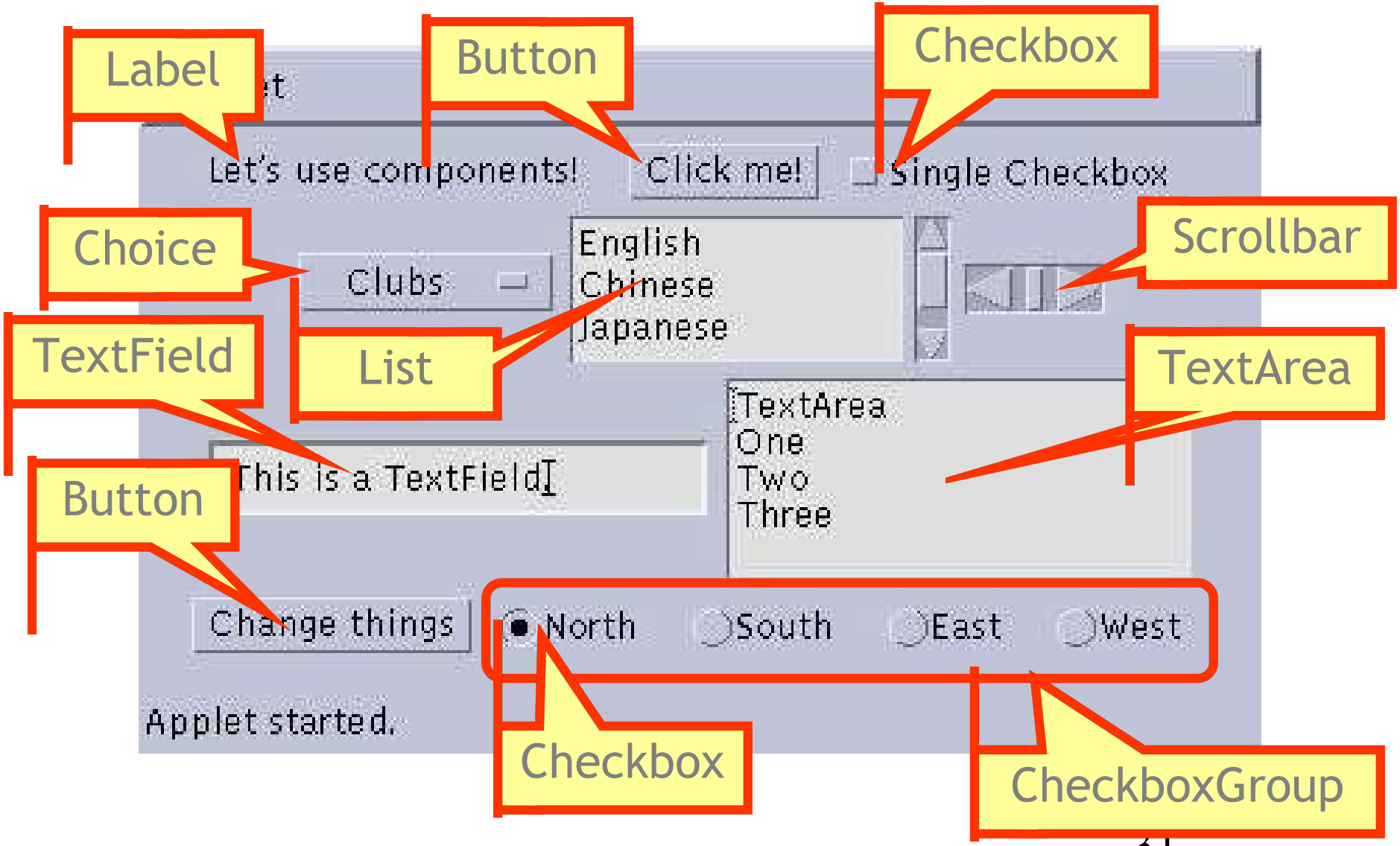
Applets

- An application has a
  `public static void main(String args[ ])`
  method, but an Applet usually does not
- An Applet's main method is in the Browser
- To write an Applet, you extend Applet and override some of its methods
- The most important methods are init( ), start( ), and paint(Graphics g)

To create an applet

- public class MyApplet extends Applet { … }
  - this is the *only* way to make an Applet
- You can add components to the applet
- The best place to add components is in init( )
- You *can* paint directly on the applet, but…
- …it's better to paint on a contained component
- Do all painting from paint(Graphics g)

Some types of components



Creating components

```
Label lab = new Label ("Hi, Dave!");
Button but = new Button ("Click me!");
Checkbox toggle = new Checkbox ("toggle");
TextField txt =
    new TextField ("Initial text.", 20);
Scrollbar scrolly = new Scrollbar
   (Scrollbar.HORIZONTAL, initialValue,
    bubbleSize, minValue, maxValue);
```

Adding components to the Applet

```
class MyApplet extends Applet {
    public void init () {
        add (lab); // same as this.add(lab)
        add (but);
        add (toggle);
        add (txt);
        add (scrolly);
        ...
```

Creating a Frame

- When you create an Applet, you get a Panel "for free"
- When you write a GUI for an *application,* you need to create and use a Frame:
  - Frame frame = new Frame();
  - frame.setTitle("My Frame");
  - frame.setSize(300, 200); // width, height
  - ***... add components ...***
  - frame.setVisible(true);
- Or:
  - class MyClass extends Frame {
  -

Arranging components

- Every Container has a layout manager
- The default layout for a Panel is FlowLayout
- An Applet is a Panel
- Therefore, the default layout for a Applet is FlowLayout
- You could set it explicitly with
  `setLayout (new FlowLayout( ));`
- You could change it to some other layout manager

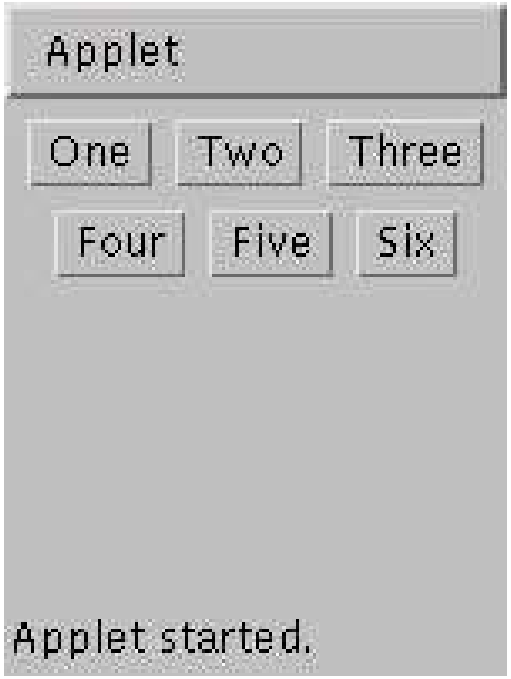
FlowLayout

- Use `add(`*component*`);` to add to a component when using a FlowLayout
- Components are added left-to-right
- If no room, a new row is started
- Exact layout depends on size of Applet
- Components are made as small as possible
- FlowLayout is convenient but often ugly

Complete example: FlowLayout

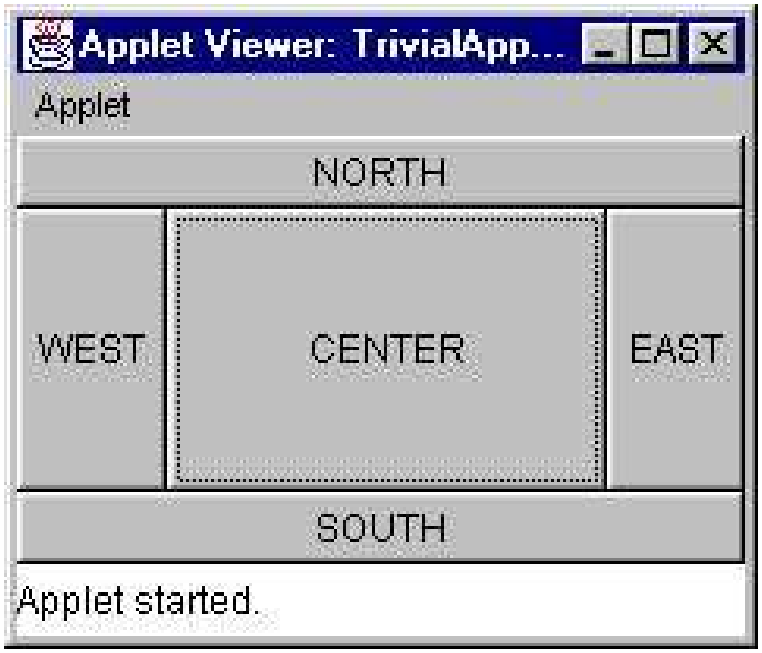
```
import java.awt.*;
import java.applet.*;

public class FlowLayoutExample extends Applet {
  public void init () {
    setLayout (new FlowLayout ()); // default
    add (new Button ("One"));
    add (new Button ("Two"));
    add (new Button ("Three"));
    add (new Button ("Four"));
    add (new Button ("Five"));
    add (new Button ("Six"));
  }
}
```



BorderLayout

- At most five components can be added
- If you want more components, add a Panel, then add components to it.
- setLayout (new BorderLayout());




```
add (new Button("NORTH"), BorderLayout.NORTH);
```

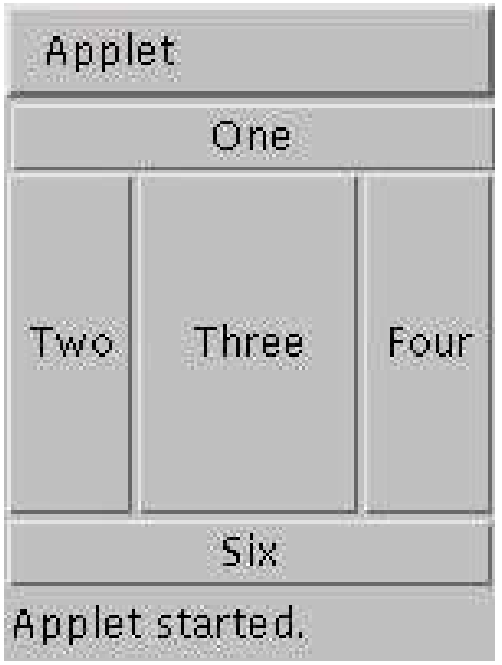
BorderLayout with five Buttons

```
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);
}
```

Complete example: BorderLayout

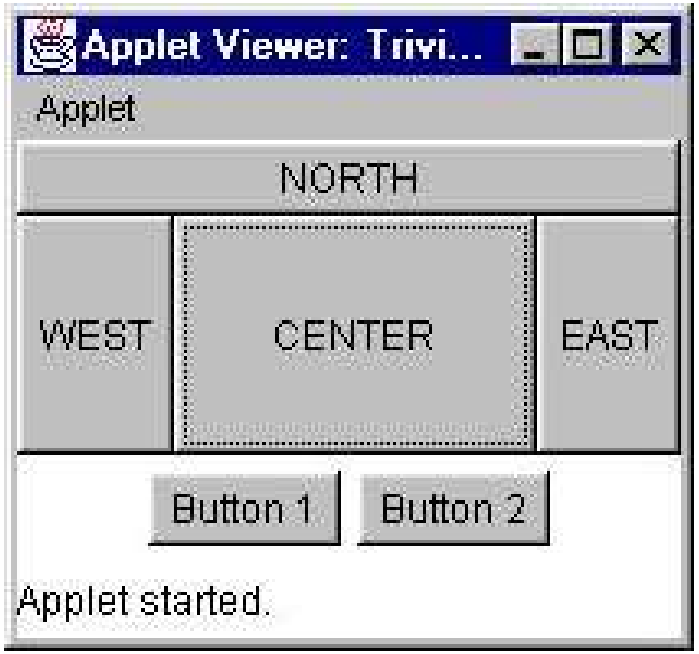
```
import java.awt.*;
import java.applet.*;

public class BorderLayoutExample extends Applet {
    public void init () {
        setLayout (new BorderLayout());
        add(new Button("One"), BorderLayout.NORTH);
        add(new Button("Two"), BorderLayout.WEST);
        add(new Button("Three"), BorderLayout.CENTER);
        add(new Button("Four"), BorderLayout.EAST);
        add(new Button("Five"), BorderLayout.SOUTH);
        add(new Button("Six"), BorderLayout.SOUTH);
    }
}
```



Using a Panel

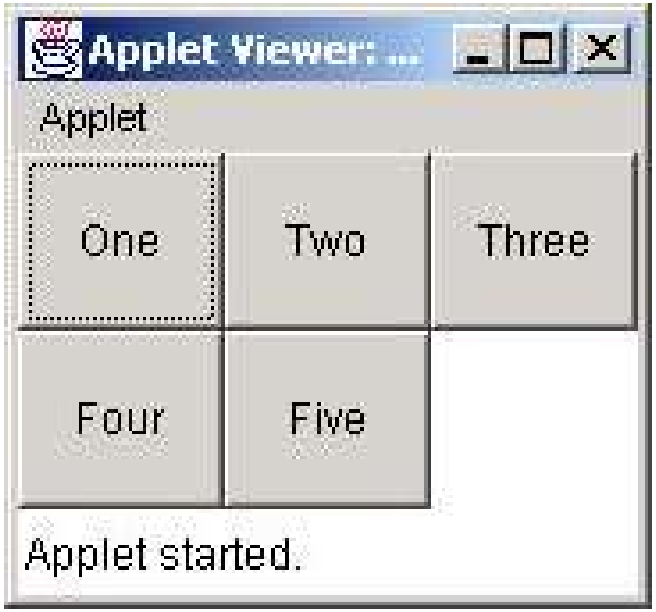
```
Panel p = new Panel();
add (p, BorderLayout.SOUTH);
p.add (new Button ("Button 1"));
p.add (new Button ("Button 2"));
```



GridLayout

- The GridLayout manager divides the container up into a given number of rows and columns:

  new GridLayout(*rows, columns*)

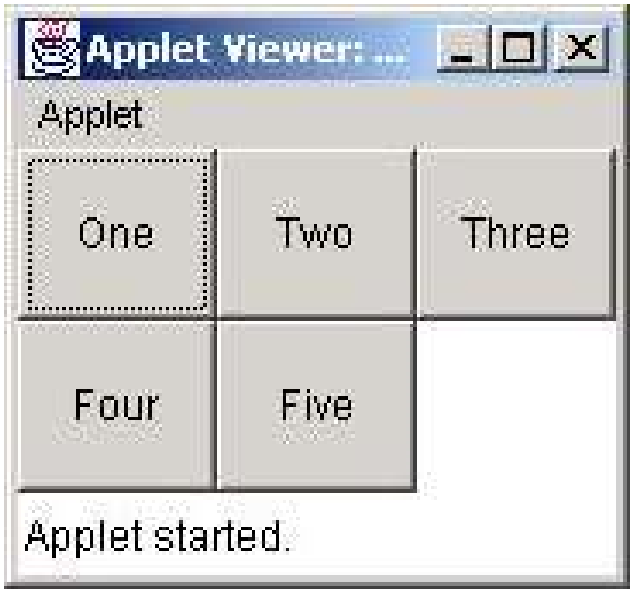


- All sections of the grid are equally sized and as large as possible

Complete example: GridLayout

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

public class GridLayoutExample extends Applet {
  public void init () {
    setLayout(new GridLayout(2, 3));
    add(new Button("One"));
    add(new Button("Two"));
    add(new Button("Three"));
    add(new Button("Four"));
    add(new Button("Five"));
  }
}
```



Making components active

- Most components already *appear* to do something--buttons click, text appears
- To associate an action with a component, attach a *listener* to it
- Components send events, listeners listen for events
- Different components may send different events, and require different listeners

Listeners

- Listeners are interfaces, not classes
  - class MyButtonListener implements ActionListener {
- An interface is a group of methods that *must* be supplied
- When you say implements, you are *promising* to supply those methods

Writing a Listener

- For a Button, you need an ActionListener

```
b1.addActionListener
      (new MyButtonListener ( ));
```

- An ActionListener must have an actionPerformed(ActionEvent) method

```
public void actionPerformed(ActionEvent e) {
    …
}
```

MyButtonListener


```
public void init () {
   ...
   b1.addActionListener (new MyButtonListener ());
}

 class MyButtonListener implements ActionListener {
   public void actionPerformed (ActionEvent e) {
     showStatus ("Ouch!");
   }
 }
```

Listeners for TextFields

- An ActionListener listens for someone hitting the Enter key
- An ActionListener requires this method:

  public void actionPerformed (ActionEvent e)
- You can use getText( ) to get the text

- A TextListener listens for any and all keys
- A TextListener requires this method:

  public void textValueChanged(TextEvent e)

When *not* to use listeners

- Some GUI elements are active—the user expects them to do something
  - Buttons and some menu items are active
  - When an active element is used, the program should do something, and *should look like it's done something!*
- Most GUI elements are passive
  - Text fields, text areas, checkboxes, radio buttons, pulldown lists—these provide data or flags, but don't (shouldn't) cause the program to actually do anything
  - The *appearance* of the passive element changes, but that's all
  - In general, it's a bad idea to put listeners on passive elements
- Active GUI elements should get the state of passive elements as needed
  - For example, clicking a button might cause the program to do `getText()`

AWT and Swing

- AWT Buttons vs. Swing JButtons:
  - A Button is a Component
  - A JButton is an AbstractButton, which is a JComponent, which is a Container, which is a Component
- Containers:
  - Swing uses AWT Containers and AWT Components
- AWT Frames vs. Swing JFrames:
  - A Frame is a Window is a Container is a Component
  - A JFrame is a Frame, etc.
- Layout managers:
  - Swing uses the AWT layout managers, plus a couple of its own
- Listeners:
  - Swing uses many of the AWT listeners, plus a couple of its own

- Bottom line: Not only is there a lot of similarity between AWT and Swing, but Swing actually uses much of the AWT

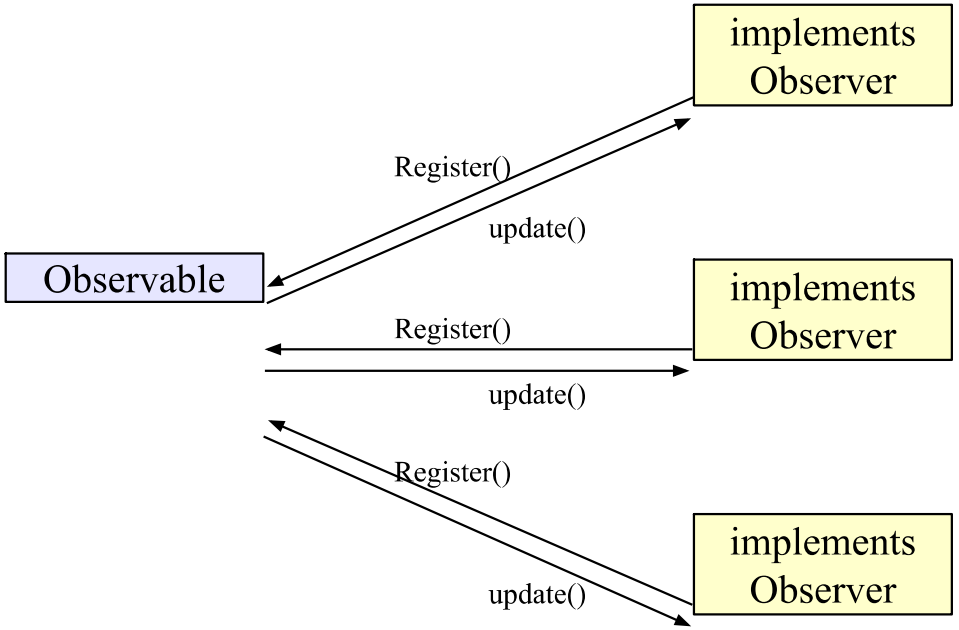
Design Constraints

- The Presentation layer is responsible for display
  - The presentation layer contains objects whose responsibilities focus upon displaying information to the user and receiving input from the user
  - When system entities change, the display must be updated
  - When the user changes the display, the appropriate system entitled must be updated
- However, the Presentation layer must only be loosely coupled to the System entities
  - Presentation layer components are often written by a third party.
  - When GUI components are written, the authors have no concept of the System Entities which are to be displayed
- The Application layer is responsible for processing
  - This layer contains objects whose responsibilities focus on solving the business problem
- The application layer must only be loosely coupled to the Presentation layer objects
  - If the presentation changes, it should not be necessary to change the system entities

Design Pattern: Observer

- To deal with this dilemma, a design pattern called Observer is used
- With this design pattern, there are two types of entities:
  - The Observer
  - The Observable
- The observable entity encapsulates some data of interest
  - When that data changes, the entity notifies all observers watching that data
- An observer is interested in the data
  - Observers register with observables to be notified when the observable's data changes
- To reduce coupling, all communication conforms to a simple interface

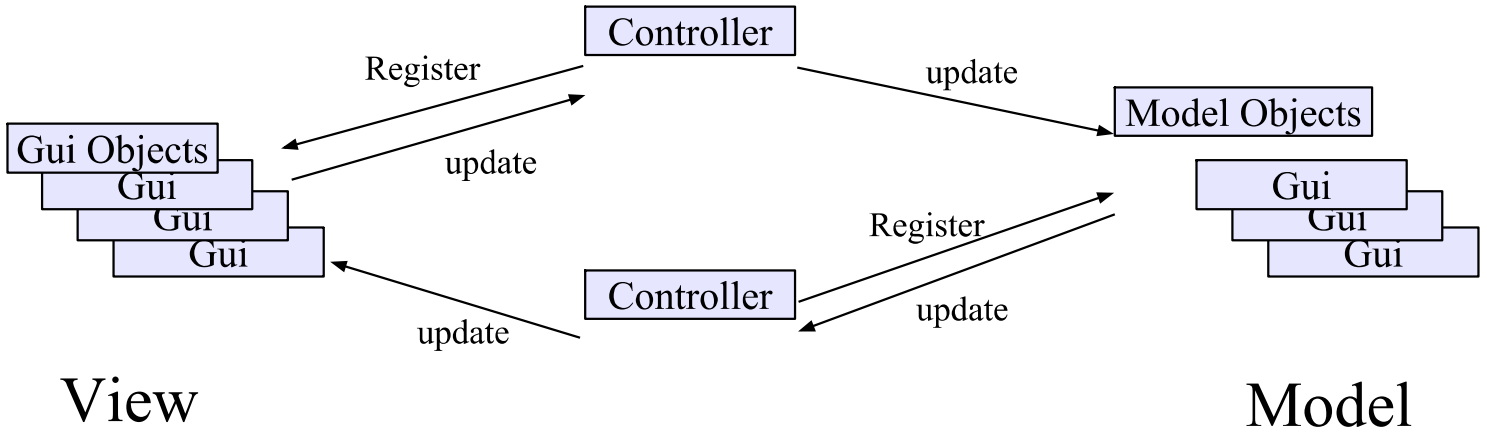
Design Pattern: Observer




```
public interface Observer
{
  public void update();
}
```

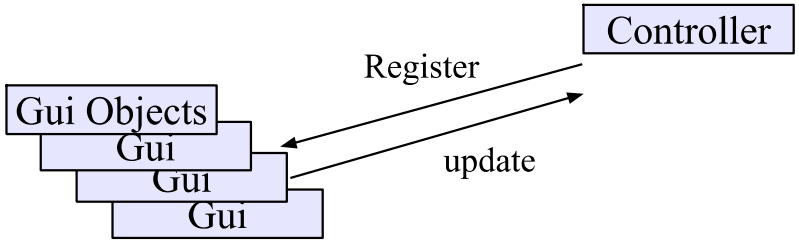
Applicability to the GUI

- A controller object is created to observe the model
  - The model notifies the observer when the state of the model has changed
  - The controller updates the appropriate GUI objects to display the new state of the model
- A controller object is created to observe the GUI
  - The GUI notifies the controller when the state of the GUI has changed
  - The controller updates the appropriate model objects based on the changes made at the GUI.
- This structure is called model-view-controller



Applicability to Java

- Java provides a series of GUI components.
  - Many Java components have Listeners
  - Listeners are equivalent to Observers in the Design Pattern
- Each Listener must implement a Listener interface
  - There are Listener interfaces defined based on what data the listener is looking for (ie. Events)
- When a user does something in the interface, this produces an Event
  - The event is included as a parameter to the method defined in the Listener interface



AWTEvent

- An Event is something that the user does at the GUI
  - Mouse click, key press, etc.
- Events are represented by Event Objects
  - AWTEvent is the abstract superclass to all event objects generated by AWT components
- AWT encapsulates all common features of awt events
  - Each event has an ID
  - Each event has a reference to the source object
  - Each event keeps track of whether it has been consumed or not.
- The following slide lists all of the subclasses of AWTEvent defined in the Java API.

AWT Events

- The following is a list of events in the java.awt.event package:
  - ActionEvent - Action has occurred (eg. button pressed)
  - AdjustmentEvent - "Adjustable" Component changed
  - ComponentEvent - Component Changed
  - ContainerEvent - Container changed (add or remove)
  - FocusEvent - Focus Changed
  - HierarchyEvent - Change in Component Hierarchy
  - InputEvent - Superclass of KeyEvent and MouseEvent
  - InputMethodEvent - Text Input Events
  - ItemEvent - Item Selected or Deselected
  - KeyEvent - Keyboard event
  - MouseEvent - Mouse event
  - PaintEvent - Low level; do not use.
  - TextEvent - Text Changed events
  - WindowEvent - Window related Events

Listener Interfaces

- The Following events have Listeners associated with them:
  - ActionEvent - ActionListener
  - AdjustmentEvent - AdjustmentListener
  - ComponentEvent - ComponentListener
  - ContainerEvent - ContainerListener
  - FocusEvent - FocusListener
  - HierarchyEvent - HierarchyListener
  - InputMethodEvent - InputMethodListener
  - ItemEvent - ItemListener
  - KeyEvent - KeyListener
  - MouseEvent - MouseListener
    - MouseMotionListener
  - TextEvent - TextListener
  - WindowEvent - WindowListener

How to Use Events

- Components defined in the AWT generate AWTEvents
- Examine the API documentation for the component to see what kinds of events it generates
  - Identify which events you wish to listen for (some components generate more than one type of event)
  - Identify the Listener class which Listens for that event
- Once you have identified the Listener interface, implement the Listener interface
  - Listener interfaces are usually implemented by controller classes
- Your listener class must register with the component to received events
  - Call the addXXXListener method where XXX is the Event type.

Events Example: Button

- A check of the documentation for the button class indicates that it generates ActionEvents
  - It provides an addActionListener method
  - Documentation provides links to ActionEvent and ActionListener classes
- The ActionListener interface defines 1 method

```
public void actionPerformed(ActionEvent e)
```

- Create a controller class which implements ActionListener
- Register an instance of your controller class using the addActionListener method of the Button Class

Events Example: ButtonController

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

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

Events Example: FrameController

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

public class FrameController
{
  public static void main(String[] args)
  {
     Frame aFrame = new Frame("Test");
     Button aButton = new Button("Ok");
     ButtonController aController = new ButtonController();

     aButton.addActionListener(aController);

     aFrame.add(aButton);
     aFrame.setVisible(true);
  }
}
```

Components and Events

- Button
  - A push button which contains descriptive text
  - Generates ActionEvents when pressed
- TextField
  - A single line of editable text
  - Generates ActionEvents when the return key is pressed within the field
  - Generates TextEvents when text changes
- TextArea
  - Provides multiple lines of editable text
  - Generates TextEvents when text changes
- Checkbox
  - Provides a checkbox with descriptive text
  - Generates an ItemEvent when its state changes
- Choice
  - Provides a drop-down list of selectable items
  - Generates an ItemEvent when an item is selected

Components and Events

- Frame/Window
  - Represents an on-screen Window
  - Generates WindowEvents
- List
  - A scrollable list of selectable items
  - Generates ActionEvent when and item is double clicked
  - Generates ItemEvents when items are selected or deselected

Components and Events

- Some Events are common to all AWT Components
  - Generate FocusEvents when gaining or losing focus
  - Generates ComponentEvents when the component is shown, hidden, moved or resized
  - Generates HierarchyEvents when its component hierarchy changes
  - Generates KeyEvents when the component has focus and the keyboard is pressed or released.
  - Generates MouseEvents when the mouse is clicked, released or moved within the component. Similarly, generates events when the mouse enters or leaves its bounding box on the screen.

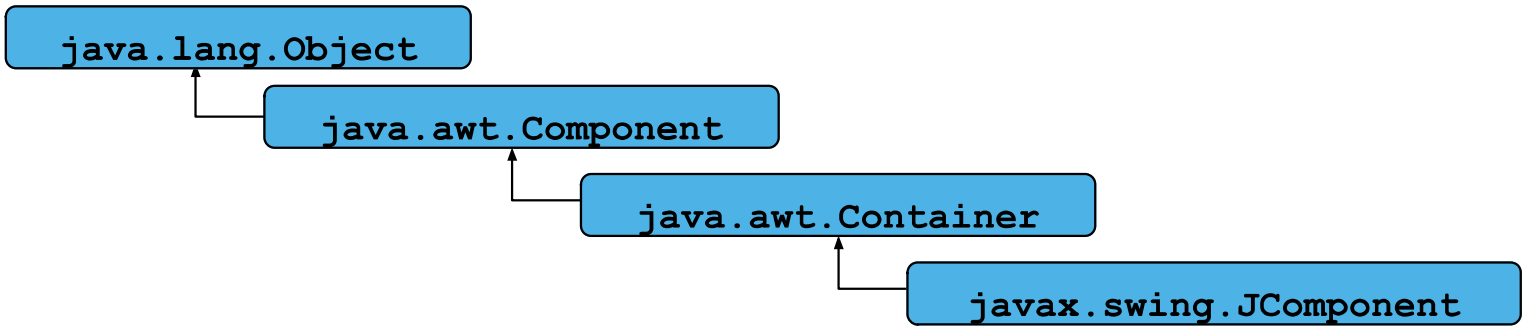
Adapter Classes

- Some Listener interfaces define several methods
  - MouseListener defines 5 methods
  - WindowListener defines 7 methods
- The Java API provides several Adapter classes which provide empty method implementations of the Listener interfaces
  - If a programmer wishes to implement only 1 of the WindowListener methods, he/she can subclass WindowAdapter and override the single method that he/she wishes to provide an implementation for
- Adapter classes provided are:
  - ComponentAdapter
  - ContainerAdapter
  - FocusAdapter
  - KeyAdapter
  - MouseAdapter
  - MouseMotionAdapter
  - WindowAdapter

Swing overview

- Defined in package **`javax.swing`**
- Original GUI components from AWT in **`java.awt`**
  - **Heavyweight** components - rely on local platform's windowing system for look and feel
- Swing components are *lightweight*
  - Not weighed down by GUI capabilities of platform
  - More portable than heavyweight components
- Swing components allow programmer to specify look and feel
  - Can change depending on platform
  - Can be same across all platforms

Swing component inheritance hierarchy



- **Component** defines methods used in its subclasses
  (for example, **paint** and **repaint**)
- **Container** - collection of related components
  - When using **JFrame**, add components to content pane
  (a **Container**)
- **JComponent** - superclass to most Swing components

Jcomponent features

- Pluggable look and feel
  - Can look like different platforms, at run-time
- Shortcut keys (mnemonics)
  - Direct access to components through keyboard
- Common event handling
  - If several components perform same actions
- Tool tips
  - Describe component when mouse rolls over it

Menus

Menu Bar

- JMenuBar()
- add(JMenu)

Menu

- JMenu(String)
- add(JMenuItem)

JMenuItem(String)
JMenuItem(String,int)

```
JMenuBar mb = new JMenuBar();    //create a menu bar
JMenu fileMenu = new JMenu ("File");    //create a menu
mb.add( fileMenu );              //add menu to menu bar
setMenuBar( mb );  // add a menu bar to frame
fileMenu.setMnemonic( KeyEvent.VK_F ); // add a hotkey to menu

JMenuItem miOpen = new JMenuItem( "Open...", KeyEvent.VK_O );
JMenuItem miExit = new JMenuItem( "Exit" );

fileMenu.add( miOpen ); // add a menu item
fileMenu.addSeparator(); // add a menu separator
fileMenu.add( miExit );
```

JLabel

- Labels
  - Provide text instructions on a GUI
  - Read-only text
  - Programs rarely change a label's contents
  - Class **`JLabel`** (subclass of **`JComponent`**)

- Methods
  - Can declare label text in constructor
  - **`myLabel.setToolTipText( "Text" )`**
    - Displays **`"Text"`** in a tool tip when mouse over label
  - **`myLabel.setText( "Text" )`**
  - **`myLabel.getText()`**

JLabel

- **`Icon`**
  - Object that implements interface **`Icon`**

  - One class is **`ImageIcon`** (**`.gif`** and **`.jpeg`** images)

```
24          Icon bug = new ImageIcon( "bug1.gif" );
```

    - Assumed same directory as program
  - Display an icon with **`JLabel`**'s **`setIcon`** method

```
33          label3.setIcon( bug );
```

    - **`myLabel.setIcon( myIcon );`**
    - **`myLabel.getIcon //returns current Icon`**

```
1    // Fig. 12.4: LabelTest.java
2    // Demonstrating the JLabel class.
3    import javax.swing.*;
4    import java.awt.*;
5    import java.awt.event.*;
6
7    public class LabelTest extends JFrame {
8       private JLabel label1, label2, label3;
9
10      public LabelTest()
11      {
12         super( "Testing JLabel" );
13
14         Container c = getContentPane();
15         c.setLayout( new FlowLayout() );
16
17         // JLabel constructor with a string argument
18         label1 = new JLabel( "Label with text" );
19         label1.setToolTipText( "This is label1" );
20         c.add( label1 );
21
22         // JLabel constructor with string,
23         // alignment arguments
24         Icon bug = new ImageIcon( "bug1.gif" );
25         label2 = new JLabel( "Label with text and icon",
26                              bug, SwingConstants.LEFT );
27         label2.setToolTipText( "This is label2" );
28         c.add( label2 );
29
```

Create a **Container** object, to which we attach **JLabel** objects (subclass of **JComponent**).

Initialize text in **JLabel** constructor.

Set the tool tip text, and attach component to **Container c**.

Create a new **ImageIcon** (assumed to be in same directory as program).

Set **ImageIcon** and alignment of text in **JLabel** constructor.

JButton

- Methods of class **JButton**
  - Constructors

```
JButton myButton = new JButton( "Label" );
JButton myButton = new JButton( "Label", myIcon );
```

  - **setRolloverIcon(myIcon)**
    - Sets image to display when mouse over button
- Class **ActionEvent getActionCommand**
    - returns label of button that generated event


```
Icon bug1 = new ImageIcon( "bug1.gif"
);
fancyButton = new JButton( "Fancy
Button", bug1 );
fancyButton.setRolloverIcon( bug2 );
```

JCheckBox

- When **JCheckBox** changes
  - **ItemEvent** generated
    - Handled by an **ItemListener**, which must define **itemStateChanged**
  - Register handlers with with **addItemListener**

```
private class CheckBoxHandler implements ItemListener
    public void itemStateChanged( ItemEvent e )
```

- Class **ItemEvent**
  - **getStateChange**
    - Returns **ItemEvent.SELECTED** or **ItemEvent.DESELECTED**

```
1    // Fig. 12.12: CheckBoxTest.java
2    // Creating Checkbox buttons.
3    import java.awt.*;
4    import java.awt.event.*;
5    import javax.swing.*;
6
7    public class CheckBoxTest extends JFrame {
8       private JTextField t;
9       private JCheckBox bold, italic;
10
11      public CheckBoxTest()
12      {
13         super( "JCheckBox Test" );
14
15         Container c = getContentPane();
16         c.setLayout(new FlowLayout());
17
18         t = new JTextField( "Watch the font style change",
19         t.setFont( new Font( "TimesRoman", Font.PLAIN, 14 )
20         c.add( t );
21
22         // create checkbox objects
23         bold = new JCheckBox( "Bold" );
24         c.add( bold );
25
26         italic = new JCheckBox( "Italic" );
27         c.add( italic );
28
29         CheckBoxHandler handler = new CheckBoxHandler();
30         bold.addItemListener( handler );
```

Create **JCheckBox**es

```
31          italic.addItemListener( handler );
32
33          addWindowListener(
34             new WindowAdapter() {
35                public void windowClosing( WindowEvent e )
36                {
37                   System.exit( 0 );
38                }
39             }
40          );
41
42          setSize( 275, 100 );
43          show();
44       }
45
46       public static void main(
47       {
48          new CheckBoxTest();
49       }
50
51       private class CheckBoxHandler implements Iter
52          private int valBold = Font.PLAIN;
53          private int valItalic = Font.PLAIN;
54
55          public void itemStateChanged( ItemEvent e )
56          {
57             if ( e.getSource() == bold )
58                if ( e.getStateChange() == ItemEvent.SELECTED
59                   valBold = Font.BOLD;
60                else
61                   valBold = Font.PLAIN;
```

Because **CheckBoxHandler** implements **ItemListener**, it must define method **itemStateChanged**

getStateChange returns **ItemEvent.SELECTED** or **ItemEvent.DESELECTED**

JRadioButton

- Radio buttons
  - Have two states: selected and deselected
  - Normally appear as a group
    - Only one radio button in group selected at time
    - Selecting one button forces the other buttons off
  - Mutually exclusive options
  - **ButtonGroup** - maintains logical relationship between radio buttons
- Class **JRadioButton**
  - Constructor
    - **JRadioButton("Label", selected)**
    - If selected **true**, **JRadioButton** initially selected

```
1  // Fig. 12.12: RadioButtonTest.java
2  // Creating radio buttons using ButtonGroup and
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class RadioButtonTest extends JFrame {
8     private JTextField t;
9     private Font plainFont, boldFont,
10                 italicFont, boldItalicFont;
11    private JRadioButton plain, bold, italic, boldItalic;
12    private ButtonGroup radioGroup;
13
14    public RadioButtonTest()
15    {
16       super( "RadioButton Test" );
17
18       Container c = getContentPane();
19       c.setLayout( new FlowLayout() );
20
21       t = new JTextField( "Watch the fo
22       c.add( t );
23
24       // Create radio buttons
25       plain = new JRadioButton( "Plain", true );
26       c.add( plain );
27       bold = new JRadioButton( "Bold", false);
28       c.add( bold );
29       italic = new JRadioButton( "Italic", false );
30       c.add( italic );
```

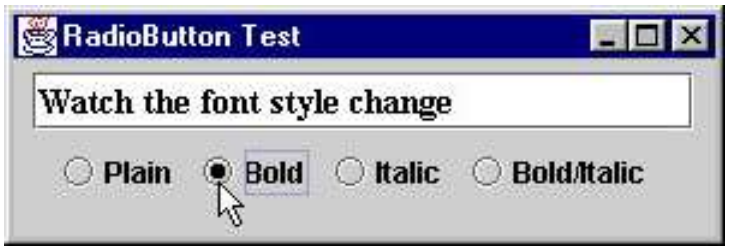
Initialize radio buttons. Only one is initially selected.

```
31          boldItalic = new JRadioButton( "Bold/Italic", false
32          c.add( boldItalic );
33
34          // register events
35          RadioButtonHandler handler = new
36          plain.addItemListener( handler )
37          bold.addItemListener( handler );
38          italic.addItemListener( handler
39          boldItalic.addItemListener( handler );
40
41          // create logical relationship between
42          radioGroup = new ButtonGroup();
43          radioGroup.add( plain );
44          radioGroup.add( bold );
45          radioGroup.add( italic );
46          radioGroup.add( boldItalic );
47
48          plainFont = new Font( "TimesRoman", Font.PLAIN, 14
49          boldFont = new Font( "TimesRoman", Font.BOLD, 14 );
50          italicFont = new Font( "TimesRoman", Font.ITALIC,
51          boldItalicFont =
52             new Font( "TimesRoman", Font.BOLD + Font.ITALIC,
53          t.setFont( plainFont );
54
55          setSize( 300, 100 );
56          show();
57       }
58
```

Create a **ButtonGroup**. Only one radio button in the group may be selected at a time.

Method **add** adds radio buttons to the **ButtonGroup**


RadioButton Test
Watch the font style change
Plain
Bold
Italic
Bold/Italic


RadioButton Test
Watch the font style change
Plain
Bold
Italic
Bold/Italic


RadioButton Test
Watch the font style change
Plain
Bold
Italic
Bold/Italic


RadioButton Test
Watch the font style change
Plain
Bold
Italic
Bold/Italic

JList

- List
  - Displays series of items
  - may select one or more items

- Class **JList**
  - Constructor **JList(arrayOfNames)**
    - Takes array of **Objects** (**String**s) to display in list
  - **setVisibleRowCount(n)**
    - Displays **n** items at a time
    - Does not provide automatic scrolling

```
30          // create a list with the items in the colorNames
31          colorList = new JList( colorNames );
32          colorList.setVisibleRowCount( 5 );
33
34          // do not allow multiple selections
35          colorList.setSelectionMode(
36             ListSelectionModel.SINGLE_SELECTION );
37
38          // add a JScrollPane containing the JList
39          // to the content pane
40          c.add( new JScrollPane( colorList ) );
41
42          // set up event handler
43          colorList.addListSelectionListener(
44             new ListSelectionListener() {
45                public void valueChanged( ListSelectionEvent
46                {
47                   c.setBackground(
48                      colors[ colorList.getSelectedIndex() ]
49                }
50             }
51          );
52
53          setSize( 350, 150 );
54          show();
55       }
56
57       public static void main( String args[] )
58       {
59          ListTest app = new ListTest();
```

Initialize **`JList`** with array of **`String`**s, and show 5 items at a time.

Make the **`JList`** a single-selection list.

Create a new **`JScrollPane`** object, initialize it with a **`JList`**, and attach it to the content pane.

Change the color according to the item selected (use **`getSelectedIndex`**).

```
1  // Fig. 12.20: MouseDetails.java
2  // Demonstrating mouse clicks and
3  // distinguishing between mouse buttons.
4  import javax.swing.*;
5  import java.awt.*;
6  import java.awt.event.*;
7
8  public class MouseDetails extends JFrame {
9     private String s = "";
10    private int xPos, yPos;
11
12    public MouseDetails()
13    {
14       super( "Mouse clicks and buttons" );
15
16       addMouseListener( new MouseClickHandler() );
17
18       setSize( 350, 150 );
19       show();
20    }
21
22    public void paint( Graphics g )
23    {
24       g.drawString( "Clicked @ [" + xPos + ", " + yPos +
25                     xPos, yPos );
26    }
27
```

Another example, illustrating mouse events in AWT and Swing

Add a listener for a mouse click.

```
28      public static void main( String args[] )
29      {
30         MouseDetails app = new MouseDetails();
31
32         app.addWindowListener(
33            new WindowAdapter() {
34               public void windowClosing( WindowEvent e )
35               {
36                  System.exi
37               }
38            }
39         );
40      }
41
42      // inner class to handle mouse events
43      private class MouseClickHandler extends MouseAdapter {
44         public void mouseClicked( MouseEvent e )
45         {
46            xPos = e.getX();
47            yPos = e.getY();
48
49            String s =
50               "Clicked " + e.getClickCount() + " time(s)";
51
52            if ( e.isMetaDown() )      // Right mouse button
53               s += " with right mouse button";
54            else if ( e.isAltDown() )  // Middle mouse
55               s += " with center mouse button";
56            else                       // Left mouse button
57               s += " with left mouse button";
58
```

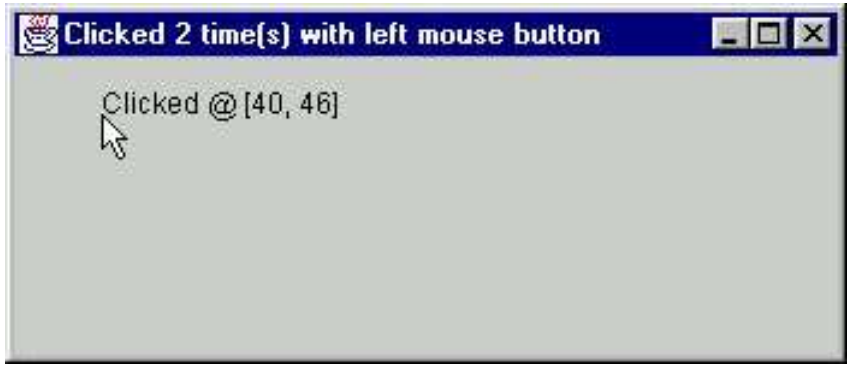
Use a named inner class as the event handler. Can still inherit from MouseAdapter (**`extends MouseAdapter`**).

Use **`getClickCount`**, **`isAltDown`**, and **`isMetaDown`** to determine the **`String`** to use.

```
59              setTitle( s );  // set the title bar of the
60              repaint();
61           }
62        }
63     }
```

Set the Frame's title bar.

Program Output



Good and bad programming practices with AWT

- Separate user interface logic from "business logic" or model
- AWT 1.1 "listeners" designed for this purpose; inner classes facilitate it further
- Separation.java example illustrates this approach:
- class BusinessLogic knows nothing about UI;
- class Separation keeps track of all UI details and talks to BusinessLogic through its public interface.

How is this design loosely coupled?
How does it support reuse?
How does it support legacy code?

- Also note use of inner classes for all "listeners" nested within Separation
- Contrast code of badidea1.java: look at code in actionPerformed:

```
public void actionPerformed(ActionEvent e)
{ Object source = e.getSource();
  if (source == b1)
     System.out.println("Button 1 pressed");
  else if (source == b2) System.out.println("Button 2 pressed");
       else System.out.println("Something else");
}
```

- badidea2.java improves on things by using adapters, but ...
- ***Why is the cascaded*** if ***above a bad idea?***

Eclipse Widgets

- **Standard Widget Toolkit (SWT)**
  - GUI toolkit released in November 2001
  - Initially designed for the Eclipse IDE
  - "Best of both worlds" approach – use native functionality when available, and Java implementation when unavailable
  - Takes on the *appearance* and *behavior* of the native platform
  - The code YOU write will be portable for all the platforms that have SWT implementations
  - http://www.eclipse.org/swt/ - SWT home page

GUI Builders

- Netbeans (Sun)
- JBuilder (Borland)
- Eclipse (IBM and others)
  - Visual Editor
    - Help Software Updates Find and Install…