CS2110. GUIS: Listening to Events

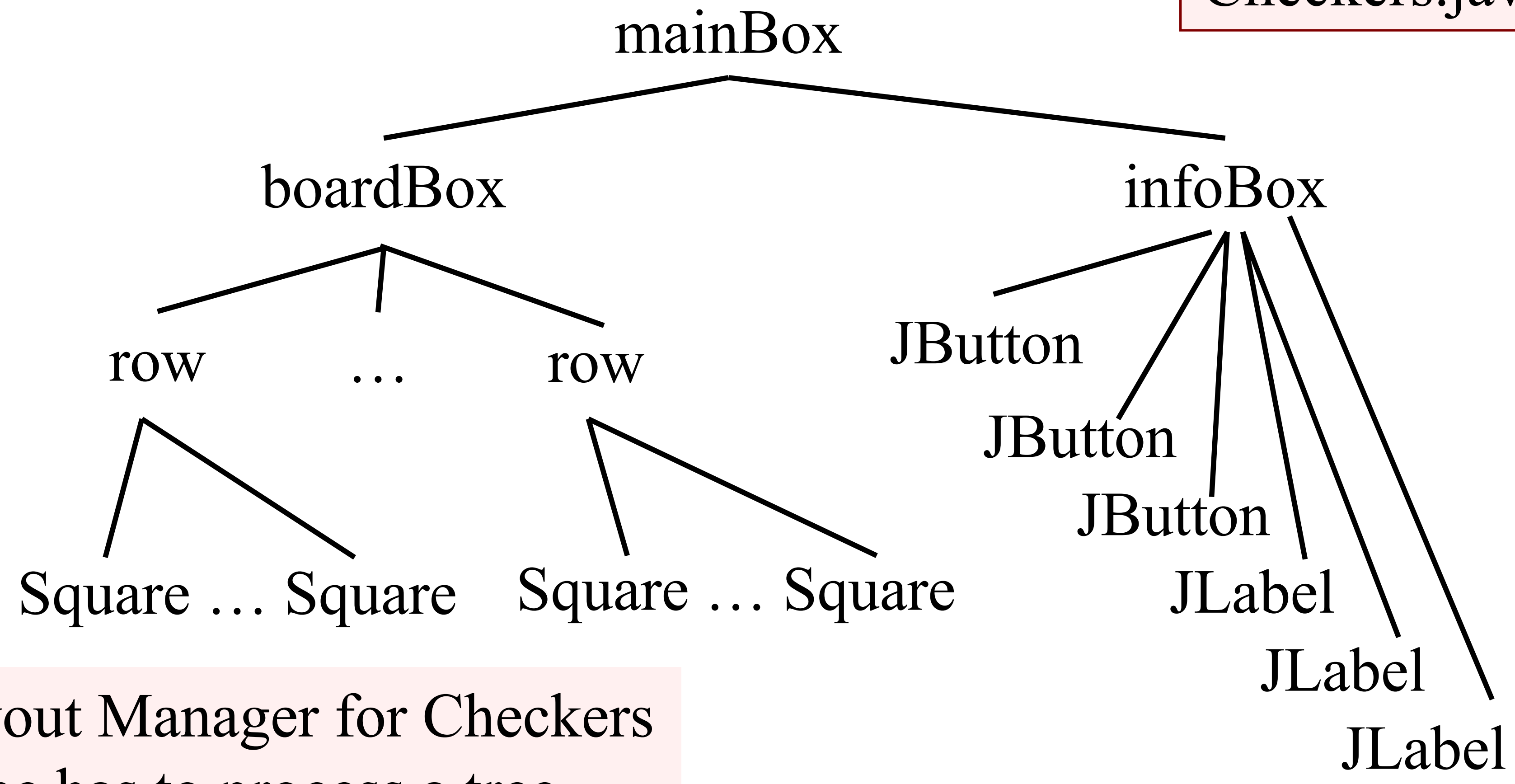
Lunch with instructors: Visit pinned Piazza post.

A4 due tonight.

Consider taking course S/U (if allowed) to relieve stress.
Need a letter grade of C- or better to get an S.

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.

Checkers.java



Layout Manager for Checkers game has to process a tree

pack(): Traverse the tree, determining the space required for each component and its position in the window

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

Have program terminate when stop button clicked

JFrame has a method setDefaultCloseOperation, which can be used to tell system to exit program when the red close button is clicked:

```
setDefaultCloseOperation(EXIT_ON_CLOSE);
```

For other possibilities, look up that method in the JFrame API documentation.

Look at its use in JFrameDemo.

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.

MouseDemo2

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, movements into and out of components, and keystrokes.

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 the most important

JButton is in package javax.swing

Listening to a JButton

1. 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);
```

> Method Jbutton.addActionListener
>
> public void addActionListener(ActionListener l)

ButtonDemo1old

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)

Anonymous functions

You used anonymous functions in A1 to test whether some statement threw an exception.

The second argument to assertThrows is an anonymous function with no parameters. Its body calls g.setAdvisor.

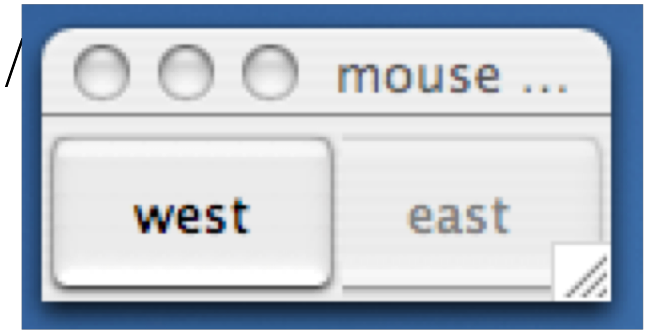
```
assertThrows(AssertionError.class,
                     () -> {g.setAdvisor1(null);});
```

We will be using anonymous functions in listening to events.

red: listening

blue: placing

```
/** 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, BLayout.WEST);
     add(eastB, BLayout, EAST);

     westB.setEnabled(false);
     eastB.setEnabled(true);
    eastB.addActionListener(
       e -> {boolean b= eastB.isEnabled();
             eastB.setEnabled(!b);
             westB.setEnabled(b);}
    );
```



Add listener to westB the same way

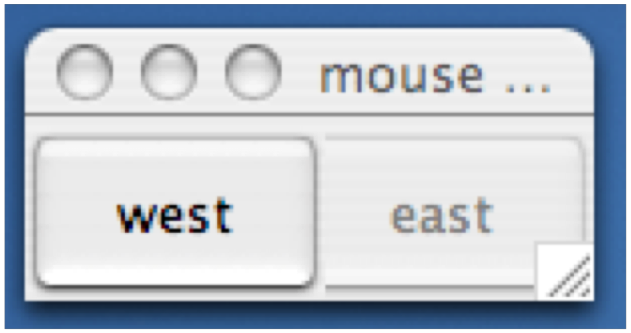
Listening to a Button

ButtonDemo1

red: listening

blue: placing

```
/** 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, BLayout.WEST);
   add(eastB, BLayout, 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);
```

ButtonDemo1

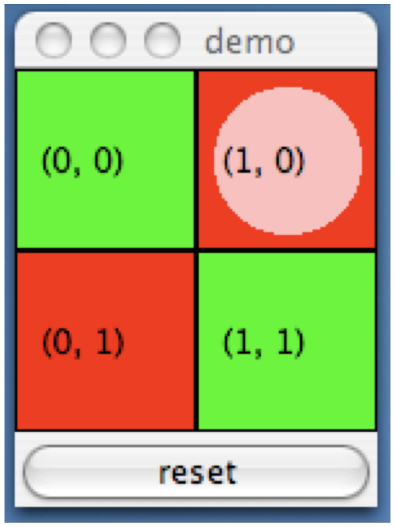
Listening to a Button

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 rest 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

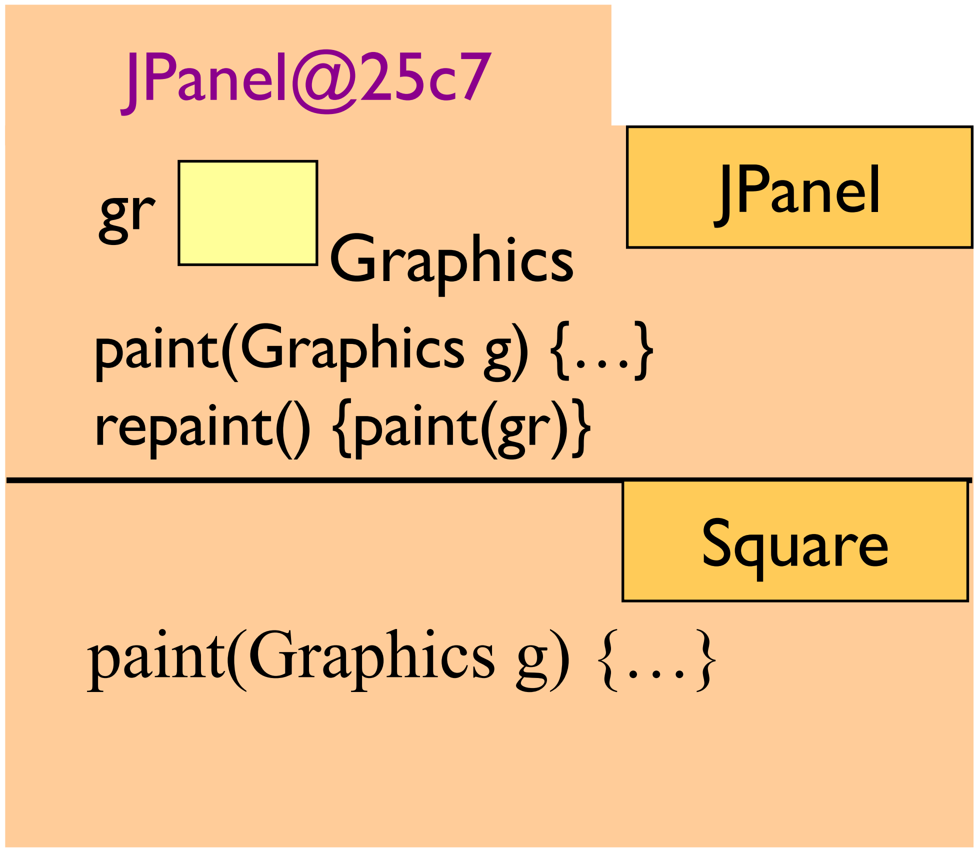


How painting works

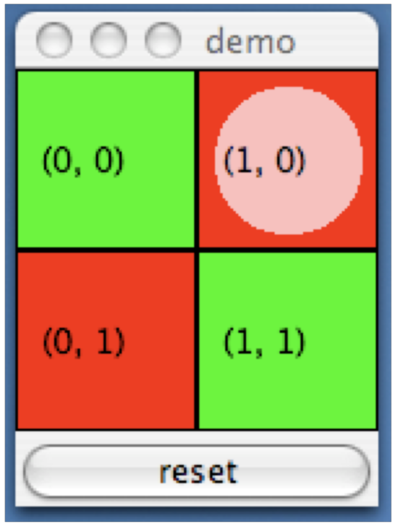
Class Graphics has methods for drawing (painting) on the JPanel. We’ll look at them soon.

Override paint to draw on the JPanel

Whenever you want to call paint to repaint the Jpanel, call repaint()



```
/** 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

Class Graphics

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

Major methods:

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

More methods

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

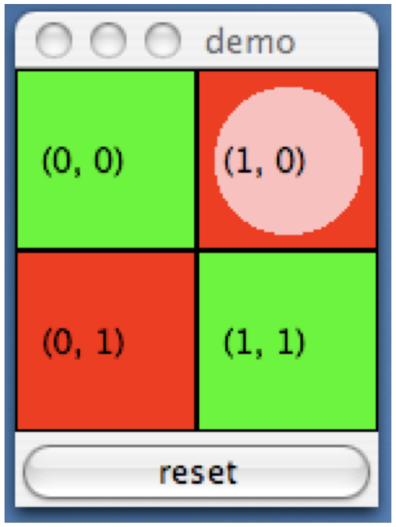
Graphics is in package java.awt

continuation of class Square

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);
 }
}
```

```
/** 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)

In package java.awt.event

```
public interface MouseListener {
    void mouseClicked(MouseEvent e);
    void mouseEntered(MouseEvent e);
    void mouseExited(MouseEvent e);
    void mousePressed(MouseEvent e);
    void mouseReleased(MouseEvent e);
}
```

Having to 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)

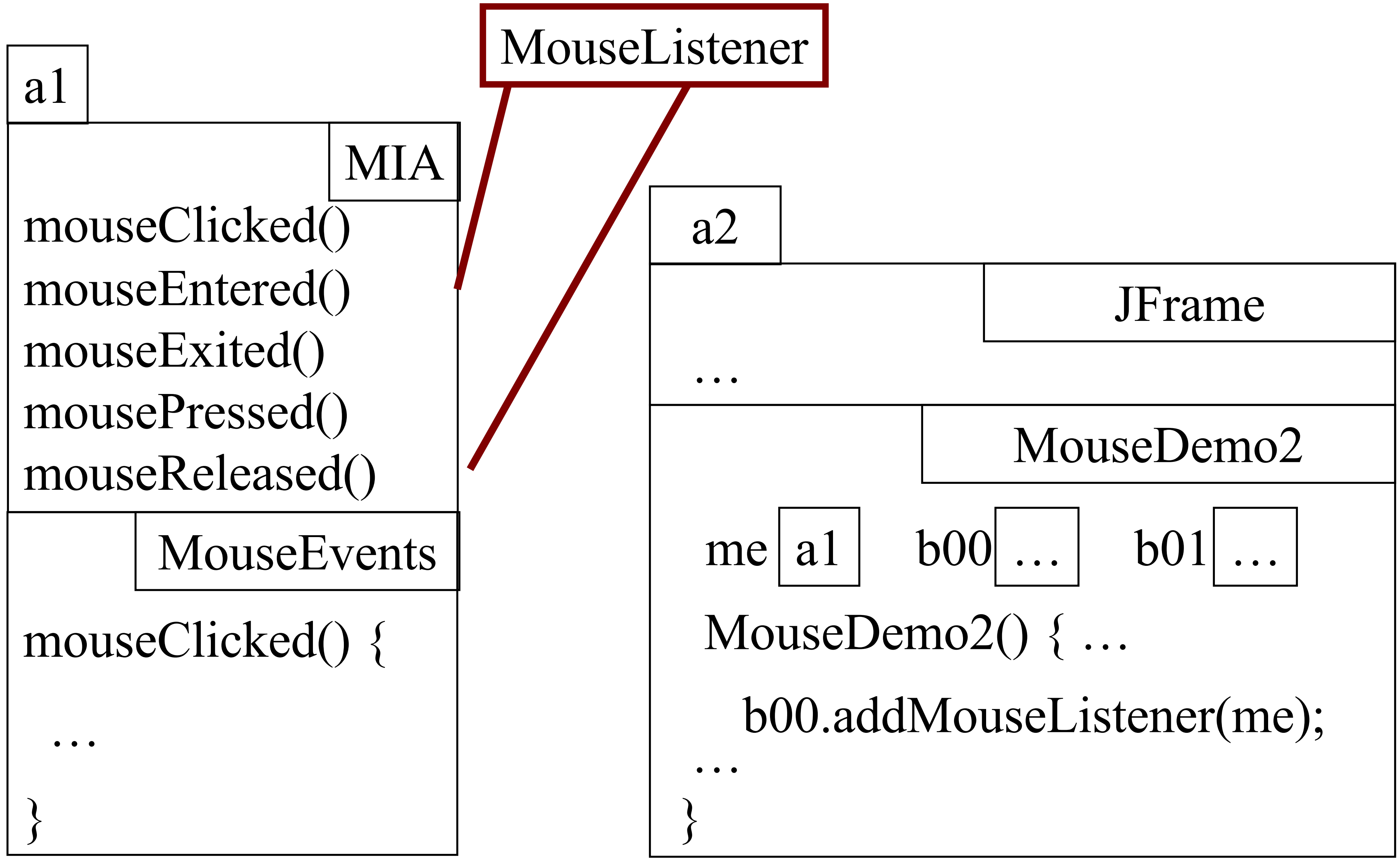
In package java.swing.event

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

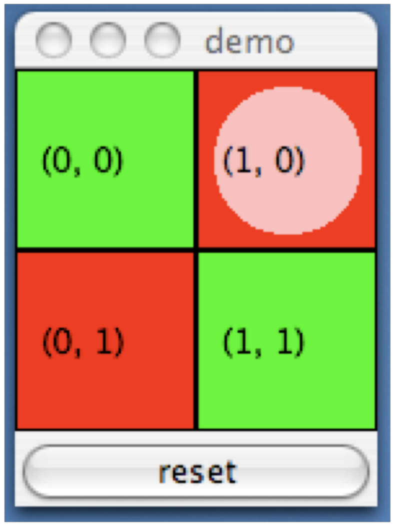


A class that listens to a mouseclick in a Square

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

/** 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();
      }
   }
}
```

red: listening

blue: placing



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

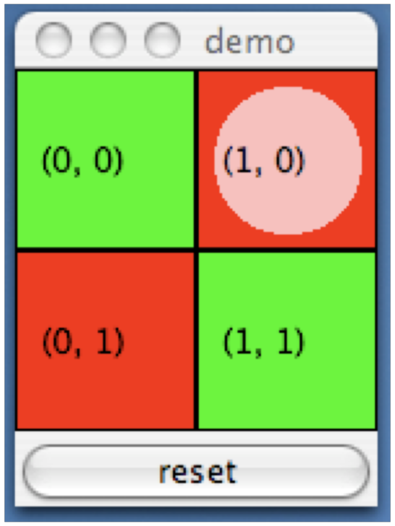
mouse click
mouse press
mouse release
mouse enters component
mouse leaves component
mouse dragged beginning in component

Our class overrides only the method that processes mouse clicks

```
public class MD2 extends JFrame {

   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());
  }
}
```

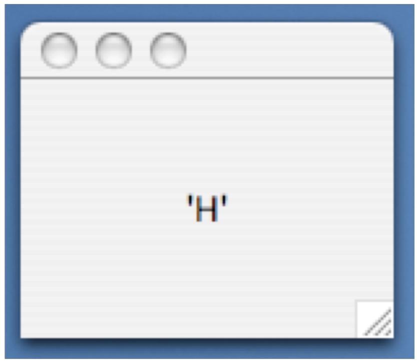
red: listening

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

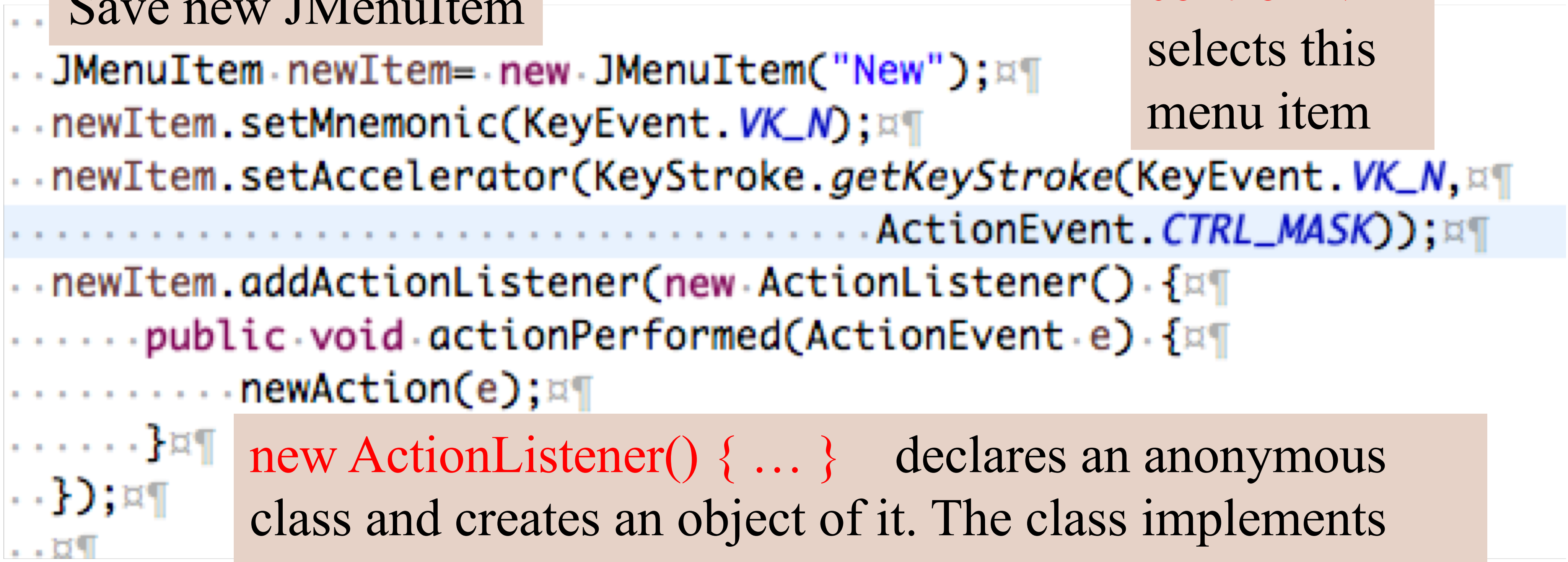
4. Put in class body

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

ANONYMOUS CLASS IN A6.
PaintGUI. setUpMenuBar, fixing item "New"

Save new JMenuItem

Fix it so that control-N selects this menu item



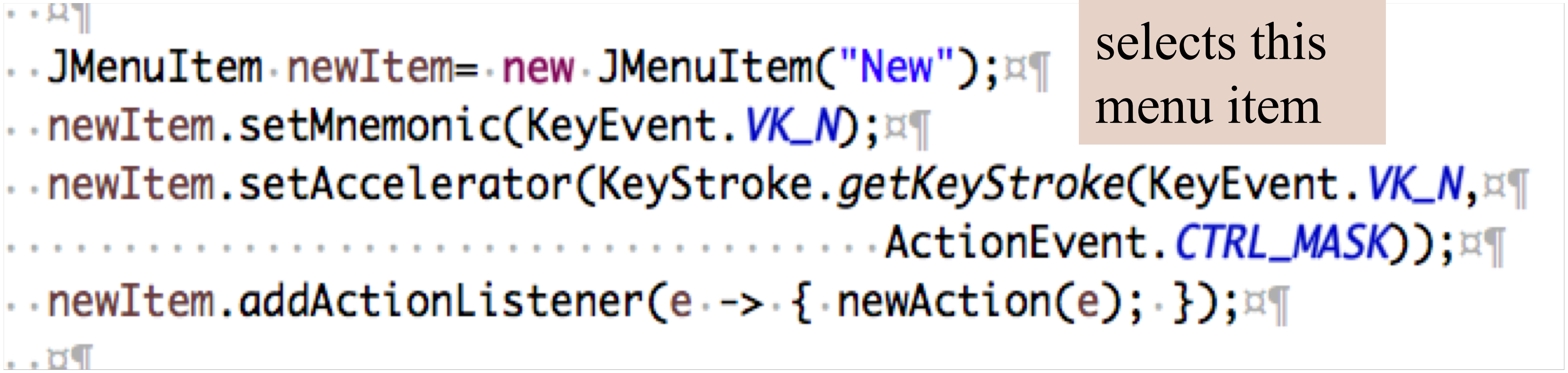
```
  JMenuItem newItem= new JMenuItem("New");
  newItem.setMnemonic(KeyEvent.VK_N);
  newItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_N,
                                  ActionEvent.CTRL_MASK));
  newItem.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent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

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 A6 function (only in Java 8!)
PaintGUI. setUpMenuBar, fixing item “New”

Save new JMenuItem

Fix it so that control-N selects this menu item



```
  
  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); });
  
```

argument e -> { newAction(e);}
of addActionListener is a function that, when called, calls newAction(e).

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 …
```

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

```
/** 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);
```

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.