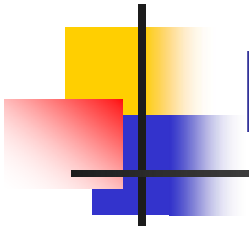
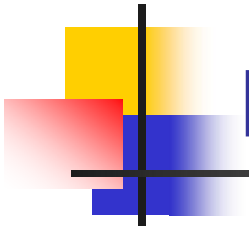


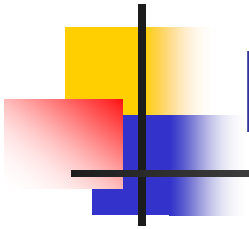
Event Handling

- One of the key concepts in GUI programming is Event Handling.
- Something (Mouse click / Key press, Menu Selection, Timer expiring, Network message received) happens. These would be all *Event Sources*.
- Some GUI components might care about one of these things happening and need to react to it. These components would register themselves with the Event Source, so the source would tell them when something happens. These are the *Event Listeners*.



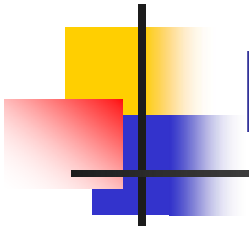
Event Handling in other Languages

- If you've ever used VB, in VB Event handling is easy. To start with, every component is click and drag. Then, each has event code prewritten in it – so a VB Button would have a procedure called VBButton_Click, and then you can put the code you want to run when that button is clicked right in there.
- Doing Event Handling in native C, on the other hand, is extremely hard. You have to manually check the OS's event queue, then decide what to do. Doing this in a clean fashion is impossible.



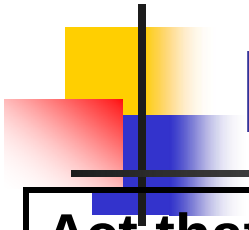
Event Handling in Java

- Java takes a middle-of-the-road approach, but in terms of ease of use and power.
- Java provides ways to hook into different events through different Interfaces.



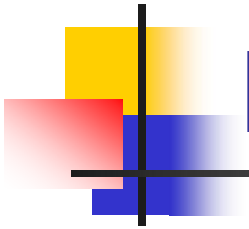
Event Basics

- All Events are objects of Event Classes.
- All Event Classes derive from EventObject.
- When an Event occurs, Java sends a message to all registered Event Listeners from the Event source (this actually was a change, the old AWT event model would send it out to everyone).



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

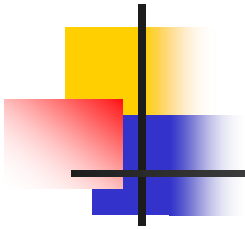


Registering Event Listeners

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

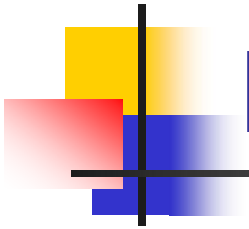
```
eventSourceListener.addEventListener(eventListenerObject);
```

- So, from here we have 2 ways to create the listener object.
  - Create a class that implements that interface (ActionListener, WindowListener, etc.)
  - Use an anonymous Inner Class and attach it



Registering Event Listeners

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

class MyListener implements ActionListener
{
      public void actionPerformed(ActionEvent e)
      {
         //code to run
      }
}

ActionListener listener = new MyListener();
JButton button = new JButton("Ok");
button.addActionListener(listener);
```

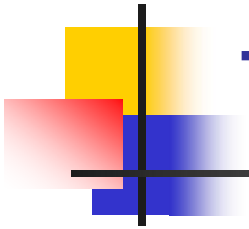


Event Listeners

- When the user clicks the button, the JButton object generates an ActionEvent object.
- It this calls the listener object's actionPerformed() method and passes that method the event object it just generated.
- Note that a single event source can have multiple listeners listening for its events. In this case, the source calls the actionPerformed() method of each of its listeners when the event is generated.

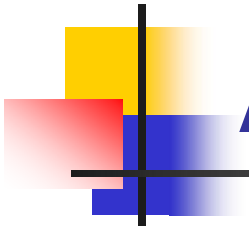
Or… (the way everyone does it.. with a Inner Class)

```
JButton button=new JButton("OK");
button.addActionListener( new ActionListener()   //Anon Inner class
{
    public void actionPerformed(ActionEvent e)
    {
       //handling code here
    }
});
```



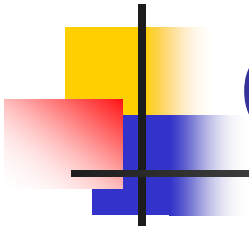
Things to realize

- If we're doing Event handling, we probably need to import javaw.swing.* (for the Swing components), java.awt.* and java.awt.event.* (for the Event handling stuff). Another example of how Swing and AWT comingle.
- the ActionListener interface defines only 1 method, actionPerformed. Others, like WindowListener and the MouseListeners, define more than 1. You have to implement them all to use it. More on this shortly.



Another Example

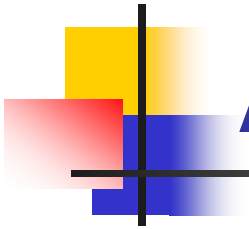
```
public class ButtonTest extends JPanel {
        JButton actionButton;
        JTextField theField;

        ButtonTest()     //constructor
        {
            theField=new JTextField(20);
            actionButton=new JButton("Set Text Field");
            this.add(theField);     // add components to ourself (we're a panel)
            this.add(actionButton);
            actionButton.addActionListener(new ActionListener()
            {
              public void actionPerformed(ActionEvent e)
              {
                theField.setText("JButton was pressed");
              }
            });
        }}
```



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, for instance, WindowListener, which looks like

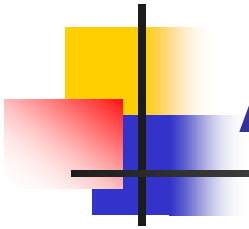
```
public interface WindowListener {
   void windowOpened(WindowEvent e);
   void windowClosing(WindowEvent e);
   void windowClosed(WindowEvent e);
   void windowIconified(WindowEvent e);
   void windowDeIconified(WindowEvent e);
   void windowActivated(WindowEvent e);
   void windowDeActivated(WindowEvent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



Adapter Classes

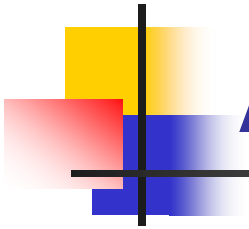
- So?
- Well, since these are interfaces, it means you have to defined every method listed. So, to add an windowListener to a JFrame, you'd have to

```
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
});
```



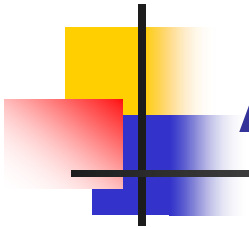
Adapter Classes

- This is the kind of mindless garbage that drives Developers nuts. If you're the boss, this is not what you want your $50+/hr. Software Engineers doing.
- 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

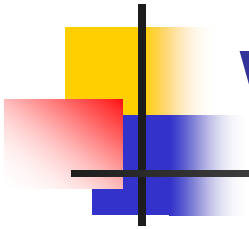
```
public class winAdapter extends WindowAdapter
{
    //WindowAdapter overloads all 7 methods in the WindowListner interface
   //Now we overload those we want to do something
    public void windowClosing(WindowEvent e)
    {
        System.exit(0);
    }
}

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



Adapter Classes

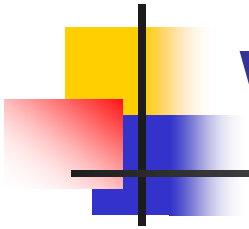
- We can even bring this a level further. Just use an anonymous inner class to use the Adapter class.

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



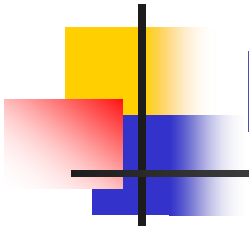
Which way is better?

- Which approach is better? Using anonymous inner classes, or creating new classes which handle the listener events?
- From a performance standard, it's a tie. Inner classes are just transformed into regular classes by the compiler, delimited with a $ to denote the inner and outer class names.



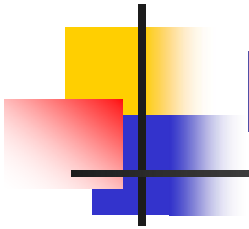
Which way is better?

- However, from an OOP standpoint, the inner class method gets the nod. Each event source gets its own listener, which can directly modify whatever created it, because it’s an inner class.
- It also makes it more readable, since the code is right there where the event gets attached.



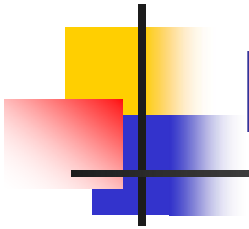
Different types of Event

- Java makes a useful distinction between two types of events. low-level events and semantic events.
  - A semantic event is something like a user clicking a button. ActionEvents are semantic events.
  - A low-level event is something like a key being pressed, or a mouse moving.
- Semantic Events are built upon low level events and are handled for you



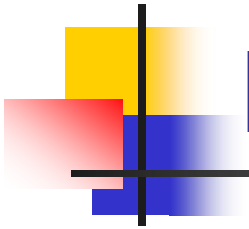
Different types of Events

- So, semantic events are ActionEvents, AdjustmentEvents (scroll bar moves), ItemEvents (something selected from a CheckBox) and TextEvents (text is a field was changed).
- Low level events are KeyEvents, MouseEvents, MouseWheelEvents, FocusEvents, WindowEvents, ContainerEvents (something added or removed from a container), and ComponentEvents (something being resized)



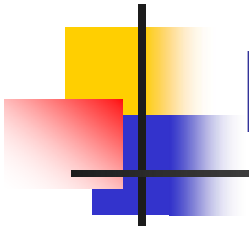
Focus Events

- A *FocusEvent* occur when a component gains or loses focus (when you like in a JTextArea, it gains the focus).
- FocusListener has two methods, focusGained() and focusLost(). The code within focusGained will be run when the attached component gains focus, and focusLost() will be run with that component loses focus.



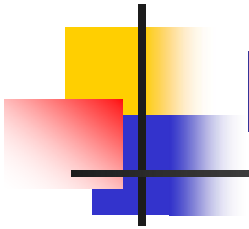
Key Events

- A *KeyEvent* is generated when a key is pressed or released.
- KeyListener must implement three methods, keyPressed(), keyReleased(), and keyTyped().
- keyPressed() will run whenever a key is pressed, keyReleased() will run whenever that key is released, and keyTyped() combines the two – its runs when the key is pressed and then released, and signifies a keystroke.



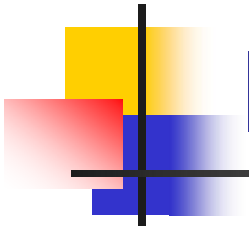
Key Events

- But who do you register the KeyEvent to? Isn't the keyboard kind of a system-wide thing?
- Any component can be a KeyListener. If you make JTextField the KeyListener, it will generate the appropriate events to dispatch.



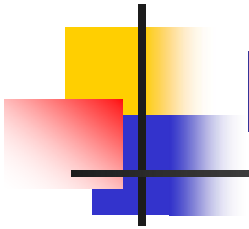
Mouse Events

- Mouse Events are generated whenever a mouse moves. If you're only concerned with Semantic events (Buttons clicked, scrollbars moved, etc.) then you do not need to worry about MouseEvents.
- MouseEvents are generated like KeyEvents – mousePressed(), mouseReleased(), and mouseClicked(). You can ignore the first two if you only care about clicking.



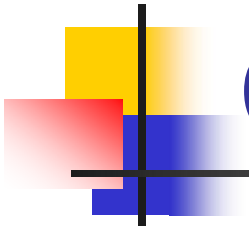
Mouse Events

- You can call the getClickCount() method on a MouseEvent object to distinguish between a single and a double click.
- You can distinguish between the various mouse buttons by calling the getModifiers() method on a MouseEvent object.
- MouseEvents are also generated when the mouse pointer enters and leaves components (mouseEntered() and mouseExited()).



Mouse Events

- All of these methods are part of the *MouseListener* interface.
- The actual movement of the mouse is handled with the *MouseMotionListener* interface.
- This is done because most applications only care about where you click and not how and where you move the mouse pointer around.



Custom Events

- It is possible to create custom events in Java.
- Basically, you create an Event class that extends AWTEvent, and insert into the event queue using the Toolkit.getDefaultToolKit()
  .getSystemEventQueue()
  .postEvent(event);
- However, this should only be used where there is no other possible way. This also has security implications, and it isn't allowed in Applets.