

# CS2110. GUIs: Listening to Events

Also anonymous classes versus Java 8 functions

Lunch with instructors:

Visit Piazza pinned post to reserve a place

Download demo zip file from course website, look at demos of GUI things: sliders, scroll bars, listening to events, etc. We'll update it after today's lecture.

A4 submissions: this Thursday

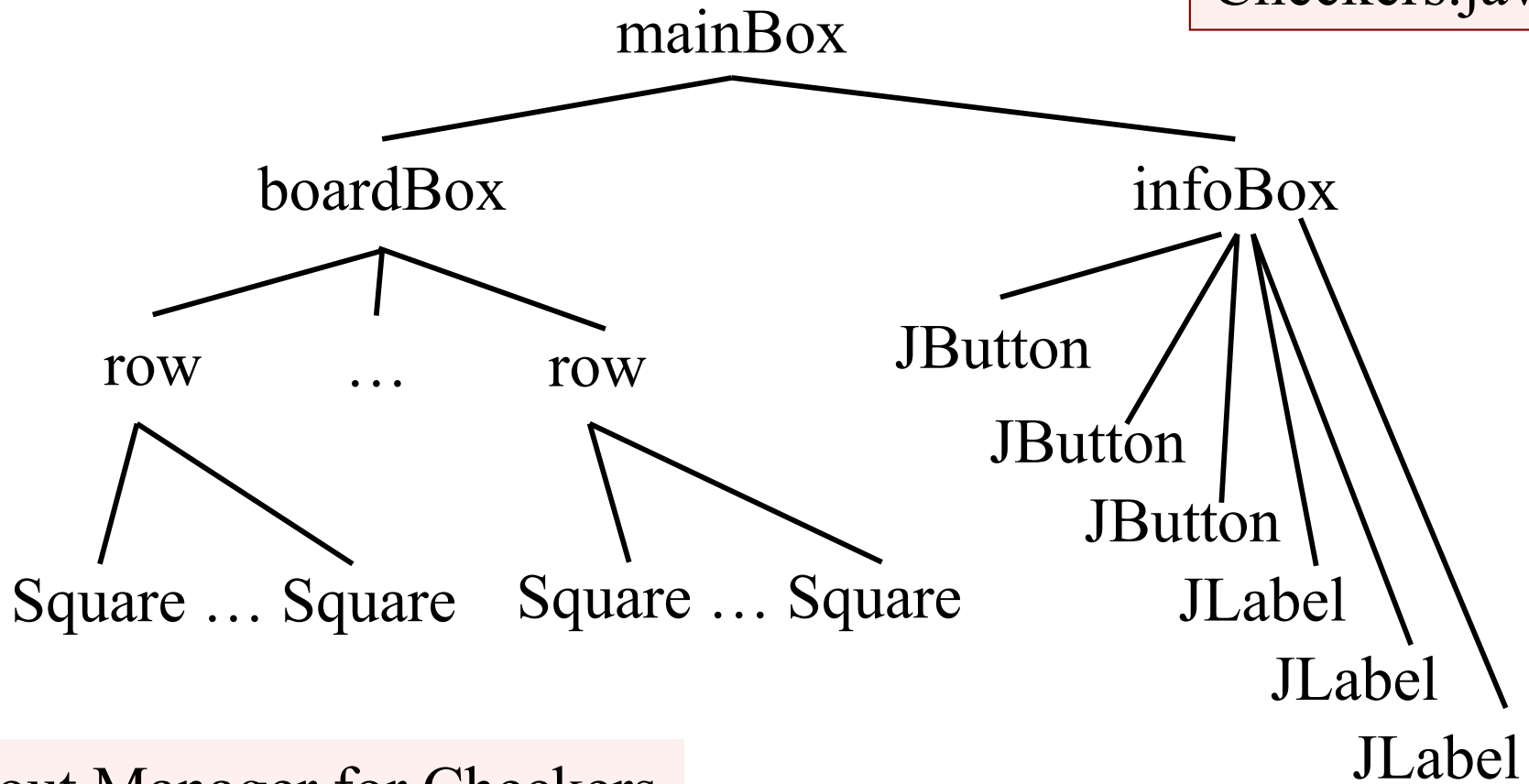
A4 deadline for late submissions: Sunday

21 March: last day to drop or change grade to S/U

Consider taking course S/U (if allowed) to relieve stress.

Need a letter grade of C- or better to get an S.

Right now: 13 AUDIT, 45 S/U |



Layout Manager for Checkers game has to process a tree

pack(): Traverse the tree, determining the space required for each component

boardBox: vertical Box  
row: horizontal Box  
Square: Canvas or JPanel  
infoBox: vertical Box

## Listening to events: mouse click, mouse movement into or out of a window, a keystroke, etc.

- An **event** is a mouse click, a mouse movement into or out of a window, a keystroke, etc.
- To be able to “listen to” a kind of event, you have to:
  1. Have some class C implement an interface IN that is connected with the event.
  2. In class C, override methods required by interface IN; these methods are generally called when the event happens.
  3. Register an object of class C as a *listener* for the event. That object’s methods will be called when event happens.

We show you how to do this for clicks on buttons, clicks on components, and keystrokes.

# Anonymous functions

You know about interface Comparable.

```
public interface Comparable<T> {  
    /** Return neg, 0 or pos ...*/  
    int compareTo(T ob);  
}
```

```
public abstract class Shape implements Comparable {
```

```
    ...
```

```
    /** Return the area of this shape */  
    public abstract double area() ;
```

```
    /** Return neg, 0, or pos ... */  
    public int compareTo(Shape ob) {
```

```
        ...
```

```
    }  
}
```

```
In some class:  
Shape[] s= ...;  
...  
Arrays.sort(s);
```

```
Use an anonymous function  
to make this easier!
```

# Anonymous functions

You know about interface Comparable.

```
public interface Comparable<T> {  
    /** Return neg, 0 or pos ...*/  
    int compareTo(T ob);  
}
```

```
public abstract class Shape implements Comparable {
```

```
    ...
```

```
    /** Return the area of this shape */  
    public abstract double area() ;
```

```
    /** Return neg, 0, or pos ... */  
    public int compareTo(Shape ob) {
```

```
        ...
```

```
    }  
}
```

```
In some class:  
Shape[] s= ...;  
...  
Arrays.sort(s);
```

Use an anonymous function to make this easier!

# Anonymous functions

Here is a function:

```
public int f(Person b, Person c) {  
    return b.age – c.age;  
}
```

```
public class Person {  
    public String name;  
    public int age;  
    ...  
}
```

Written as an anonymous function

`(Person b, Person c) -> b.age – c.age`

Anonymous because it does not have a name.

Don't need keyword **return**. Can put braces around the body if it is more than a single expression.

Depending on where it is written, don't need to put in types of b, c if the types can be inferred.

# Anonymous functions

In some class:

```
Person p[] = new Person[10];  
... code to put in 10 Persons ...
```

```
/** Sort p on age
```

```
Arrays.sort(p, (Person b, Person c) -> b.age - c.age);
```

```
/** Sort p in descending order of age
```

```
Arrays.sort(p, (b, c) -> c.age - b.age);
```

```
public class Person {  
    public String name;  
    public int age;  
    ...  
}
```

When Java compiles these calls, it will eliminate the anonymous functions and turn it into code that uses interface Comparable! This is “syntactic sugar”!

We use anonymous functions to listen to button clicks.

## What is a JButton?

Instance: associated with a “button” on the GUI,  
which can be clicked to do something

```
jb1= new JButton()           // jb1 has no text on it
jb2= new JButton(“first”)    // jb2 has label “first” on it
jb2.isEnabled()             // true iff a click on button can be
                             // detected
jb2.setEnabled(b);          // Set enabled property
jb2.addActionListener(object); // object must have a method,
                             // which is called when button jb2 clicked (next page)
```

At least 100 more methods; these are most important

JButton is in package javax.swing



# Listening to a JButton

I. Implement interface ActionListener:

```
public class C extends JFrame  
    implements ActionListener { ... }
```

So, C must implement actionPerformed, and it will be called when the button is clicked

```
public interface ActionListener extends .. {  
    /** Called when an action occurs. */  
    public abstract void actionPerformed(ActionEvent e);  
}
```

# Listening to a JButton

1. Implement interface ActionListener:

```
public class C extends JFrame  
    implements ActionListener { ... }
```

2. In C override actionPerformed --called when button is clicked:

```
/** Process click of button */  
public void actionPerformed(ActionEvent e) { ... }
```

```
public interface ActionListener extends EventListener {  
    /** Called when an action occurs. */  
    public abstract void actionPerformed(ActionEvent e);  
}
```

# Listening to a JButton

1. Implement interface ActionListener:

```
public class C extends JFrame  
    implements ActionListener { ... }
```

2. In C override actionPerformed --called when button is clicked:

```
/** Process click of button */  
public void actionPerformed(ActionEvent e) { ... }
```

3. Add an instance of class C an “action listener” for button:

```
button.addActionListener(this);
```

**But instead, we use an anonymous function!**

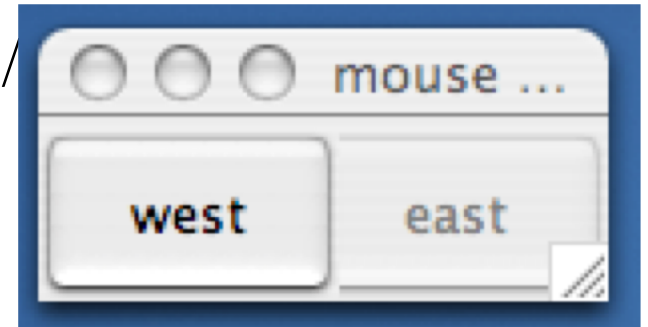
```
Method JButton.addActionListener  
public void addActionListener(ActionListener l)
```

```
/** USE anonymous function */
```

```
class ButtonDemo1 extends JFrame {  
    /** exactly one of eastB, westB is enabled */  
    JButton westB= new JButton("west");  
    JButton eastB= new JButton("east");  
  
    public ButtonDemo1(String t) {  
        super(t);  
        add(westB, BorderLayout.WEST);  
        add(eastB, BorderLayout.EAST);  
  
        westB.setEnabled(false);  
        eastB.setEnabled(true);  
  
        eastB.addActionListener(  
            e -> {boolean b= eastB.isEnabled();  
                eastB.setEnabled(!b);  
                westB.setEnabled(b);} );  
    }  
};
```

**red: listening**

**blue: placing**



Add listener to  
westB the same way

ButtonDemo1

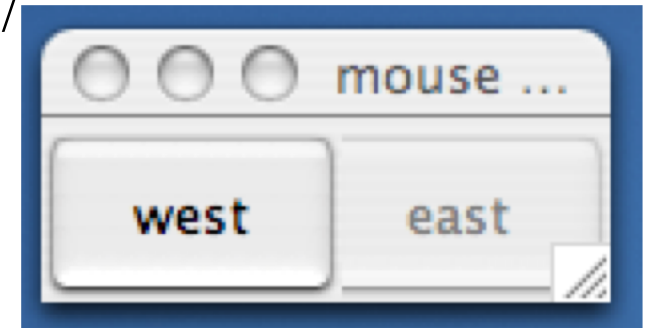
## Listening to a Button

```
/** Save anonymous function in local var*/
```

```
class ButtonDemo1 extends JFrame {  
    /** exactly one of eastB, westB is enabled */  
    JButton westB= new JButton("west");  
    JButton eastB= new JButton("east");  
  
    public ButtonDemo1(String t) {  
        super(t);  
        add(westB, BorderLayout.WEST);  
        add(eastB, BorderLayout.EAST);  
        westB.setEnabled(false);  
        eastB.setEnabled(true);  
  
        ActionListener al=  
            e -> {boolean b= eastB.isEnabled();  
                eastB.setEnabled(!b);  
                westB.setEnabled(b);};  
  
        eastB.addActionListener(al);  
        westB.addActionListener(al);  
        pack(); setVisible(true);  
    }  
}
```

**red: listening**

**blue: placing**



```
eastB.addActionListener(al);  
westB.addActionListener(al);  
pack(); setVisible(true);
```

ButtonDemo1

**Listening to a Button**

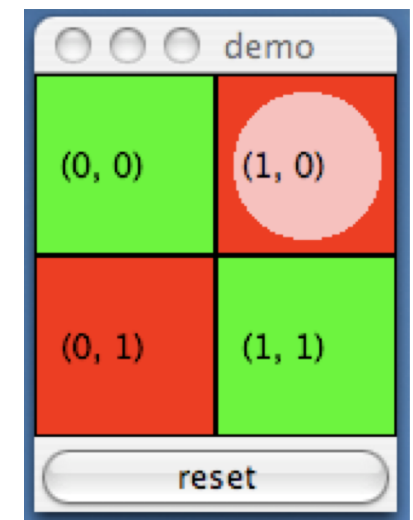
## A JPanel that is painted

MouseDemo2

- The JFrame has a JPanel in its CENTER and a “reset” button in its SOUTH.
- The JPanel has a horizontal box b, which contains two vertical Boxes.
- Each vertical Box contains two instances of class Square.
- Click a Square that has no pink circle, and a pink circle is drawn. Click a square that has a pink circle, and the pink circle disappears.
- Click the reset button and all pink circles disappear.

- This GUI has to listen to:  
(1) a click on Button reset

These are different kinds of events, and they need different listener methods



```
/** Instance: JPanel of size (WIDTH, HEIGHT).
```

```
Green or red: */
```

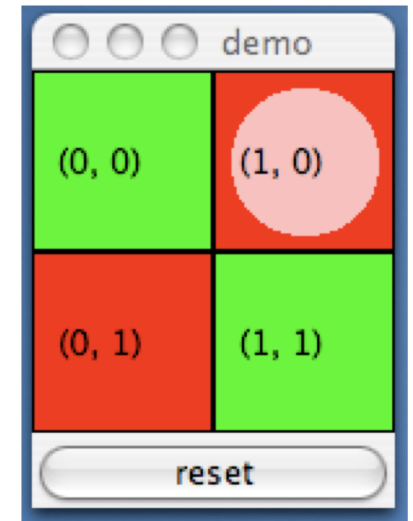
```
public class Square extends JPanel {  
    public static final int HEIGHT= 70;  
    public static final int WIDTH= 70;  
    private int x, y; // Panel is at (x, y)  
    private boolean hasDisk= false;
```

```
/** Const: square at (x, y). Red/green? Parity of x+y. */
```

```
public Square(int x, int y) {  
    this.x= x;    this.y= y;  
    setPreferredSize(new Dimension(WIDTH, HEIGHT));  
}
```

```
/** Complement the "has pink disk" property */
```

```
public void complementDisk() {  
    hasDisk= ! hasDisk;  
    repaint(); // Ask the system to repaint the square  
}
```



**Class  
Square**

continued on later

JPanel, JFrame, and Canvas have a method

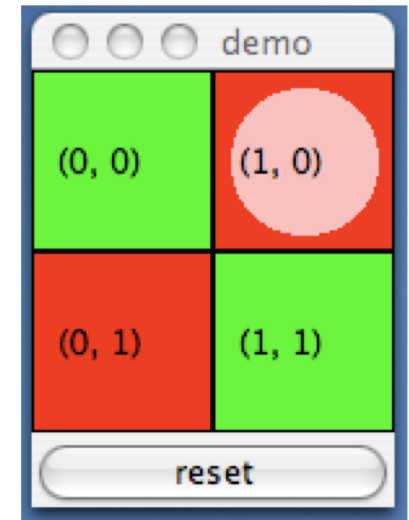
```
paint(Graphics g) //does nothing
```

Override it to “paint” on the component (below).

When the operating system needs the component to be repainted, it calls method paint.

When your code needs to have it repainted, call inherited method repaint().

```
/** Complement the "has pink disk" property */  
public void complementDisk() {  
    hasDisk= ! hasDisk;  
    repaint(); // Ask the system to repaint the square  
}
```



**Class  
Square**



# Class Graphics

An object of abstract class **Graphics** has methods to draw on a component (e.g. on a JPanel, or canvas).

Major methods:

|   |   |
|---|---|
| <code>drawString("abc", 20, 30);</code>     | <code>drawLine(x1, y1, x2, y2);</code>      |
| <code>drawRect(x, y, width, height);</code> | <code>fillRect(x, y, width, height);</code> |
| <code>drawOval(x, y, width, height);</code> | <code>fillOval(x, y, width, height);</code> |
| <code>setColor(Color.red);</code>           | <code>getColor()</code>                     |
| <code>getFont()</code>                      | <code>setFont(Font f);</code>               |

*More methods*

You won't create an object of Graphics; you will be given one to use when you want to paint a component

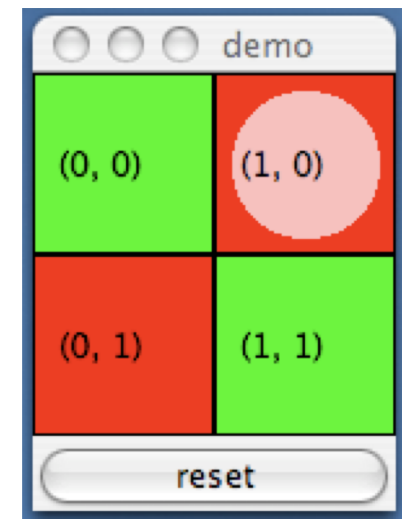
Graphics is in package java.awt

## continuation of class Square

```
/** Paint this square using g. System calls  
    paint whenever square has to be redrawn.*/  
public void paint(Graphics g) {  
    if ((x+y)%2 == 0) g.setColor(Color.green);  
    else g.setColor(Color.red);  
    g.fillRect(0, 0, WIDTH-1, HEIGHT-1);  
    if (hasDisk) {  
        g.setColor(Color.pink);  
        g.fillOval(7, 7, WIDTH-14, HEIGHT-14);  
    }  
    g.setColor(Color.black);  
    g.drawRect(0, 0, WIDTH-1, HEIGHT-1);  
    g.drawString("(" + x + ", " + y + ")", 10, 5 + HEIGHT/2);  
}  
}
```

## Class Square

```
/** Remove pink disk  
    (if present) */  
public void clearDisk() {  
    hasDisk = false;  
    // Ask system to  
    // repaint square  
    repaint();  
}
```



## Listen to mouse event (click, press, release, enter, leave on a component)

```
public interface MouseListener { In package java.awt.event  
    void mouseClicked(MouseEvent e);  
    void mouseEntered(MouseEvent e);  
    void mouseExited(MouseEvent e);  
    void mousePressed(MouseEvent e);  
    void mouseReleased(MouseEvent e);  
}
```

Having write all of these in a class that implements **MouseListener**, even though you don't want to use all of them, can be a pain. So, a class is provided that implements them in a painless way.

## Listen to mouse event (click, press, release, enter, leave on a component)

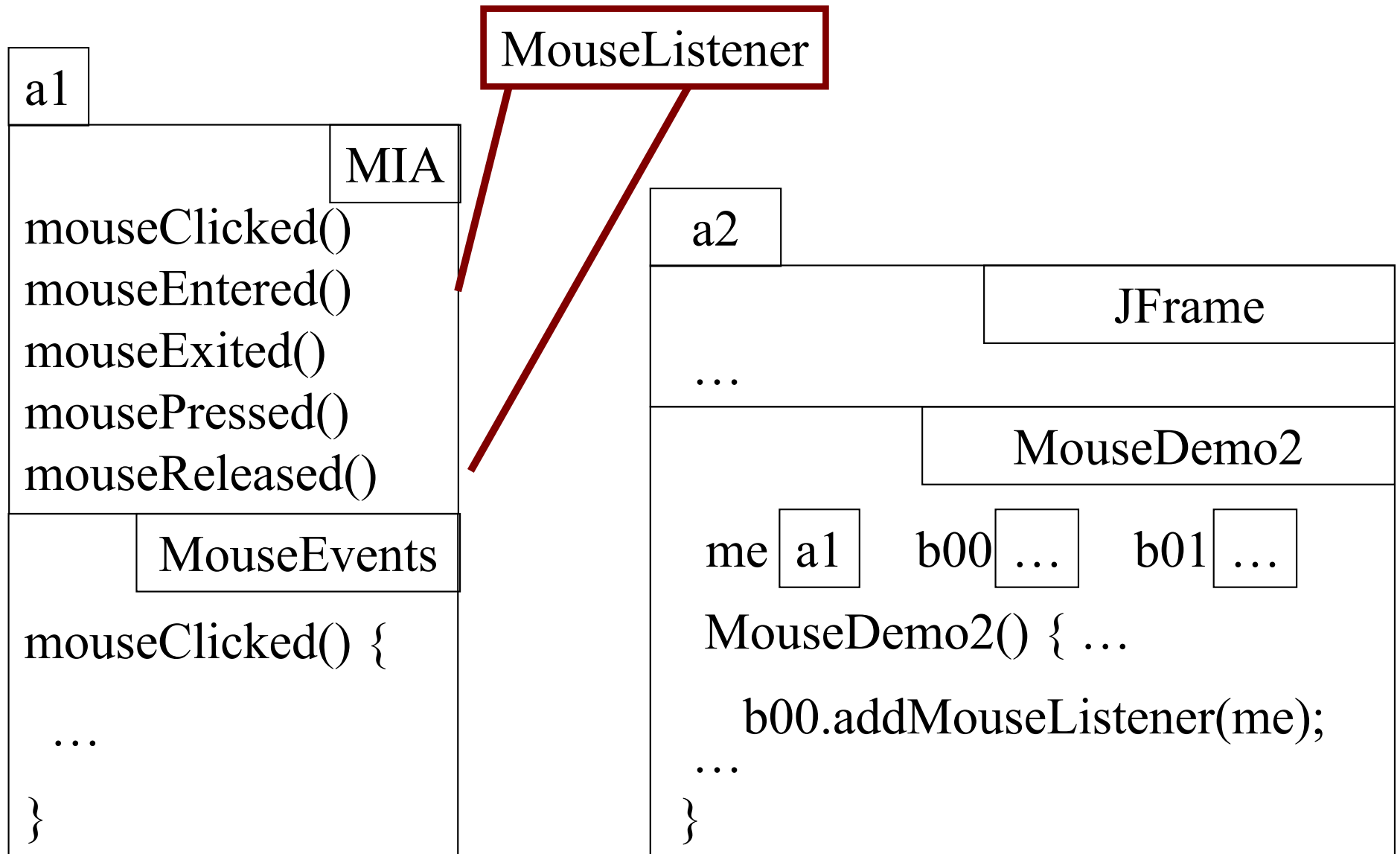
In package java.swing.event

MouseEvents

```
public class MouseInputAdaptor  
    implements MouseListener, MouseInputListener {  
    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) {}  
    ... others ...
```

So, just write a subclass of MouseInputAdaptor and  
} override only the methods appropriate for the application

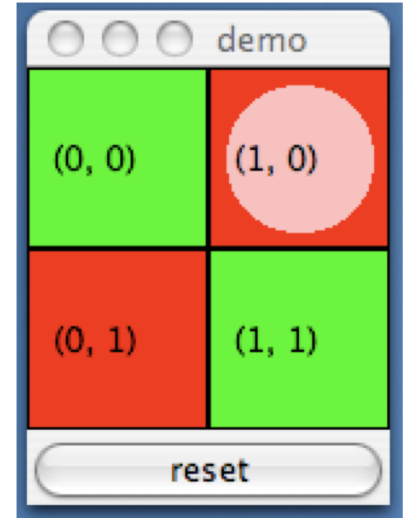
# Javax.swing.event.MouseInputAdapter implements MouseListener



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

## A class that listens to a mouseclick in a Square

**red: listening**  
**blue: placing**



```
/** Contains a method that responds to a
    mouse click in a Square */
```

```
public class MouseEvents
```

```
    extends MouseInputAdapter {
```

```
    // Complement "has pink disk" property
```

```
    public void mouseClicked(MouseEvent e) {
```

```
        Object ob= e.getSource();
```

```
        if (ob instanceof Square) {
```

```
            ((Square)ob).complementDisk();
```

```
        }
```

```
    }
```

```
}
```

This class has several methods (that do nothing) to process mouse events:

mouse click  
mouse press  
mouse release  
mouse enters component  
mouse leaves component  
mouse dragged beginning in component

Our class overrides only the method that processes mouse clicks

```

public class MD2 extends JFrame
    implements ActionListener {
    Box b= new Box(...X_AXIS);
    Box leftC= new Box(...Y_AXIS);
    Square b00, b01= new squares;
    Box riteC= new Box(..Y_AXIS);
    Square b10, b01= new squares;
    JButton jb= new JButton("reset");
    MouseEvents me=
        new MouseEvents();

/** Constructor: ... */
public MouseDemo2() {
    super("MouseDemo2");
    place components in JFrame;
    pack, make unresizeable, visible;

```

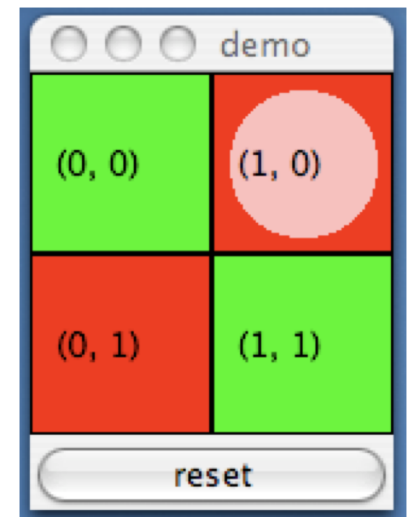
```

        jb.addActionListener(
            e -> clearDisks(e));
        b00.addMouseListener(me);
        b01.addMouseListener(me);
        b10.addMouseListener(me);
        b11.addMouseListener(me);
    }
    public void clearDisks(
        ActionEvent e) {
        call clearDisk() for
        b00, b01, b10, b11
    }

```

red: listening  
blue: placing

MouseDemo2



# Listening to the keyboard

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

```
public class AllCaps extends KeyAdapter {  
    JFrame capsFrame= new JFrame();  
    JLabel capsLabel= new JLabel();  
  
    public AllCaps() {  
        capsLabel.setHorizontalAlignment(SwingConstants.CENTER);  
        capsLabel.setText(":)");  
        capsFrame.setSize(200,200);  
        Container c= capsFrame.getContentPane();  
        c.add(capsLabel);  
        capsFrame.addKeyListener(this);  
        capsFrame.show();  
    }  
  
    public void keyPressed (KeyEvent e) {  
        char typedChar= e.getKeyChar();  
        capsLabel.setText(("" + typedChar + "").toUpperCase());  
    }  
}
```

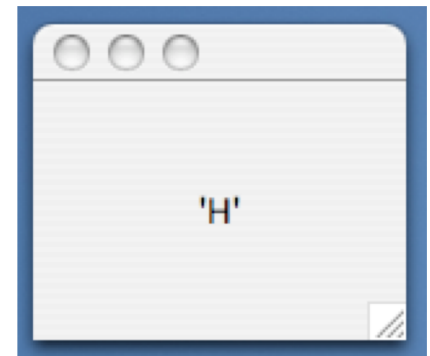
**red: listening**

**blue: placing**

1. Extend this class.

3. Add this instance as a key listener for the frame

2. Override this method. It is called when a key stroke is detected.





```

public class BDemo3 extends JFrame {
    private JButton wB, eB ...;

    public ButtonDemo3() {
        Add buttons to JFrame, ...
        wB.addActionListener(this);
        eB.addActionListener(new BeListener()),
    }

    public void disableE(ActionEvent e) {
        eB.setEnabled(false); wB.setEnabled(true);
    }

    public void disableW(ActionEvent e) {
        eB.setEnabled(true); wB.setEnabled(false);
    }

}

```

Have a different listener for each button

ButtonDemo3

## ANONYMOUS CLASS

You will see anonymous classes in A5 and other GUI programs

Use sparingly, and only when the anonymous class has 1 or 2 methods in it, because the syntax is ugly, complex, hard to understand.

The last two slides of this ppt show you how to eliminate BeListener by introducing an anonymous class.

**You do not have to master this material**

Have a class for which only one object is created?

Use an **anonymous class**.

Use sparingly, and only when the anonymous class has 1 or 2 methods in it, because the syntax is ugly, complex, hard to understand.

```
public class BDemo3 extends JFrame implements ActionListener {
    private JButton wButt, eButt ...;

    public ButtonDemo3() { ...
        eButt.addActionListener(new BeListener());
    }

    public void actionPerformed(ActionEvent e) { ... }

    private class BeListener implements ActionListener {
        public void actionPerformed(ActionEvent e) { body }
    }
}
1 object of BeListener created. Ripe for making anonymous
```

## Making class anonymous will replace **new BeListener()**

Expression that creates object of BeListener

```
eButt.addActionListener( new BeListener () );
```

```
private class BeListener implements ActionListener  
    { declarations in class }  
}
```

1. Write **new**

2. Write **new ActionListener**

3. Write **new ActionListener ()**

4. Write **new ActionListener ()**  
 { declarations in class }

2. Use name of interface that BeListener implements

3. Put in arguments of constructor call

4. Put in class body

5. Replace **new BeListener()** by new-expression

## ANONYMOUS CLASS IN A5.

PaintGUI. setUpMenuBar, fixing item “New”

Save new JMenuItem

```
.. JMenuItem newItem = new JMenuItem("New");  
.. newItem.setMnemonic(KeyEvent.VK_N);  
.. newItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_N,   
.....ActionEvent.CTRL_MASK));  
.. newItem.addActionListener(new ActionListener() {  
.....public void actionPerformed(ActionEvent e) {  
.....newAction(e);  
.....}  
..});
```

Fix it so that **control-N** selects this menu item

**new ActionListener() { ... }** declares an anonymous class and creates an object of it. The class implements **ActionListener**. Purpose: call `newAction(e)` when `actionPerformed` is called

## Using an A5 function (only in Java 8!). PaintGUI.setUpMenuBar, fixing item “New”

### Save new JMenuItem

```
..  
.. JMenuItem newItem = new JMenuItem("New");  
.. newItem.setMnemonic(KeyEvent.VK_N);  
.. newItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_N,  
..... ActionEvent.CTRL_MASK));  
.. newItem.addActionListener(e -> { newAction(e); });  
..
```

Fix it so that  
**control-N**  
selects this  
menu item

argument **e -> { newAction(e); }**

of addActionListener is a function that, when called, calls newAction(e).

# ANONYMOUS CLASS VERSUS FUNCTION CALL

PaintGUI.setUpMenuBar, fixing item “New”

The Java 8 compiler will change this:

```
newItem.addActionListener(e -> { newAction(e); });
```

back into this:

```
newItem.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
        newAction(e);  
    }  
});
```

and actually change that back into an inner class