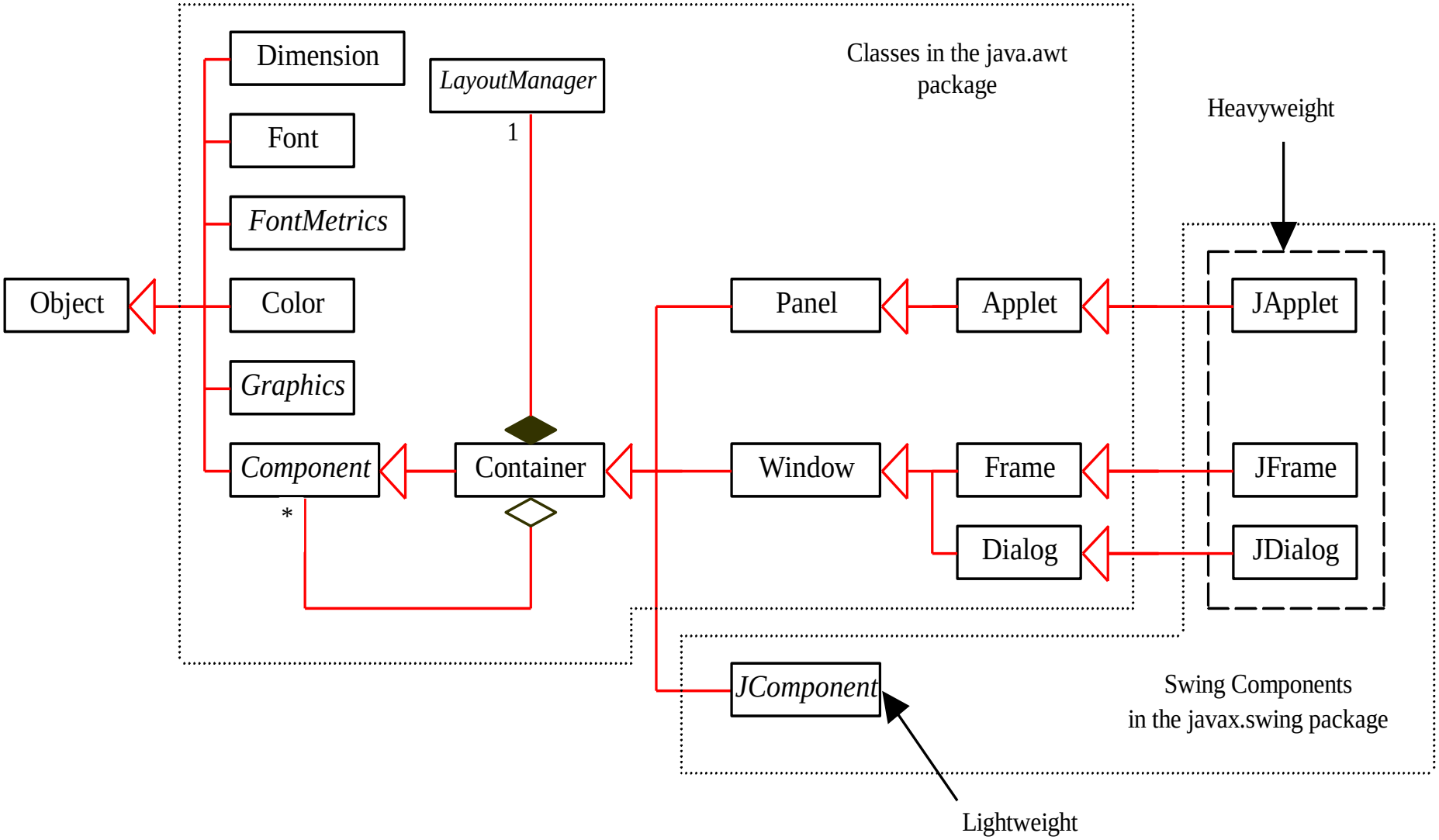
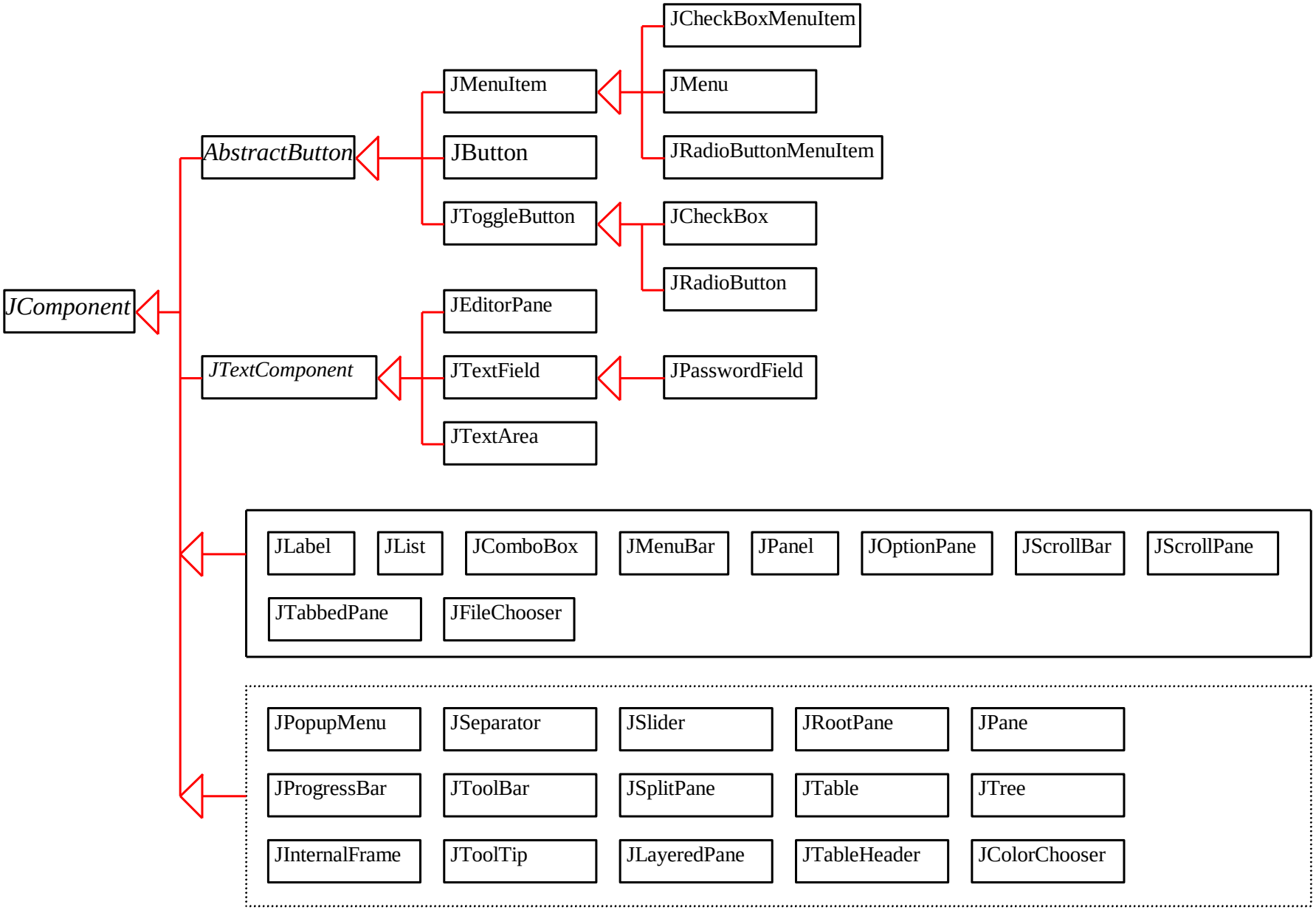
# User Interface Components with Swing

# GUI Class Hierarchy (Swing)



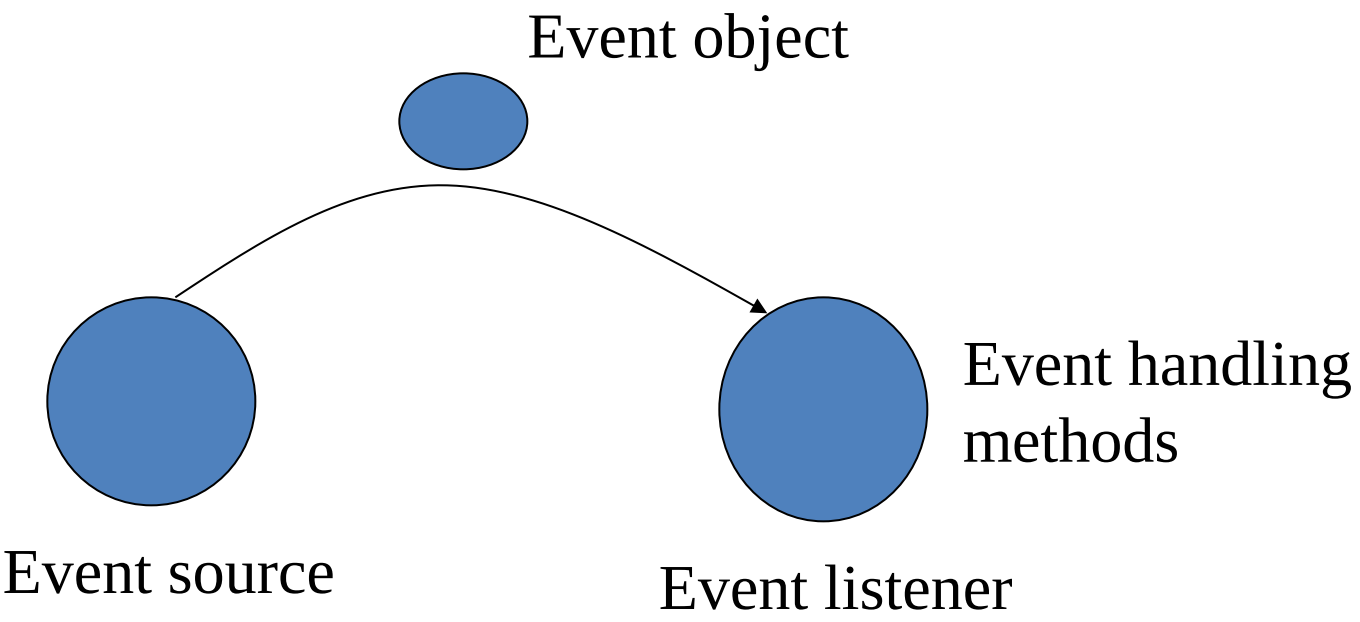
# JComponent


JComponent
AbstractButton
JMenuItem
JCheckBoxMenuItem
JMenu
JRadioButtonMenuItem
JButton
JToggleButton
JCheckBox
JRadioButton
JTextComponent
JEditorPane
JTextField
JPasswordField
JTextArea
JLabel
JList
JComboBox
JMenuBar
JPanel
JOptionPane
JScrollBar
JScrollPane
JTabbedPane
JFileChooser
JPopupMenu
JSeparator
JSlider
JRootPane
JPane
JProgressBar
JToolBar
JSplitPane
JTable
JTree
JInternalFrame
JToolTip
JLayeredPane
JTableHeader
JColorChooser

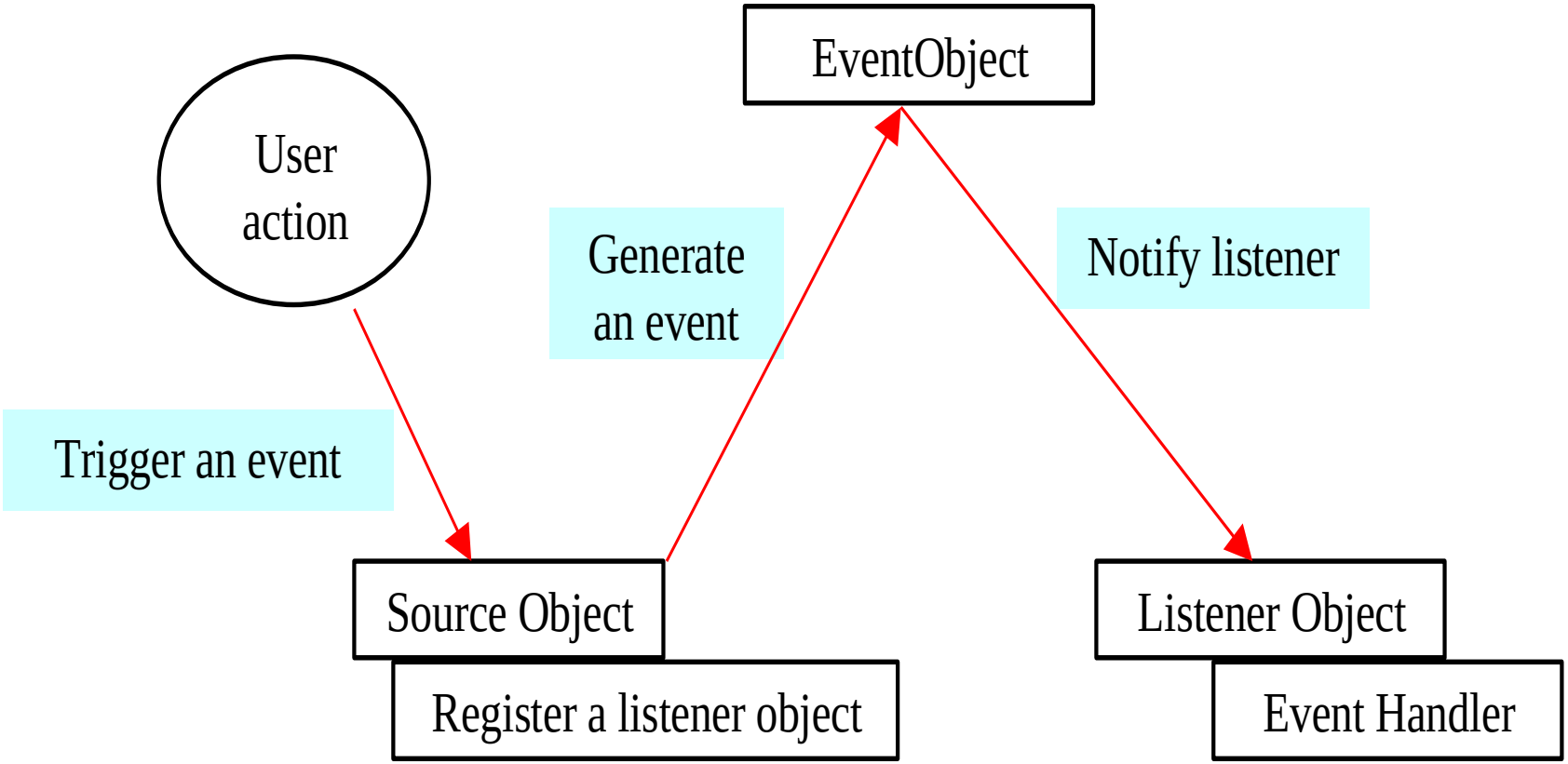
# To Make an Interactive GUI Program

- To make an interactive GUI program, you need:
  - **Components: Event sources**
    - buttons, windows, menus, etc.
  - **Events**
    - mouse clicked, window closed, button clicked, etc.
  - **Event listeners** (interfaces) and **event handlers** (methods)
    - listen for events to be trigged, and then perform actions to handle them

# Event Handling Model



# The Delegation Model



# Objects involved

- **Source**
- **Event**
- **Listener**
  - **Method: Action to be performed**

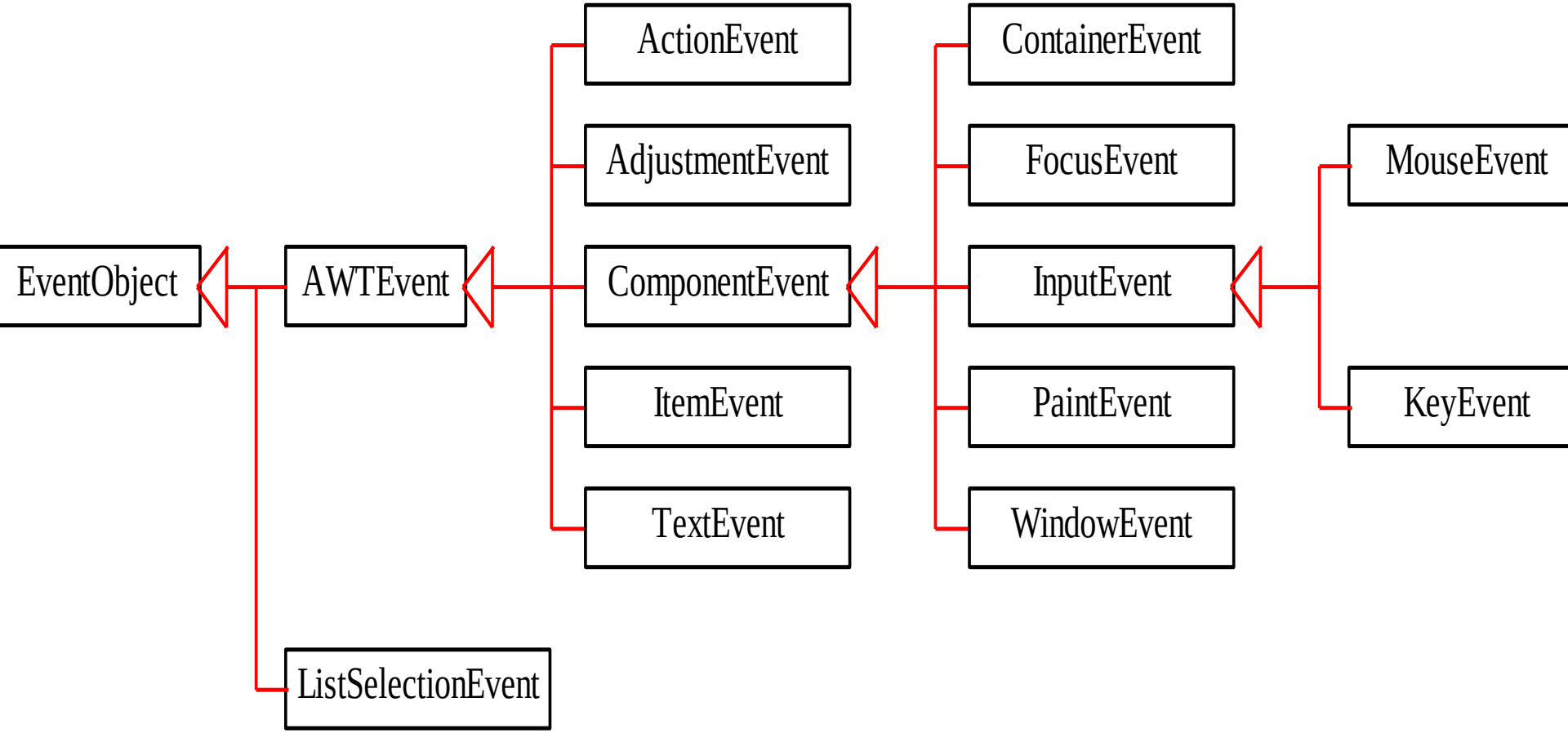
# Event **Sources**

- Possible event sources
  - Button
  - List
  - TextField
  - MenuItem
  - …

# Event Objects

- All Events are objects of Event Classes.
- All Event Classes derive from EventObject.

# Event Classes



# Events

- An *event* can be defined as a type of signal to the program that something has happened.

- The event is generated by external user actions such as mouse movements, mouse button clicks, and keystrokes, or by the operating system, such as a timer.

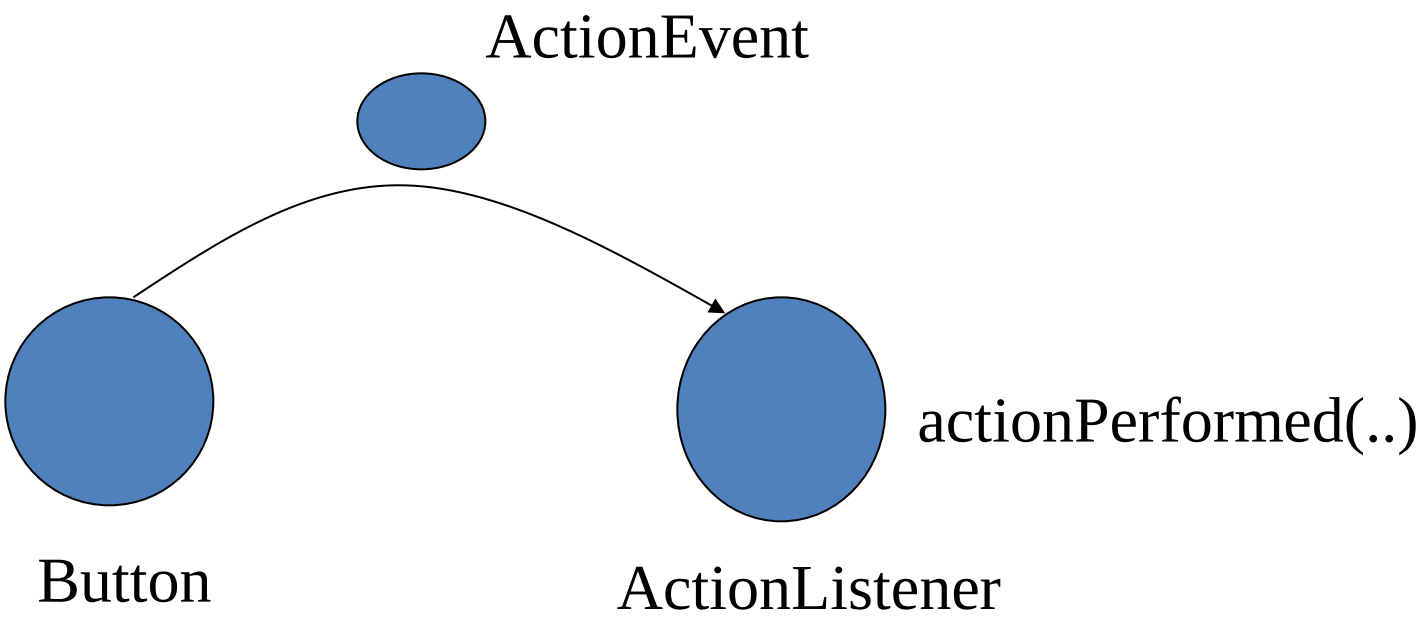
# Event Information

- `id`: A number that identifies the event.
- `target`: The source component upon which the event occurred.
- `arg`: Additional information about the source components.
- `x, y coordinates`: The mouse pointer location when a mouse movement event occurred.
- `clickCount`: The number of consecutive clicks for the mouse events. For other events, it is zero.
- `when`: The time stamp of the event.
- `key`: The key that was pressed or released.

# Selected User Actions

| User Action | Source Object | Event Type Generated |
|---|---|---|
| **Clicked on a button** | **JButton** | **ActionEvent** |
| Changed text | JTextComponent | TextEvent |
| Double-clicked on a list item | JList | ActionEvent |
| Selected or deselected an item with a single click | Jlist | ItemEvent |
| Selected or deselected an item | JComboBox | ItemEvent |

# Action Events on Buttons



# Event Listeners

# Event Listeners

- Listener objects is added to Button
- When the user clicks the button,
- Button object generates an ActionEvent object.
- This calls the listener object's **method** and passes the ActionEvent object generated.

# How to Attach an Event Listener to an Event Source

o is an event source
h is an event listener of type XXX

**o.addXXX(h)**

where XXX is one of the following:

ActionListener
MouseListener
MouseMotionListener
KeyListener
WindowListener

ComponentListener
FocusListener
TextListener
AdjustmentListener
ItemListener

# Registering Event Listeners

- To register a listener object with a source object, you use lines of code that follow the model

```
source.addEventListener(eventListenerObject);
```

# The ActionListener Interface

```
interface ActionListener {
    public  void actionPerformed(ActionEvent e);
}
```

# version 1

```
JButton hw = new JButton("Hello World!");
   panel.add(hw);

   hw.addActionListener(new ActionListener(){
     public void actionPerformed(ActionEvent e){
         System.exit(0);      }    });
```

# version 2

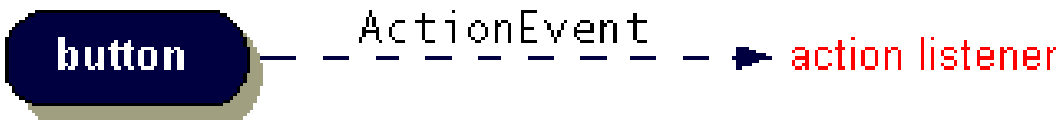
```
class MyFrame extends JFrame implements ActionListener
{  public MyFrame(){
        JButton hw = new JButton("Hello World!");
        panel.add(hw);
        hw.addActionListener(this);

   }

  public void actionPerformed(ActionEvent o){
          System.exit(0);
  }
}
```

# version 3

```
class MyFrame extends Jframe  {
   Button hw;
{  public MyFrame(){
       JButton hw = new JButton("Hello World!");
       panel.add(hw);
       hw.addActionListener(new MyActionListener());   }
}
}

class MyActionListener implements ActionListener {
            public void actionPerformed(ActionEvent o){
                  System.exit(0);
            }
}
```

# Events Handling

- Every time a user types a character or pushes a mouse button, an ***event*** occurs.
- Any object can be notified of an event by registering as an ***event listener*** on the appropriate ***event source***.
- Multiple listeners can register to be notified of events of a particular type from a particular source.



# Event Handling in Java

| **Act that results in the event** | **Listener type** |
|---|---|
| User clicks a button, presses Return while typing in a text field, or chooses a menu item | ActionListener |
| User closes a frame (main window) | WindowListener |
| User presses a mouse button while the cursor is over a component | MouseListener |
| User moves the mouse over a component | MouseMotionListener |
| Component becomes visible | ComponentListener |
| Component gets the keyboard focus | FocusListener |
| Table of list selection changes | ListSelectionListener |

# Implementing an Event Handler

- Implement a listener interface or extend a class that implements a listener interface.
- Register an instance of the event handler class as a listener upon one or more components.
- Implement the methods in the listener interface to handle the event.

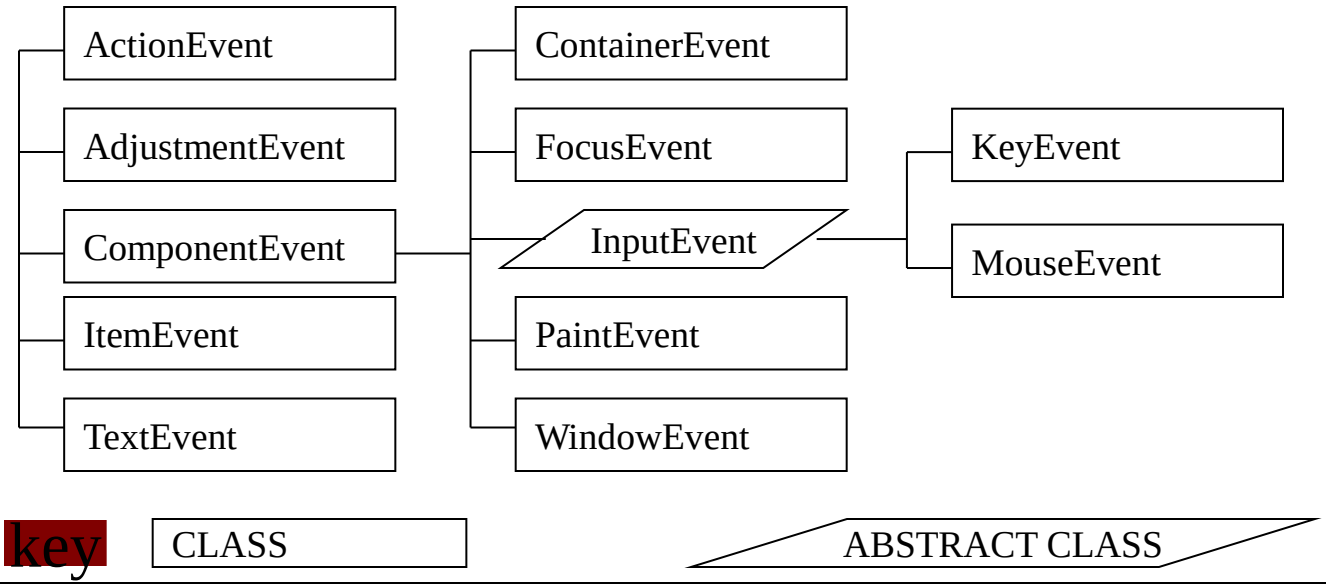
```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class ButtonListener implements ActionListener {
  public void actionPerformed (ActionEvent e) {
     System.out.println ("Got a button press:” + e);
  }
}

public class Main {
  private static void showGUI() {
      JFrame frame = new JFrame("Swing GUI");
      java.awt.Container content = frame.getContentPane();
      content.setLayout(new FlowLayout());
      content.add(new JLabel ("Yo!"));
      JButton button = new JButton ("Click Me");
      button.addActionListener(new ButtonListener());
      content.add(button);
      frame.pack();
      frame.setVisible(true);
   }
```

```
class ButtonListener implements ActionListener {
  public void actionPerformed (ActionEvent e) {
    if (e.getActionCommand().equals ("On")) {
      System.out.println("On!");
    } else if (e.getActionCommand().equals("Off")) {
      System.out.println("Off!");
    } else {
      System.out.println("Unrecognized button press!"); } } }

public class main {
  private static void showGUI() {
      ...
      ButtonListener bl = new ButtonListener();
      JButton onButton = new JButton ("On");
      onButton.addActionListener(bl);
      content.add(onButton);
      JButton offButton = new JButton ("Off");
      offButton.addActionListener(bl);
      content.add(offButton);
```

# Event-Handling Model

- Some GUIs are *event driven* — they generate events when the user interacts with the GUI
  - E.g, moving the mouse, clicking a button, typing in a text field, selecting an item from a menu, etc.
  - When a user interaction occurs, an event is sent to the program. Many event types are defined in packages `java.awt.event` and `javax.swing.event`

some event classes in package `java.awt.event`



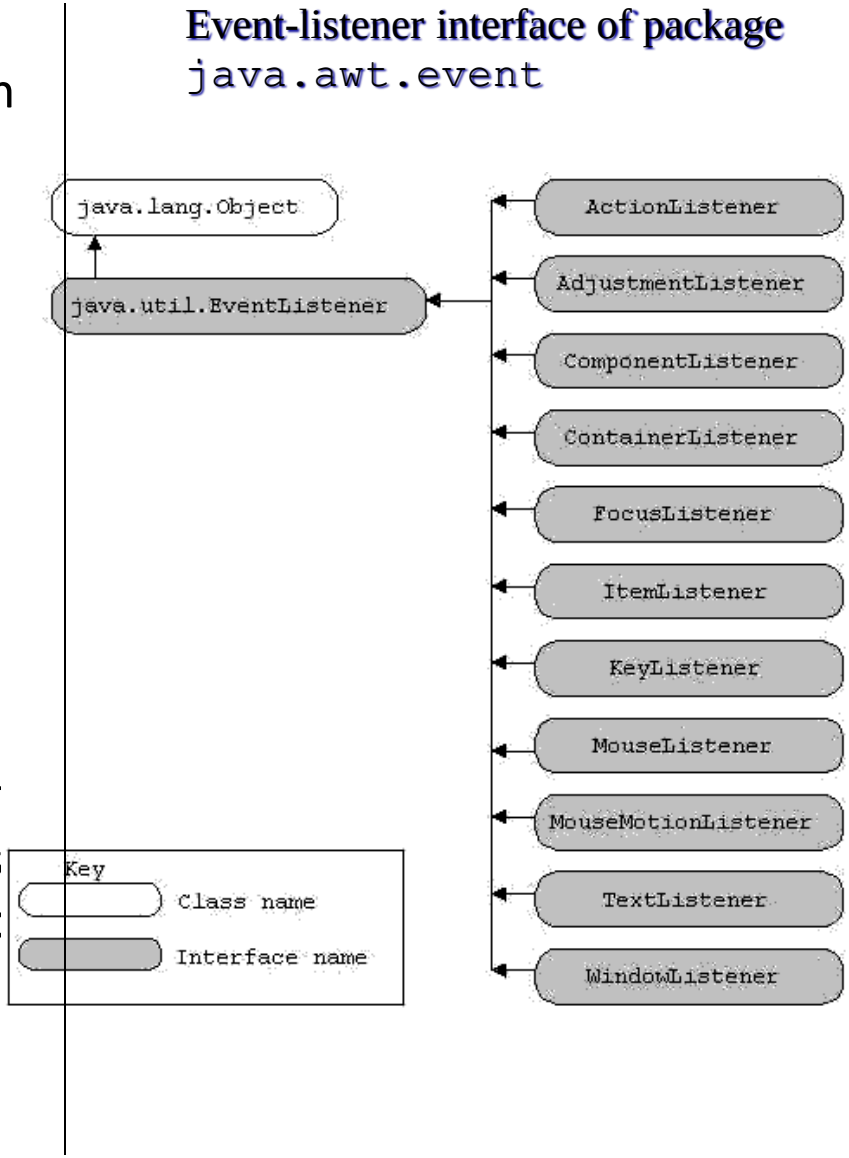
# Event-Handling Model

- Three parts of the event-handling mechanism
  - *event source*: the GUI component with which the user interacts
  - *event object*: encapsulated informatio about the occurred event
  - *event listener*: an object which is notified by the event source when an event occurs, and provides responds to the event
- two tasks to process a GUI event

1. register an *event listener*

- An object of a class that implements or or more of the event-listener interfaces from packages `java.awt.event` and `javax.swing.event`

2. implement an *event handling method*



# Other Listener Classes

- So far, we've only dealt with ActionListeners, which only have 1 method defined, actionPerformed.
- Some Event classes have multiple methods defined,

```
public interface WindowListener {
  void windowOpened(WindowEvent e);
  void windowClosing(WindowEvent e);
  void windowClosed(WindowEvent e);
  void windowIconified(WindowEvent e);
  void windowDeIconified(WindowEvent e);
  void windowActivated(WindowEvent e);
  void windowDeActivated(WindowEvent e);
}
```

# Adapter Classes

- Well, since these are interfaces, it means you have to defined every method listed. So, to add an windowListener to a JFrame, you'd have to

```
JFrame frame=new JFrame();
frame.addWindowListener(new WindowListener()
{
  public void windowOpened(WindowEvent e)
  {
    Log.journal("Frame Opened");
  }
  public void windowClosed(WindowEvent e) {}
  public void windowIconified(WindowEvent e) {}
  //… for every method in the Interface
});
```

# Adapter Classes

- Java provides something called Adapter classes, which will overload these methods automatically to do nothing.  You can then overload the select ones to do what you want

# Adapter Classes

```
public class winAdapter extends WindowAdapter
{
   //WindowAdapter overloads all 7 methods in the
    WindowListner interface
  //Now we overload those we want to do something
   public void windowClosing(WindowEvent e)
   {
     System.exit(0);
   }
}

JFrame ourFrame=new JFrame();
ourFrame.addWindowListener(new winAdapter());
```

# Adapter Classes

- We can even bring this a level further. Just use an anonymous inner class to use the Adapter class.

```
JFrame ourFrame=new JFrame();
ourFrame.addWindowListener(new WindowAdapter()
{
  public void windowClosing(WindowEvent e)
  {
      System.exit(0);
  }
});
```

# MouseListener

**mouseClicked**(MouseEvent)

Invoked when the mouse has been clicked on a component.

**mouseEntered**(MouseEvent)

Invoked when the mouse enters a component.

**mouseExited**(MouseEvent)

Invoked when the mouse exits a component.

**mousePressed**(MouseEvent)

Invoked when a mouse button has been pressed on a component.

**mouseReleased**(MouseEvent)

Invoked when a mouse button has been released on a component.

# MouseAdapter

```
class MouseAdapter implements MouseListener {
        public void mouseClicked(MouseEvent e){}

        public void mouseEntered(MouseEvent e){}

        public void mouseExited(MouseEvent e){}

        public void mousePressed(MouseEvent e){}

        public void mouseReleased(MouseEvent e){}
}
```

# Example

```
 f.addMouseListener(new MouseAdapter(){
 public void mouseClicked(MouseEvent e){
System.out.println("Mouse clicked: ("+e.getX()+","+e.getY()+")");
 }
      ....
}
```

# Listeners

- Listener methods take a corresponding type of event as an argument.
- Event objects have useful methods. For example, getSource returns the object that produced this event.
- A MouseEvent has methods getX, getY.

# Mouse Events

- Mouse events are captured by an object which is a MouseListener and possibly a MouseMotionListener.
- A mouse listener is usually attached to a JPanel component.
- It is not uncommon for a panel to serve as its own mouse listener:

```
addMouseListener(this);
addMouseMotionListener(this); // optional
```

# Mouse Events (cont'd)

- Mouse listener methods receive a MouseEvent object as a parameter.
- A mouse event can provide the coordinates of the event and other information:

```
public void mousePressed(MouseEvent e)
{
    int x = e.getX();
    int y = e.getY();
    int clicks = e.getClickCount();
}
```

# Mouse Events (cont'd)

- The MouseListener interface defines five methods:

```
void mousePressed (MouseEvent e)
void mouseReleased (MouseEvent e)
void mouseClicked (MouseEvent e)
void mouseEntered (MouseEvent e)
void mouseExited (MouseEvent e)
```

Called when the mouse cursor enters/exits component's visible area

- One click and release causes several calls. Using only mouseReleased is usually a safe bet.

# Mouse Events (cont'd)

- The MouseMotionListener interface adds two methods:

```
void mouseMoved (MouseEvent e)
void mouseDragged (MouseEvent e)
```

Called when the mouse has moved with a button held down

- These methods are usually used together with MouseListener methods (i.e., a class implements both interfaces).

# Event Handling Strategies: Pros and Cons

- Separate Listener
  - Advantages
    - Can extend adapter and thus ignore unused methods
    - Separate class easier to manage
  - Disadvantage
    - Need extra step to call methods in main window
- Main window that implements interface
  - Advantage
    - No extra steps needed to call methods in main window
  - Disadvantage
    - Must implement methods you might not care about

# Event Handling Strategies: Pros and Cons, cont.

- Named inner class
  - Advantages
    - Can extend adapter and thus ignore unused methods
    - No extra steps needed to call methods in main window
  - Disadvantage
    - A bit harder to understand
- Anonymous inner class
  - Advantages
    - Same as named inner classes
    - Even shorter
  - Disadvantage
    - Much harder to understand

# Drawing on Panels

# Drawing on Panels

- JPanel can be used to draw graphics (including text) and enable user interaction.

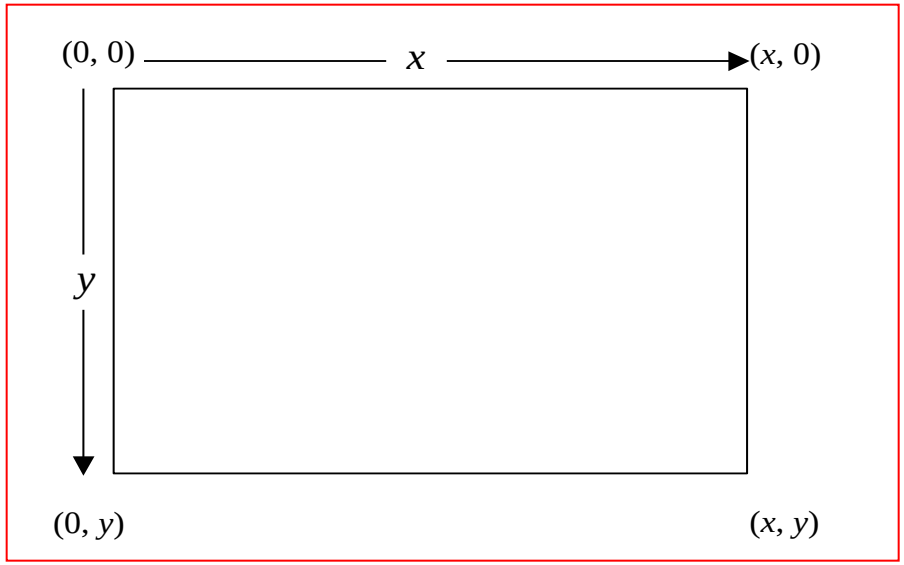
- To draw in a panel, you create a new class that **extends JPanel** and **override the paintComponent** method to tell the panel how to draw things.

- You can then display strings, draw geometric shapes, and view images on the panel.

# Drawing on Panels, cont.

```
public class DrawMessage extends JPanel {
  /** Main method */
  public static void main(String[] args) {
    JFrame frame = new JFrame("DrawMessage");
    frame.getContentPane().add(new DrawMessage());
   frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   frame.setSize(300, 200);
   frame.setVisible(true);
  }

  /** Paint the message */
  public void paintComponent(Graphics g) {
    super.paintComponent(g);

    g.drawString("Welcome to Java!", 40, 40);
  }
}
```

# Drawing on Panels, cont.

# Graphics Class

- The Graphics class is an abstract class for displaying figures and images on the screen on different platforms.
- paintComponent method to draw things on a graphics context g,
- g is an instance of a concrete subclass of the abstract Graphics class for the specific platform.

- g = comp.getGraphics();

- Whenever a component is displayed, a Graphics object is created for the component.

- paintComponent method to draw things.

- super.paintComponent(g) is necessary to ensure that the viewing area is cleared before a new drawing is displayed.

# The `Color` Class

You can set colors for GUI components by using the <u>java.awt.Color</u> class. Colors are made of red, green, and blue components, each of which is represented by a byte value that describes its intensity, ranging from 0 (darkest shade) to 255 (lightest shade). This is known as the *RGB model*.

```
Color c = new Color(r, g, b);
```

`r`, `g`, and `b` specify a color by its red, green, and blue components.

Example:

```
Color c = new Color(228, 100, 255);
```

# Setting Colors

You can use the following methods to set the component's background and foreground colors:

```
setBackground(Color c)

setForeground(Color c)
```

Example:

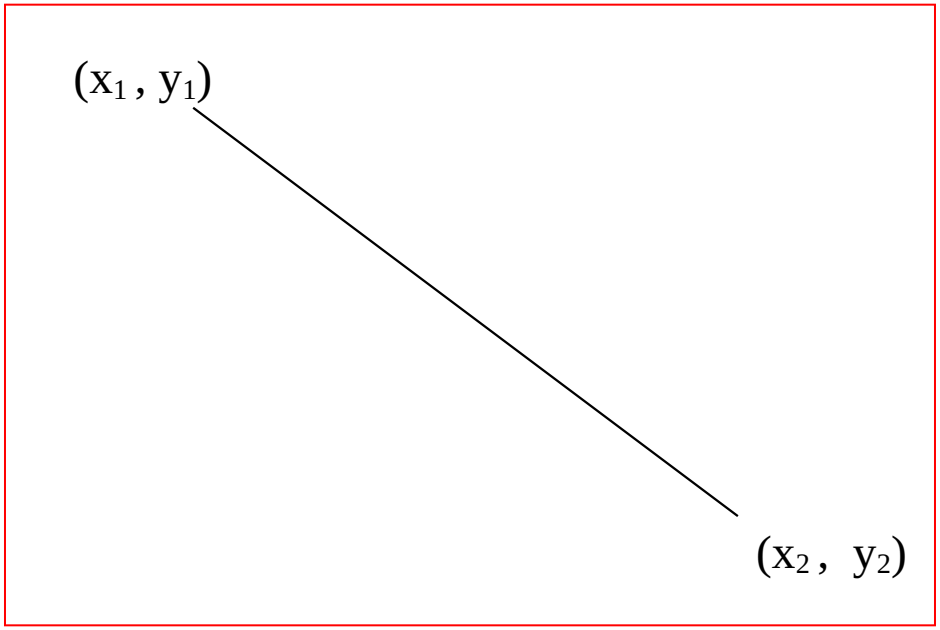
```
setBackground(Color.yellow);
setForeground(Color.red);
```

# Drawing Geometric Figures

- Drawing Lines
- Drawing Rectangles
- Drawing Ovals
- Drawing Arcs
- Drawing Polygons

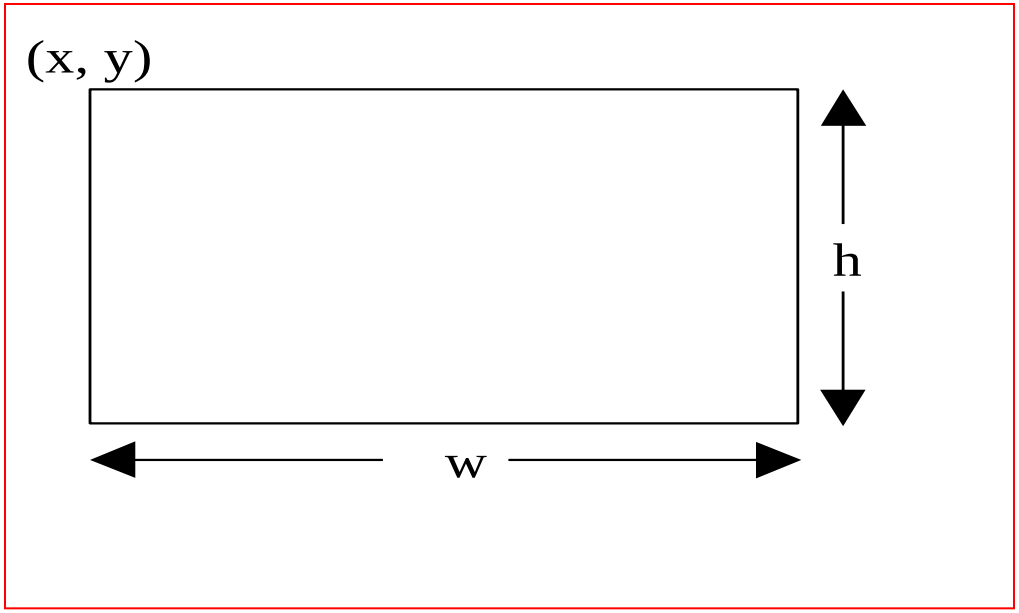
# Drawing Lines

```
drawLine(x1, y1, x2, y2);
```



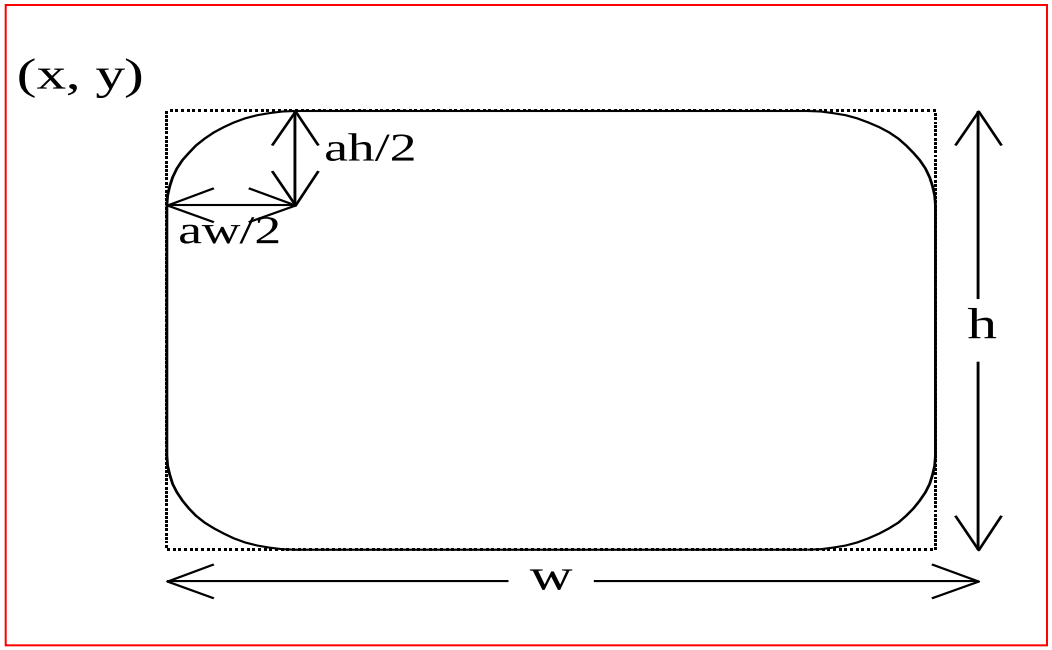
# Drawing Rectangles

- `drawRect(x, y, w, h);`
- `fillRect(x, y, w, h);`



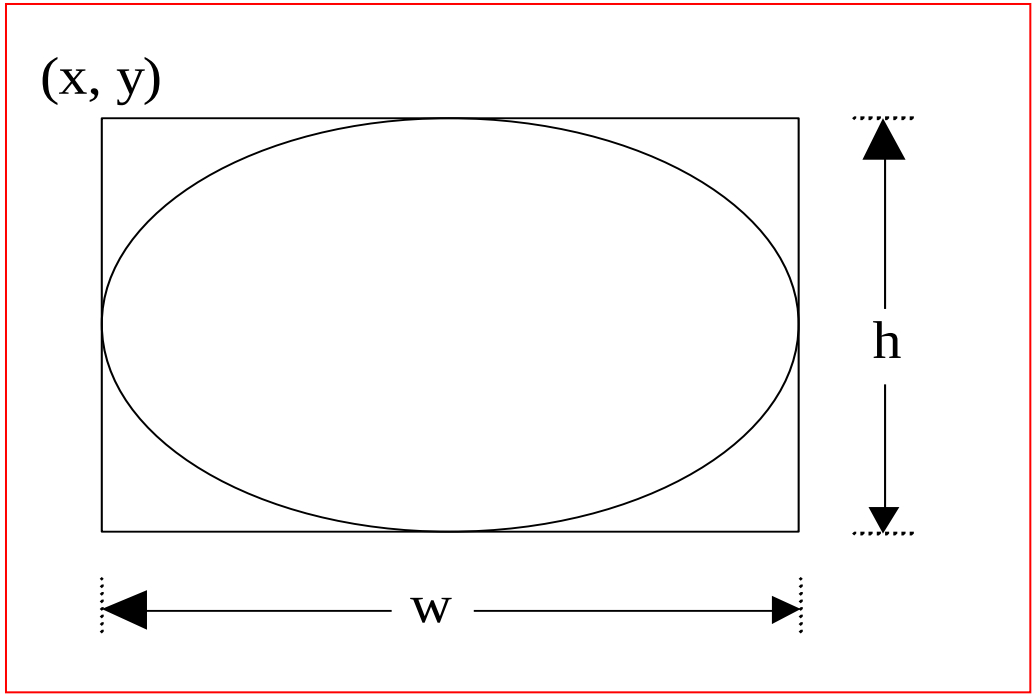
# Drawing Rounded Rectangles

- `drawRoundRect(x, y, w, h, aw, ah);`
- `fillRoundRect(x, y, w, h, aw, ah);`



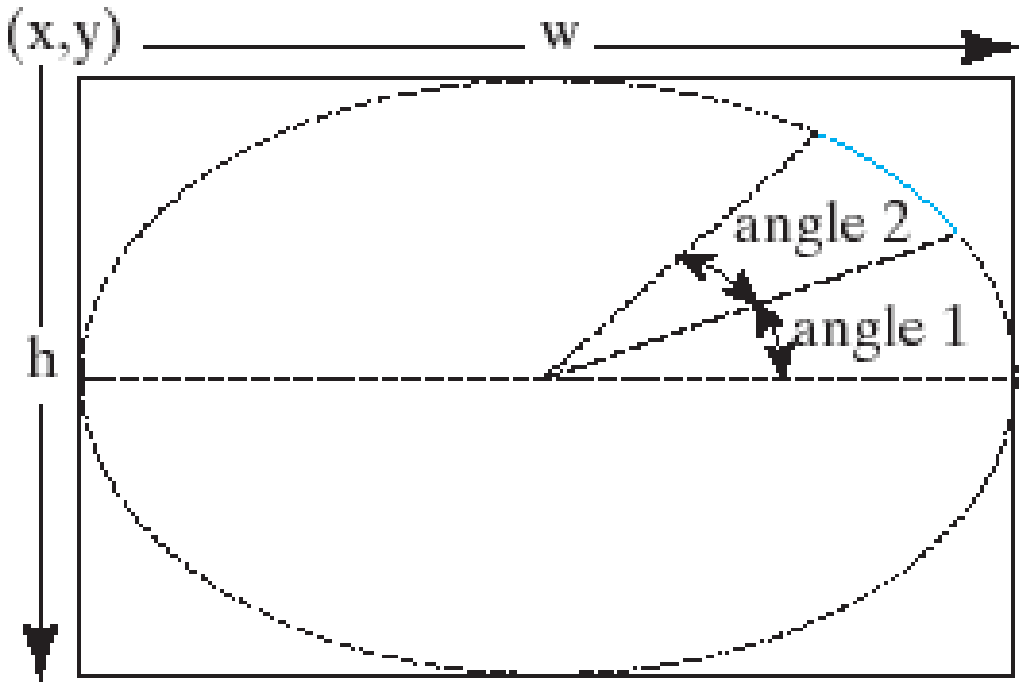
# Drawing Ovals

- `drawOval(x, y, w, h);`
- `fillOval(x, y, w, h);`



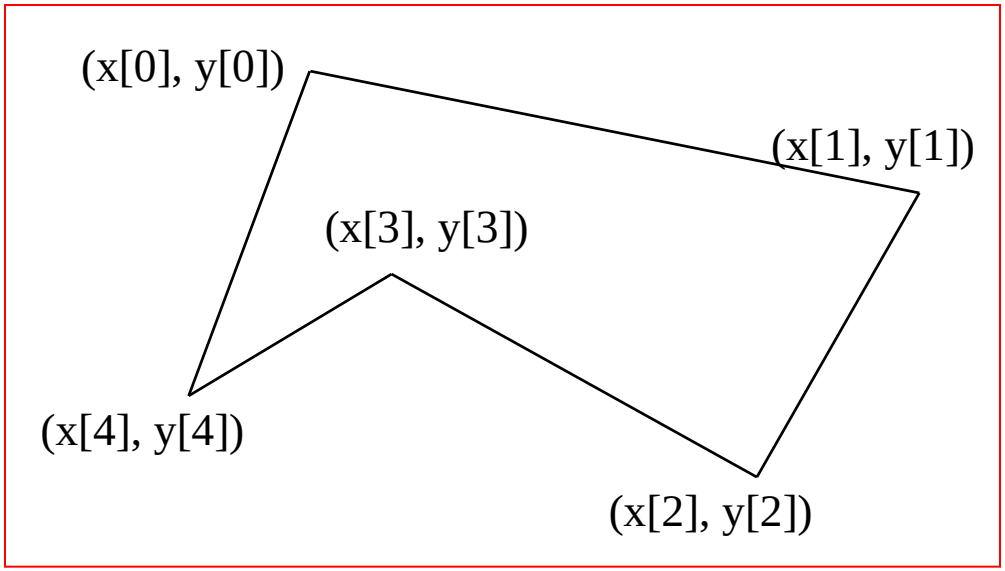
# Drawing Arcs

- `drawArc(x, y, w, h, angle1, angle2);`
- `fillArc(x, y, w, h, angle1, angle2);`



# Drawing Polygons

```
int[] x = {40, 70, 60, 45, 20};
int[] y = {20, 40, 80, 45, 60};
g.drawPolygon(x, y, x.length);
g.fillPolygon(x, y, x.length);
```



# Additional slides: Not for evaluation

# Keyboard Events

- Keyboard events are captured by an object which is a KeyListener.
- A key listener object must first obtain keyboard "focus." This is done by calling the component's requestFocus method.
- If keys are used for moving objects (as in a drawing program), the "canvas" panel may serve as its own key listener:

```
addKeyListener(this);
```

# Keyboard Events (cont'd)

- The KeyListener interface defines three methods:

```
void keyPressed (KeyEvent e)
void keyReleased (KeyEvent e)
void keyTyped (KeyEvent e)
```

- One key pressed and released causes several calls.

# Keyboard Events (cont'd)

- Use keyTyped to capture character keys (i.e., keys that correspond to printable characters).
- e.getKeyChar() returns a char, the typed character:

```
public void keyTyped (KeyEvent  e)
{
   char ch = e.getKeyChar();
   if (ch == 'A')
        ...
}
```

# Keyboard Events (cont'd)

- Use keyPressed or keyReleased to handle "action" keys, such as cursor keys, <Enter>, function keys, and so on.
- e.getKeyCode() returns an int, the key's "virtual code."
- The KeyEvent class defines constants for numerous virtual keys. For example:

| | |
|---|---|
| VK_LEFT, VK_RIGHT, VK_UP, VK_DOWN | Cursor keys |
| VK_HOME, VK_END, VK_PAGE_UP, ...etc. | Home, etc. |

# Keyboard Events (cont'd)

- e.isShiftDown(), e.isControlDown(), e.isAltDown() return the status of the respective modifier keys.
- e.getModifiers() returns a bit pattern that represents the status of all modifier keys.
- KeyEvent defines "mask" constants CTRL_MASK, ALT_MASK, SHIFT_MASK, and so on.

# Whiteboard: Adding Keyboard Events

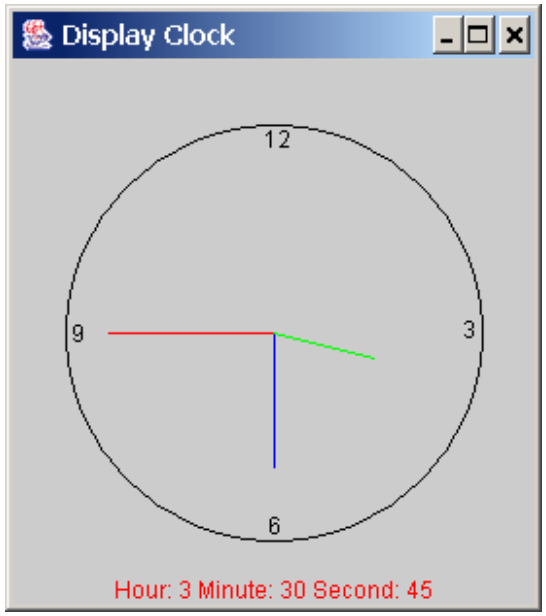
```
Font font = new Font("Serif", Font.BOLD, 20);
    setFont(font);
    fm = getFontMetrics(font);
    addKeyListener(new CharDrawer());
```

# Whiteboard

```
  ...
  private class CharDrawer extends KeyAdapter {
    // When user types a printable character,
    // draw it and shift position rightwards.
    public void keyTyped(KeyEvent event) {
      String s = String.valueOf(event.getKeyChar());
      getGraphics().drawString(s, lastX, lastY);
      record(lastX + fm.stringWidth(s), lastY);
    }
  }
}
```

# Drawing a Clock

- Objective: Use drawing and trigonometric methods to draw a clock showing the specified hour, minute, and second in a frame.



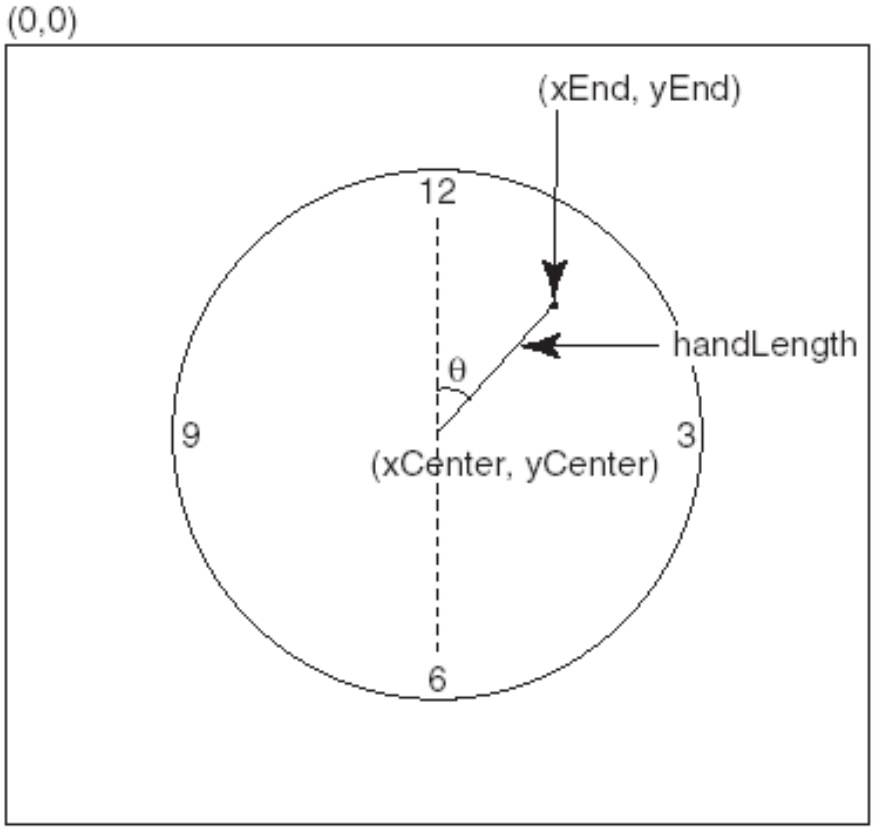
# Drawing Clock

$$xEnd = xCenter + handLength \times \sin(\theta)$$

$$yEnd = yCenter - handLength \times \cos(\theta)$$

Since there are sixty seconds in one minute, the angle for the second hand is

$$second \times (2\pi/60)$$



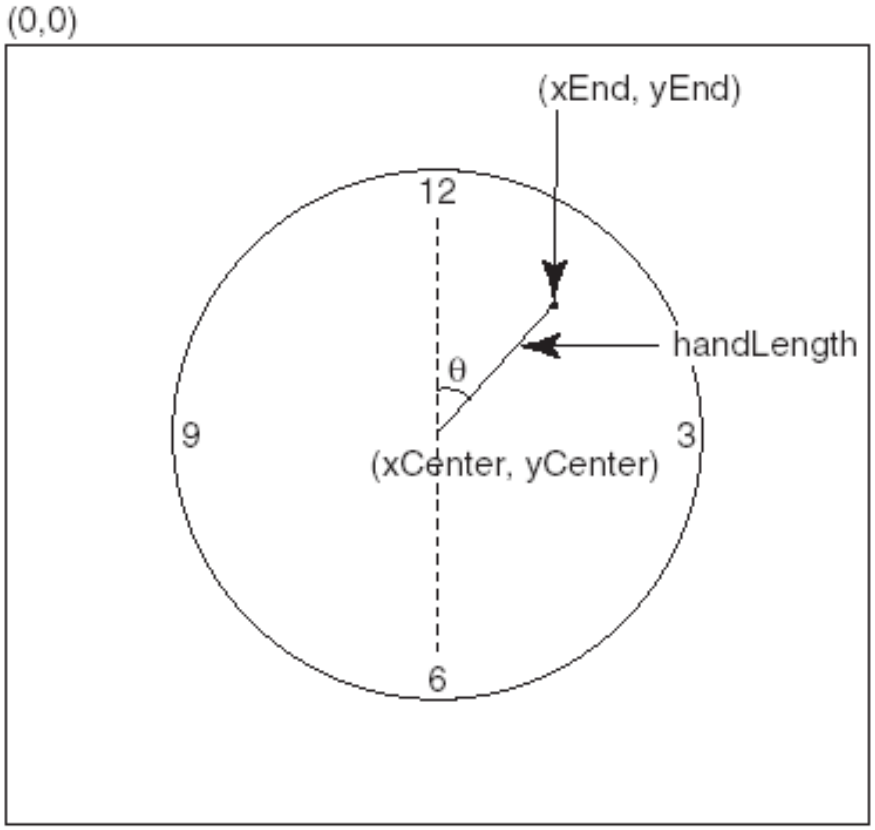
# Drawing Clock, cont.

$$xEnd = xCenter + handLength \times \sin(\theta)$$

$$yEnd = yCenter - handLength \times \cos(\theta)$$

The position of the minute hand is determined by the minute and second. The exact minute value comined with seconds is minute + second/60. For example, if the time is 3 minutes and 30 seconds. The total minutes are 3.5. Since there are sixty minutes in one hour, the angle for the minute hand is

$$(minute + second/60) \times (2\pi/60)$$

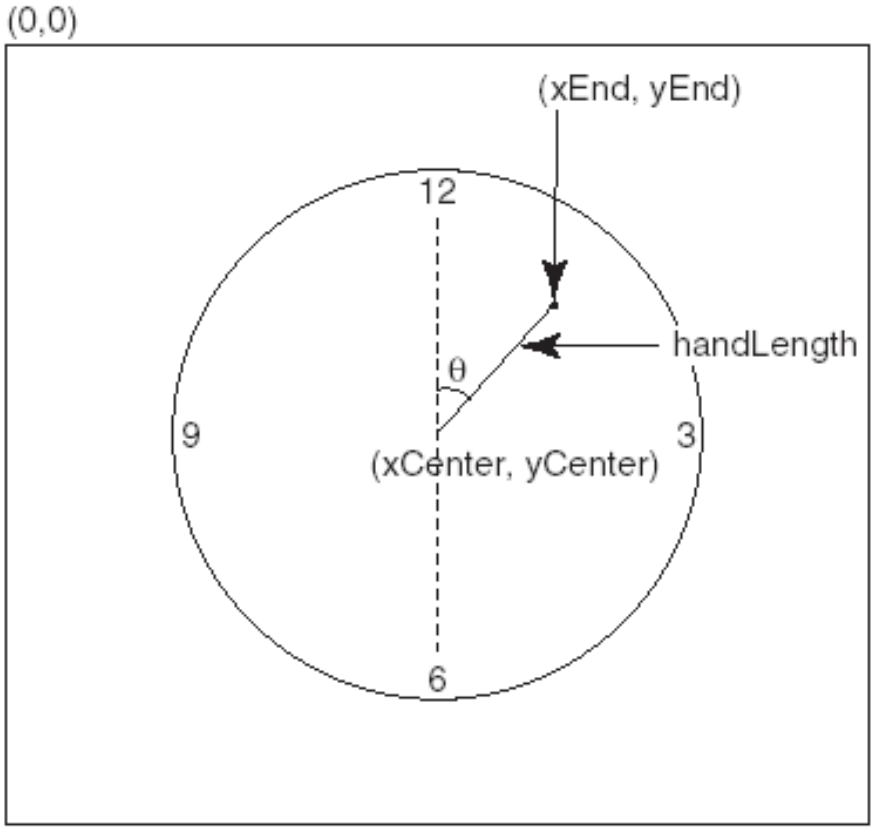


# Drawing Clock, cont.

$$xEnd = xCenter + handLength \times \sin(\theta)$$

$$yEnd = yCenter - handLength \times \cos(\theta)$$

Since one circle is divided into twelve hours, the angle for the hour hand is
$(hour + minute/60 + second/(60 \times 60))) \times (2\pi/12)$



```java
// Fig. 12.20: MouseDetails.java
// Demonstrating mouse clicks and
// distinguishing between mouse buttons.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MouseDetails extends JFrame {
   private String s = "";
   private int xPos, yPos;

   public MouseDetails()
   {
      super( "Mouse clicks and buttons" );

      addMouseListener( new MouseClickHandler() );

      setSize( 350, 150 );
      show();
   }

   public void paint( Graphics g )
   {
      g.drawString( "Clicked @ [" + xPos + ", " + yPos + "]",
                    xPos, yPos );
   }

```

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.exit( 0 )
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 button
55             s += " with center mouse button";
56          else                        // Left mouse button
57             s += " with left mouse button";
58
```

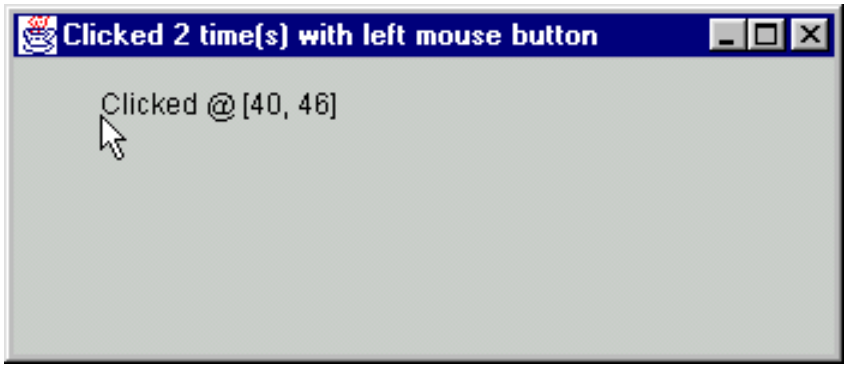
Use a named inner class as the event handler. Can still inherit from MouseAdapter (**`extends MouseAdapter`**).

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

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

Set the Frame's title bar.

# Program Output



# JButton

- Methods of class **Jbutton**

```
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**

```java
// Fig. 12.12: CheckBoxTest.java
// Creating Checkbox buttons.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class CheckBoxTest extends JFrame {
   private JTextField t;
   private JCheckBox bold, italic;

   public CheckBoxTest()
   {
      super( "JCheckBox Test" );

      Container c = getContentPane();
      c.setLayout(new FlowLayout());

      t = new JTextField( "Watch the font style change", 20 );
      t.setFont( new Font( "TimesRoman", Font.PLAIN, 14 ) );
      c.add( t );

      // create checkbox objects
      bold = new JCheckBox( "Bold" );
      c.add( bold );

      italic = new JCheckBox( "Italic" );
      c.add( italic );

      CheckBoxHandler handler = new CheckBoxHandler();
      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( String args[] )
47     {
48        new CheckBoxTest();
49     }
50
51     private class CheckBoxHandler implements ItemListener {
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

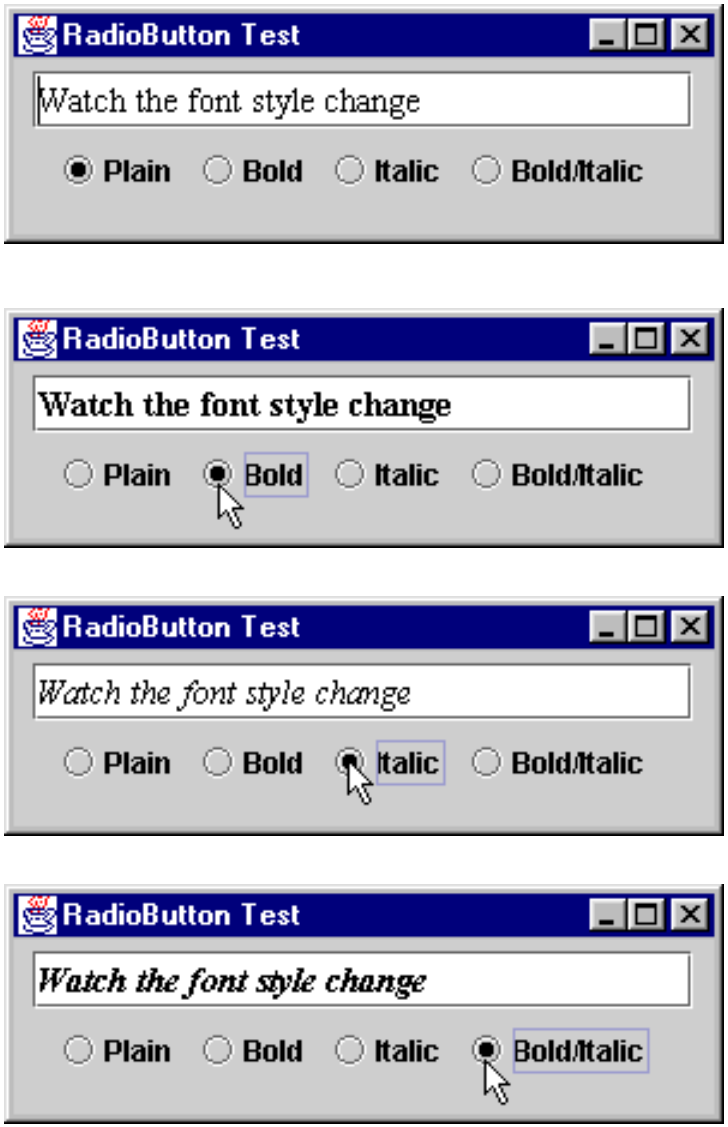
```java
// Fig. 12.12: RadioButtonTest.java
// Creating radio buttons using ButtonGroup and JRadioButton.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class RadioButtonTest extends JFrame {
   private JTextField t;
   private Font plainFont, boldFont,
                 italicFont, boldItalicFont;
   private JRadioButton plain, bold, italic, boldItalic;
   private ButtonGroup radioGroup;

   public RadioButtonTest()
   {
      super( "RadioButton Test" );

      Container c = getContentPane();
      c.setLayout( new FlowLayout() );

      t = new JTextField( "Watch the font style change"
      c.add( t );

      // Create radio buttons
      plain = new JRadioButton( "Plain", true );
      c.add( plain );
      bold = new JRadioButton( "Bold", false);
      c.add( bold );
      italic = new JRadioButton( "Italic", false );
      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 RadioButtonHandler();
36        plain.addItemListener( handler );
37        bold.addItemListener( handler );
38        italic.addItemListener( handler );
39        boldItalic.addItemListener( handler );
40
41        // create logical relationship between JRadioButtons
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, 14 );
51        boldItalicFont =
52           new Font( "TimesRoman", Font.BOLD + Font.ITALIC, 14 );
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 with

- 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

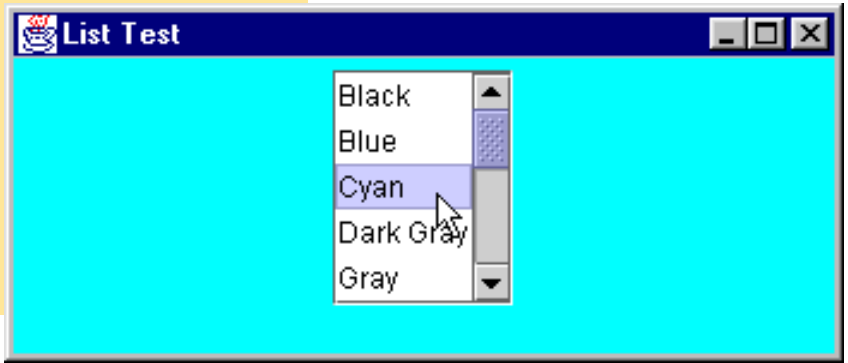
```
30        // create a list with the items in the colorNames array
31        colorList = new JList( colorNames );
32        colorList.setVisibleRowCount( 5 );
33
34        // do not allow multiple selections
35        colorList.setSelectionMode(
36           ListSelectionModel.SINGLE_SELECTION );
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 e )
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**).



# The `Font` Class

```
Font myFont = Font(name, style, size);
```

**Example:**

```
Font myFont = new Font("SansSerif ",
  Font.BOLD, 16);
Font myFont = new Font("Serif",
  Font.BOLD+Font.ITALIC, 12);
```

# Finding All Available Font Names

```
GraphicsEnvironment e =

  GraphicsEnvironment.getLocalGraphicsEnviron
  ment();
String[] fontnames =
  e.getAvailableFontFamilyNames();
for (int i = 0; i < fontnames.length; i++)
  System.out.println(fontnames[i]);
```

# Setting Fonts

```
public void paint(Graphics g) {
  Font myFont = new Font("Times", Font.BOLD,
  16);
  g.setFont(myFont);
  g.drawString("Welcome to Java", 20, 40);

  //set a new font
  g.setFont(new Font("Courier",
  Font.BOLD+Font.ITALIC, 12));
  g.drawString("Welcome to Java", 20, 70);
}
```

# Standard AWT Event Listeners (Summary)

Why no adapter class?

| Listener | Adapter Class (If Any) | Registration Method |
|---|---|---|
| ActionListener | | addActionListener |
| AdjustmentListener | | addAdjustmentListener |
| ComponentListener | ComponentAdapter | addComponentListener |
| ContainerListener | ContainerAdapter | addContainerListener |
| FocusListener | FocusAdapter | addFocusListener |
| ItemListener | | addItemListener |
| KeyListener | KeyAdapter | addKeyListener |
| MouseListener | MouseAdapter | addMouseListener |
| MouseMotionListener | MouseMotionAdapter | addMouseMotionListener |
| TextListener | | addTextListener |
| WindowListener | WindowAdapter | addWindowListener |

# Standard AWT Event Listeners (Details)

- ActionListener
  - Handles buttons and a few other actions
    - actionPerformed(ActionEvent event)
- AdjustmentListener
  - Applies to scrolling
    - adjustmentValueChanged(AdjustmentEvent event)
- ComponentListener
  - Handles moving/resizing/hiding GUI objects
    - componentResized(ComponentEvent event)
    - componentMoved (ComponentEvent event)
    - componentShown(ComponentEvent event)
    - componentHidden(ComponentEvent event)

# Standard AWT Event Listeners (Details Continued)

- ContainerListener
  - Triggered when window adds/removes GUI controls
    - `componentAdded(ContainerEvent event)`
    - `componentRemoved(ContainerEvent event)`
- FocusListener
  - Detects when controls get/lose keyboard focus
    - `focusGained(FocusEvent event)`
    - `focusLost(FocusEvent event)`

# Standard AWT Event Listeners (Details Continued)

- ItemListener
  - Handles selections in lists, checkboxes, etc.
    - `itemStateChanged(ItemEvent event)`
- KeyListener
  - Detects keyboard events
    - `keyPressed(KeyEvent event)` -- any key pressed down
    - `keyReleased(KeyEvent event)` -- any key released
    - `keyTyped(KeyEvent event)` -- key for printable char released

# Standard AWT Event Listeners (Details Continued)

- MouseListener
  - Applies to basic mouse events
    - mouseEntered(MouseEvent event)
    - mouseExited(MouseEvent event)
    - mousePressed(MouseEvent event)
    - mouseReleased(MouseEvent event)
    - mouseClicked(MouseEvent event) -- Release without drag
      - Applies on release if no movement since press
- MouseMotionListener
  - Handles mouse movement
    - mouseMoved(MouseEvent event)
    - mouseDragged(MouseEvent event)