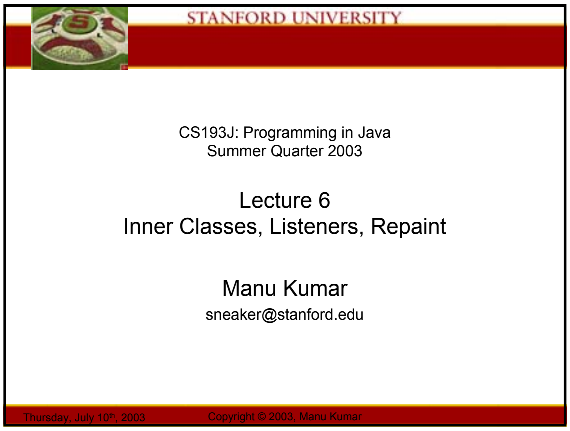
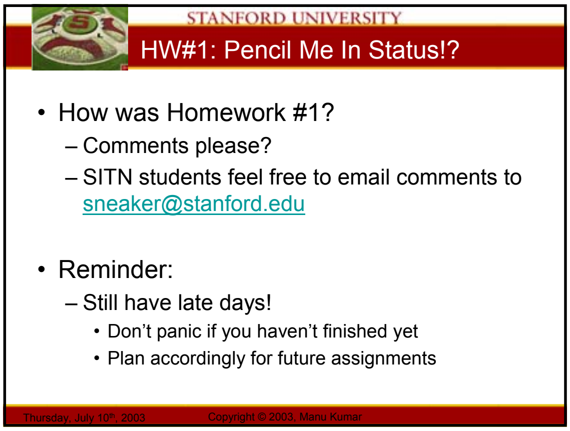


CS193J: Programming in Java
Summer Quarter 2003

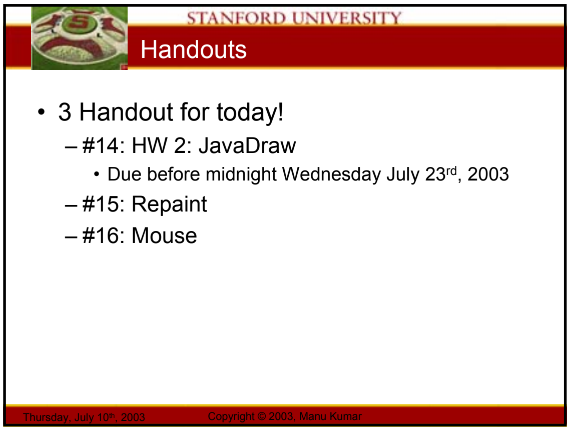
Lecture 6
Inner Classes, Listeners, Repaint

Manu Kumar
sneaker@stanford.edu



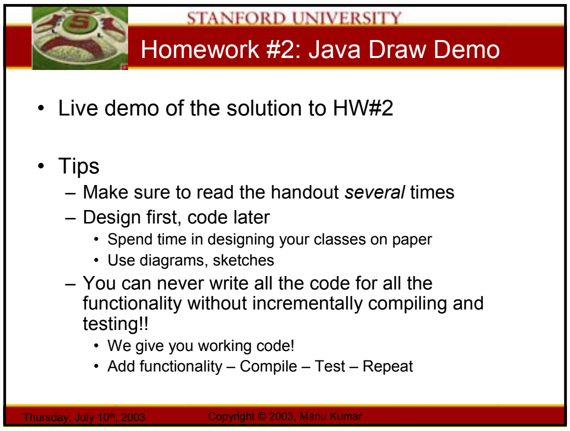
HW#1: Pencil Me In Status!?

- How was Homework #1?
  - Comments please?
  - SITN students feel free to email comments to sneaker@stanford.edu
- Reminder:
  - Still have late days!
    - Don't panic if you haven't finished yet
    - Plan accordingly for future assignments



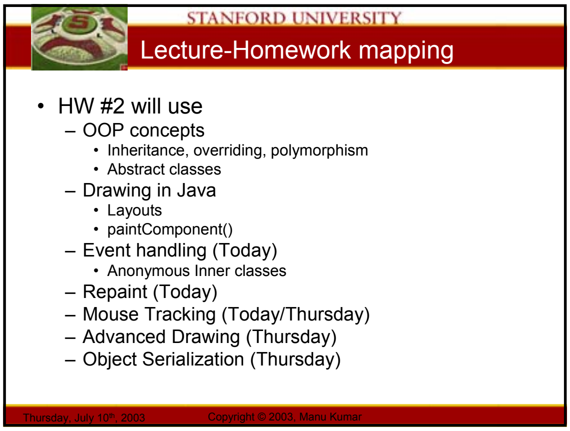
Handouts

- 3 Handout for today!
  - #14: HW 2: JavaDraw
    - Due before midnight Wednesday July 23rd, 2003
  - #15: Repaint
  - #16: Mouse



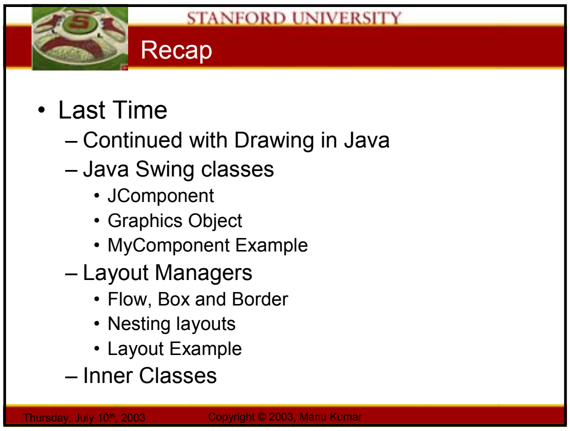
Homework #2: Java Draw Demo

- Live demo of the solution to HW#2
- Tips
  - Make sure to read the handout *several* times
  - Design first, code later
    - Spend time in designing your classes on paper
    - Use diagrams, sketches
  - You can never write all the code for all the functionality without incrementally compiling and testing!!
    - We give you working code!
    - Add functionality – Compile – Test – Repeat



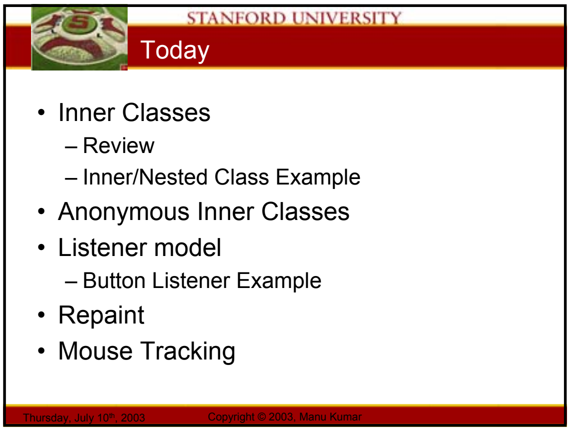
Lecture-Homework mapping

- HW #2 will use
  - OOP concepts
    - Inheritance, overriding, polymorphism
    - Abstract classes
  - Drawing in Java
    - Layouts
    - paintComponent()
  - Event handling (Today)
    - Anonymous Inner classes
  - Repaint (Today)
  - Mouse Tracking (Today/Thursday)
  - Advanced Drawing (Thursday)
  - Object Serialization (Thursday)



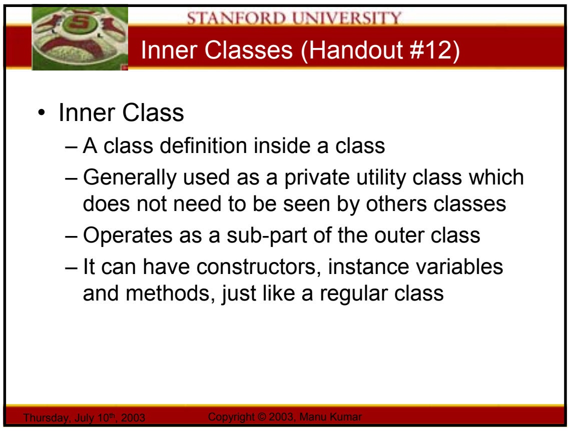
Recap

- Last Time
  - Continued with Drawing in Java
  - Java Swing classes
    - JComponent
    - Graphics Object
    - MyComponent Example
  - Layout Managers
    - Flow, Box and Border
    - Nesting layouts
    - Layout Example
  - Inner Classes



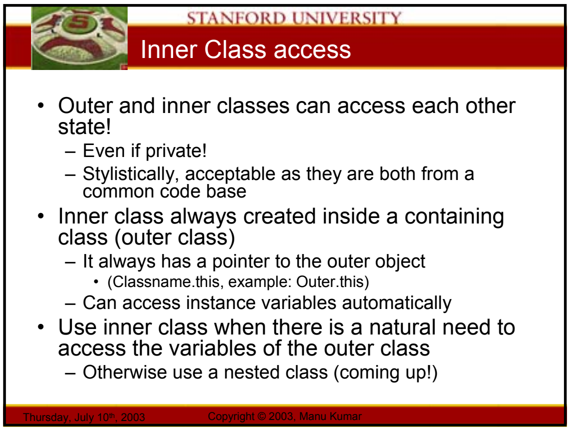
Today

- Inner Classes
  - Review
  - Inner/Nested Class Example
- Anonymous Inner Classes
- Listener model
  - Button Listener Example
- Repaint
- Mouse Tracking



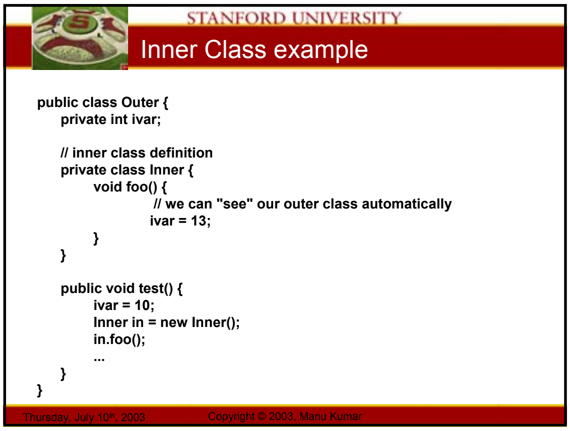
Inner Classes (Handout #12)

- Inner Class
  - A class definition inside a class
  - Generally used as a private utility class which does not need to be seen by others classes
  - Operates as a sub-part of the outer class
  - It can have constructors, instance variables and methods, just like a regular class



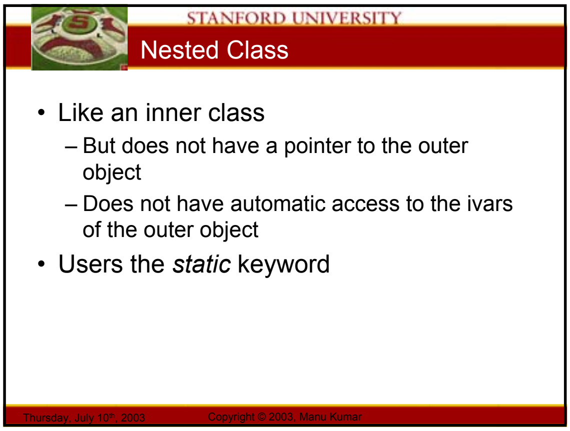
Inner Class access

- Outer and inner classes can access each other state!
  - Even if private!
  - Stylistically, acceptable as they are both from a common code base
- Inner class always created inside a containing class (outer class)
  - It always has a pointer to the outer object
    - (Classname.this, example: Outer.this)
  - Can access instance variables automatically
- Use inner class when there is a natural need to access the variables of the outer class
  - Otherwise use a nested class (coming up!)



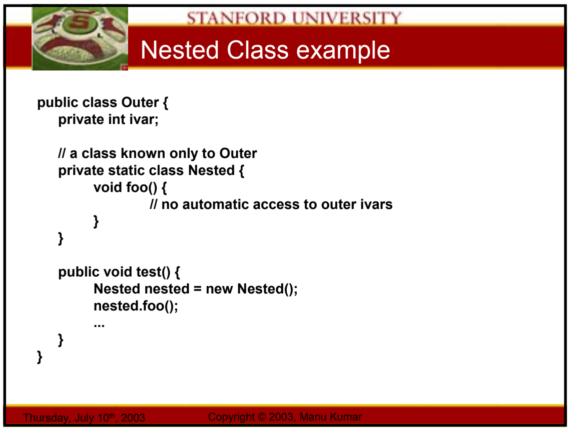
Inner Class example

```
public class Outer {
    private int ivar;

    // inner class definition
    private class Inner {
        void foo() {
                // we can "see" our outer class automatically
                ivar = 13;
        }
    }

    public void test() {
        ivar = 10;
        Inner in = new Inner();
        in.foo();
        ...
    }
}
```



Nested Class

- Like an inner class
  - But does not have a pointer to the outer object
  - Does not have automatic access to the ivars of the outer object
- Users the *static* keyword



Nested Class example

```
public class Outer {
    private int ivar;

    // a class known only to Outer
    private static class Nested {
        void foo() {
                // no automatic access to outer ivars
        }
    }

    public void test() {
        Nested nested = new Nested();
        nested.foo();
        ...
    }
}
```

STANFORD UNIVERSITY

Inner/Nested Example

- Each inner object is created in the context of a single, "owning", outer object
  - At runtime, the inner object has a pointer to its outer object which allows access to the outer object.
- Each inner object can access the ivars/methods of its outer object
  - Can refer to the outer object using its classname as "Outer.this".
- The inner/outer classes can access each other's ivars and methods, even if they are "private“
  - Stylistically, the inner/outer classes operate as a single class that is superficially divided into two.

STANFORD UNIVERSITY

Inner/Nested Example Code

```
// Outer.java

public class Outer {
    private int a;

    private void increment() {
        a++;
    }

    private class Inner extends Object {
        private int b;

        private Inner(int initB)  {
            b = initB;
        }
```

STANFORD UNIVERSITY

Inner/Nested Example Code

```
        private void demo() {
            // access our own ivar
            System.out.println("b: " + b);

            // access the ivar of our outer object
            System.out.println("a: " + a);

            // message send can also go to the outer object
            increment();

            /*
             Outer.this refers to the outer object, so could say
             Outer.this.a or Outer.this.increment()
            */
        }
    }
```

STANFORD UNIVERSITY

Inner/Nested Example Code

```
    // Nested class is like an inner class, but
    // without a pointer to the outer object.
    // (uses the keyword "static")
    private static class Nested {
        private int c;

        void demo() {
            c = 11;   // this works
            // b = 13;        // no does not compile --
            // nested object does not have pointer
            // to outer object
        }
    }
```

STANFORD UNIVERSITY

Inner/Nested Example Code

```
    public void test() {
        a = 10;
        Inner i1 = new Inner(1);
        Inner i2 = new Inner(2);

        i1.demo();
        i2.demo();

        Nested n = new Nested();
        n.demo();

    }

    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.test();
    }
}
```

STANFORD UNIVERSITY

Inner/Nested Example Output

Output:

```
b: 1
a: 10
b: 2
a: 11
```

Listeners (Handout #13)

- Anonymous Inner Classes
  - An inner class created on the fly using a quick and dirty syntax (no name!)
  - Convenient for creating small inner classes which play the role of callback function pointers (will see an example soon)
  - When compiled they look like Outer$1, Outer$2
- Stylistic notes
  - Useful for small sections of code
  - If it requires non-trivial ivars or methods, then a true inner class is better

Anonymous Inner Classes

- Do not have a name
- Does not have a constructor
  - Relies on the default constructor of the super class
- Does not have access to local stack variables (parameters to a method)
  - Unless they are declared final
- Example
  - Class Outer. Anonymous Inner class subclassed off of a class called Superclass

Anonymous Inner Class Example

```
public class Outer {
    int ivar;

    public Superclass method() {
        int sum;                        // ordinary stack var
        sum = ivar + 1;
        final int temp = ivar + 1;      // stack var, but declared final (constant)
        // Create new anonymous inner class, subclassed off Superclass
        Superclass s = new Superclass() {
                private int x = 0;
                public void foo() {
                        x++;                    // x of inner class
                        ivar++;                 // ivar of outer class
                        bar();                  // inherited from Superclass
                        // x = sum;             // no, cannot see sum
                        x = temp;               // this works, since temp is final
                }
        };
        return(s);              // later on, someone can send s.foo()
    }
...
```

final var trick

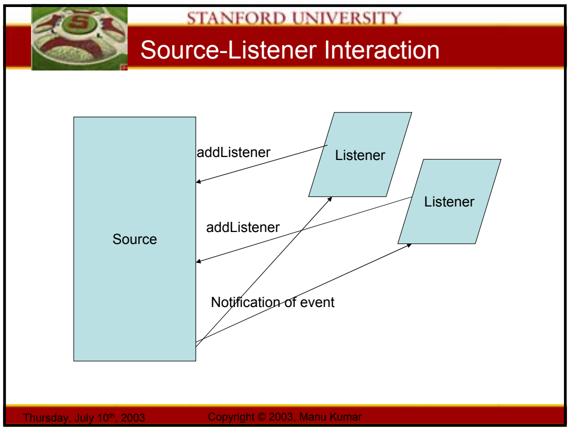
- Inner classes can see ivars of outer objects
- Inner classes **cannot** see stack variables (parameters)
- However
  - Inner classes can see “final” stack variables
- Why
  - Inlining of finals by the compiler
- *Declare stack variables as final to communicate their value to an anonymous inner class*
- Outer.this os the pointer to the outer object

Controls and Listeners

- Theory
  - Source
    - Buttons, controls etc.
  - Listener
    - An Object that wants to know when the control is operated
  - Notification Message
    - A message sent from the source to the listener as a notification that the event has occured
- Essentially: registering callbacks

Listeners and Interface

- An Object may be interested in multiple events
  - It can implement multiple listener interfaces
- Once an object implements a listener interface, it can then be added to the source buy using
  - source.addListener(Listener l)
- Interfaces are key in the ability to implement the Listener model



Source-Listener Interaction

Source

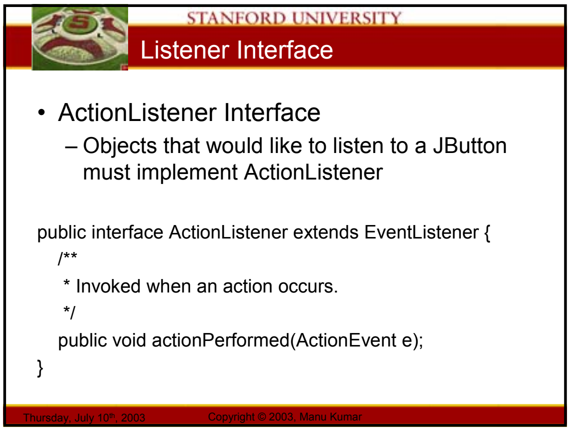
addListener

addListener

Listener

Listener

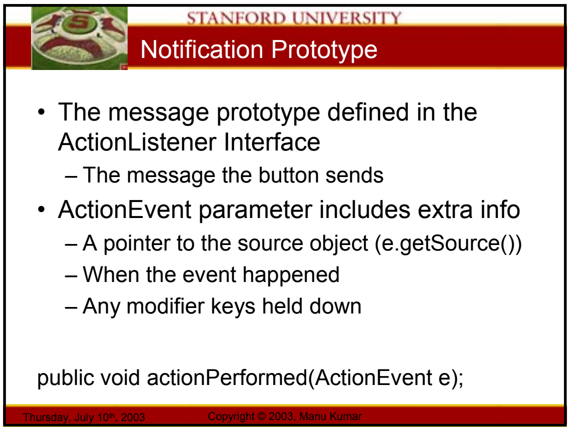
Notification of event



Listener Interface

- ActionListener Interface
  - Objects that would like to listen to a JButton must implement ActionListener

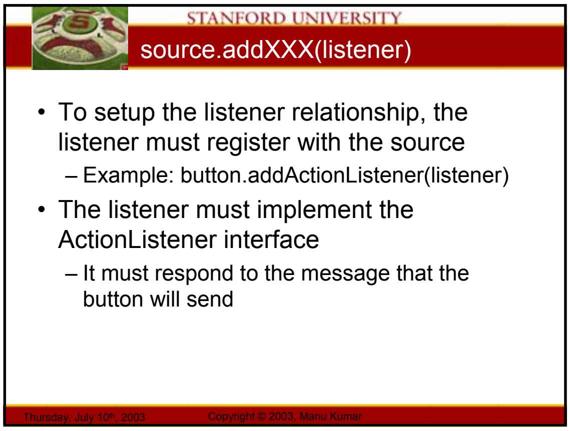
```
public interface ActionListener extends EventListener {
   /**
    * Invoked when an action occurs.
    */
   public void actionPerformed(ActionEvent e);
}
```



Notification Prototype

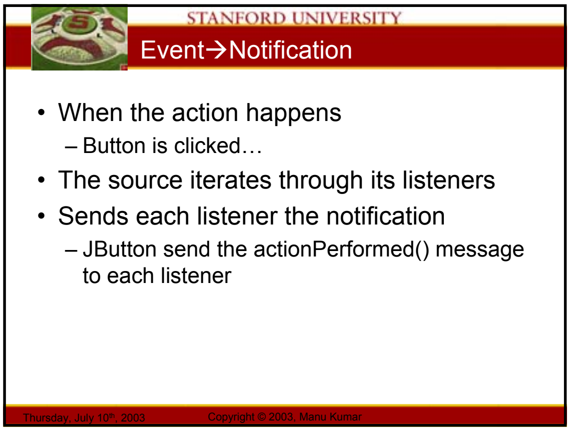
- The message prototype defined in the ActionListener Interface
  - The message the button sends
- ActionEvent parameter includes extra info
  - A pointer to the source object (e.getSource())
  - When the event happened
  - Any modifier keys held down

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



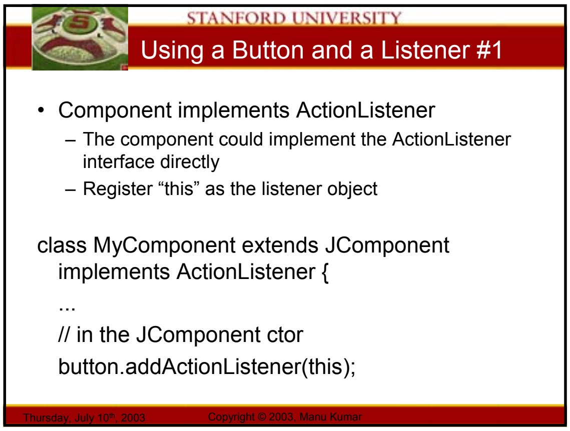
source.addXXX(listener)

- To setup the listener relationship, the listener must register with the source
  - Example: button.addActionListener(listener)
- The listener must implement the ActionListener interface
  - It must respond to the message that the button will send



Event→Notification

- When the action happens
  - Button is clicked…
- The source iterates through its listeners
- Sends each listener the notification
  - JButton send the actionPerformed() message to each listener



Using a Button and a Listener #1

- Component implements ActionListener
  - The component could implement the ActionListener interface directly
  - Register “this” as the listener object

```
class MyComponent extends JComponent
   implements ActionListener {
   ...
   // in the JComponent ctor
   button.addActionListener(this);
```

Using a Button and a Listener #2

- Create an inner class
  - Create a MyListener inner class which implements ActionListener
  - Create a new MyListener object
  - Add it via button.addXXX(listener)

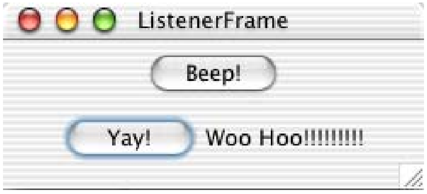
```
// in the JComponent ctor
ActionListener listener = new MyActionListener();
button.addActionListener(listener);
```

Using a Button and a Listener #3

- Anonymous Inner class
  - Most common method!
  - Create an Anonymous Inner Class that implements the interface
    - Can be created on the fly inside the method!

```
button = new JButton("Beep");
panel.add(button);
button.addActionListener(
    new ActionListener() {
        public void actionPerformed(ActionEvent e) {
                Toolkit.getDefaultToolkit().beep();
        }
    }
);
```

Button Listener Example



ButtonListener Example Code

```
// ListenerFrame.java
import java.awt.*;
import javax.swing.*;
import javax.swing.event.*;
import java.awt.event.*;
/*
 Demonstrates bringing up a frame with a couple of buttons in it.
 Demonstrates using anonymous inner class listener.
*/
public class ListenerFrame extends JFrame {
    private JLabel label;
```

Button Listener Example

```
public ListenerFrame() {
        super("ListenerFrame");

        JComponent content = (JComponent) getContentPane();
        content.setLayout(new FlowLayout());

        JButton button = new JButton("Beep!");
        content.add(button);

        // ----
        // Creating an action listener in 2 steps...

        // 1. Create an inner class subclass of ActionListener
        ActionListener listener =
                new ActionListener() {
                        public void actionPerformed(ActionEvent e) {
                                Toolkit.getDefaultToolkit().beep();
                        }
                };
```

Button Listener Example

```
// 2. Add the listener to the button
button.addActionListener(listener);

// ----
// Creating a listener in 1 step...

// Create a little panel to hold a button
// and a label
JPanel panel = new JPanel();
content.add(panel);
JButton button2 = new JButton("Yay!");
label = new JLabel("Woo Hoo");
panel.add(button2);
panel.add(label);
```

Button Listener Example

```
// This listener adds a "!" to the label.
        button2.addActionListener(
                    new ActionListener() {
                                public void actionPerformed(ActionEvent e) {
                                            String text = label.getText();
                                            label.setText(text + "!");
                                            // note: we have access to "label" of
outer class
                                            // we do not have access to local vars
like 'panel',
                                            // unless they are declared final.
                                }
                    }
        );

        pack();
        setVisible(true);
}
```

Misc Listeners

- JCheckBox
  - Uses ActionListener, like JButton
  - Responds to boolean isSelected() to see if it is currently checked
- JSlider
  - Component with min/max/current values
  - Users StateChangedListener interface
    - Notification is stateChanged(ChangeEvent e)
    - e.getSource() to get a pointer to the source
  - Responds to int getValue() to get current value

Event handling Strategies

- Listener strategy
  - Our approach so far
  - Event based
- Polling strategy
  - Do not listen to the control
  - Check the value when you choose
  - Often fraught with problems, but may have an appropriate use in some cases

Repaint (Handout #15)

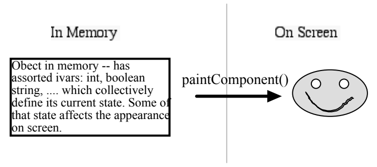
- How does a GUI work?
  - Objects in memory, storing state as strings, ints, pointers
  - System sends paintComponent() messages to Objects
  - Objects draw themselves
  - System maps user clicks, keystrokes etc. to notification messages sent to the objects
    - Object register interest in certain messages
    - Objects react to messages
    - Appears to user that their actions caused the change

paintComponent()

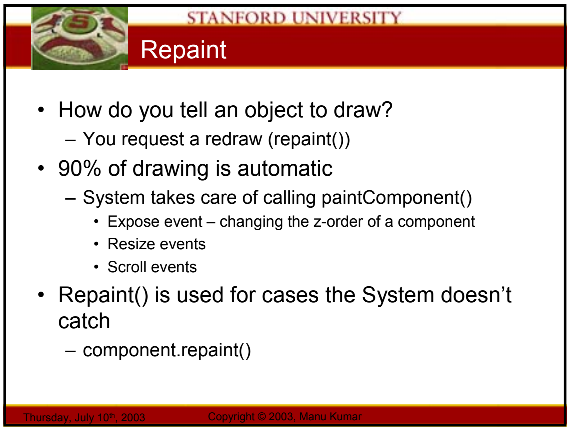
- paintComponent() is System driven
  - You do not call paintComponent
  - The System calls it when needed
- Debugging paintComponent()…
  - Add a g.drawRect() in the first line
    - Make sure it is being called
    - Similar to using System.out.println() in text mode
      - Can also use System.out.println() and look at the console
  - Check height and width of the component
  - Add a beep
    - Toolkit.getDefaultToolkit().beep()
  - Press CTRL-SHIFT-F1 to get a debugging dump

paintComponent()

- paintComponent()
  - Looks at the state of the object
  - Draws the pixels that represent that state
- Cardinal rule for paintComponent()
  - Should not modify the state of the object
  - paintComponent should be read-only



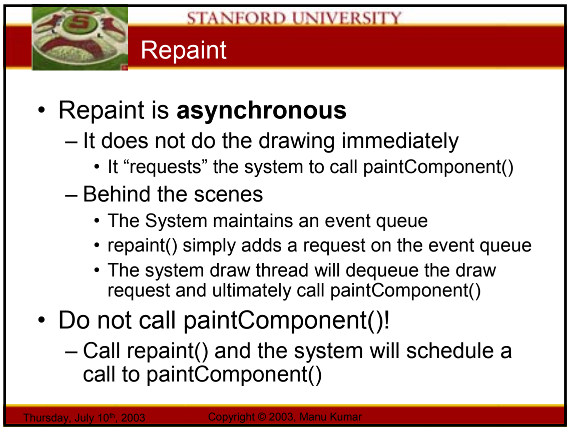
Obect in memory -- has assorted ivars: int, boolean string, ... which collectively define its current state. Some of that state affects the appearance on screen.



STANFORD UNIVERSITY

Repaint

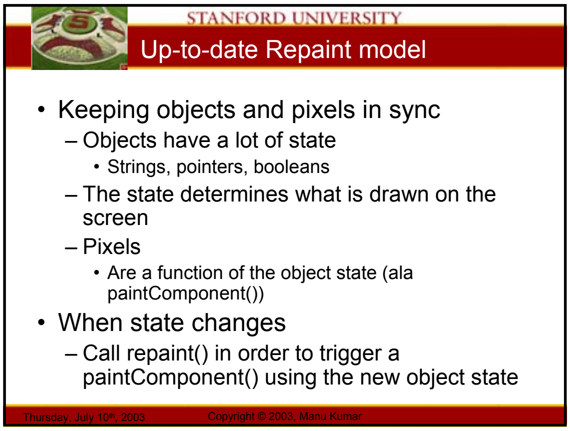
- How do you tell an object to draw?
  - You request a redraw (repaint())
- 90% of drawing is automatic
  - System takes care of calling paintComponent()
    - Expose event – changing the z-order of a component
    - Resize events
    - Scroll events
- Repaint() is used for cases the System doesn't catch
  - component.repaint()



STANFORD UNIVERSITY

Repaint

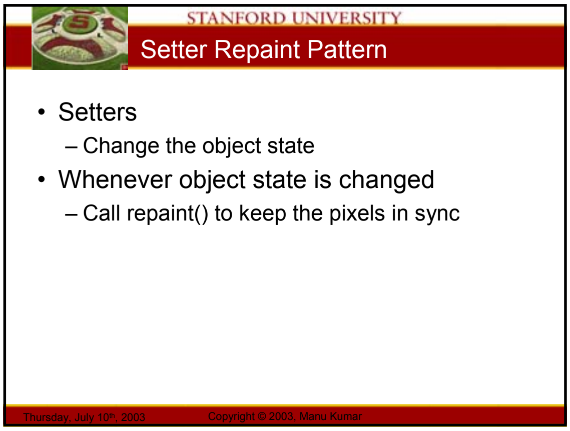
- Repaint is **asynchronous**
  - It does not do the drawing immediately
    - It "requests" the system to call paintComponent()
  - Behind the scenes
    - The System maintains an event queue
    - repaint() simply adds a request on the event queue
    - The system draw thread will dequeue the draw request and ultimately call paintComponent()
- Do not call paintComponent()!
  - Call repaint() and the system will schedule a call to paintComponent()



STANFORD UNIVERSITY

Up-to-date Repaint model

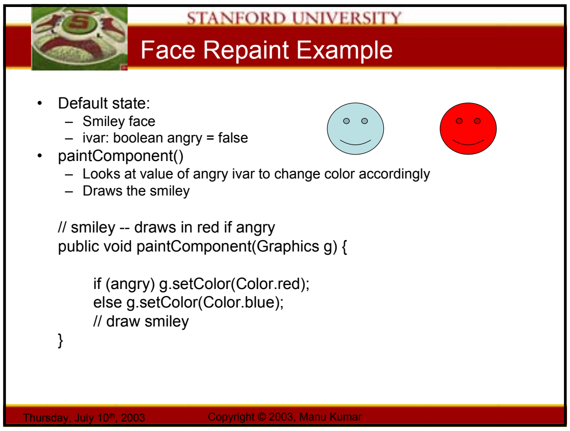
- Keeping objects and pixels in sync
  - Objects have a lot of state
    - Strings, pointers, booleans
  - The state determines what is drawn on the screen
  - Pixels
    - Are a function of the object state (ala paintComponent())
- When state changes
  - Call repaint() in order to trigger a paintComponent() using the new object state



STANFORD UNIVERSITY

Setter Repaint Pattern

- Setters
  - Change the object state
- Whenever object state is changed
  - Call repaint() to keep the pixels in sync

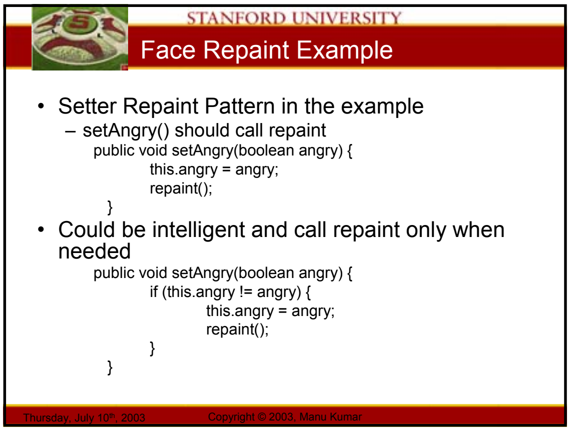


STANFORD UNIVERSITY

Face Repaint Example

- Default state:
  - Smiley face
  - ivar: boolean angry = false
- paintComponent()
  - Looks at value of angry ivar to change color accordingly
  - Draws the smiley

```
// smiley -- draws in red if angry
public void paintComponent(Graphics g) {

        if (angry) g.setColor(Color.red);
        else g.setColor(Color.blue);
        // draw smiley
}
```



STANFORD UNIVERSITY

Face Repaint Example

- Setter Repaint Pattern in the example
  - setAngry() should call repaint

```
    public void setAngry(boolean angry) {
                this.angry = angry;
                repaint();
    }
```

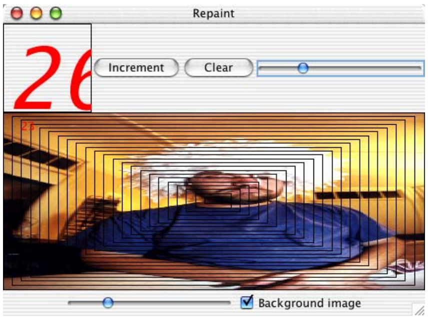
- Could be intelligent and call repaint only when needed

```
    public void setAngry(boolean angry) {
                if (this.angry != angry) {
                        this.angry = angry;
                        repaint();
                }
    }
```

Repaint tips

- Remember
  - Change in object state → call repaint
- Design tips
  - Good client design means that the programmer shouldn't have to remember when to call repaint
    - Your code should do it at the right time
  - Tempting to sprinkle repaint calls
    - Performance hit. Be smart about it.
  - What happens if paintComponent() calls repaint()?
    - "Bad things happen"

Repaint Example



Repaint Example Code

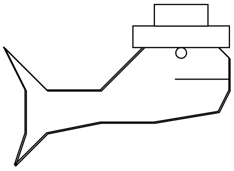
- Code walk through….
  - Widget.java
  - Boxer.java
  - Repaint.java
    - Layout
    - Event handling with listeners

Erasing

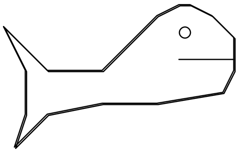
- We do not actively erase in java
  - To erase something, simply don't draw it in paintComponent
- paintComponent starts out with a erased canvas
  - Draws components back to front
    - *What you draw later is drawn on top*
- Again
  - To erase something, just don't draw it

Fish Example

- Fish with a hat



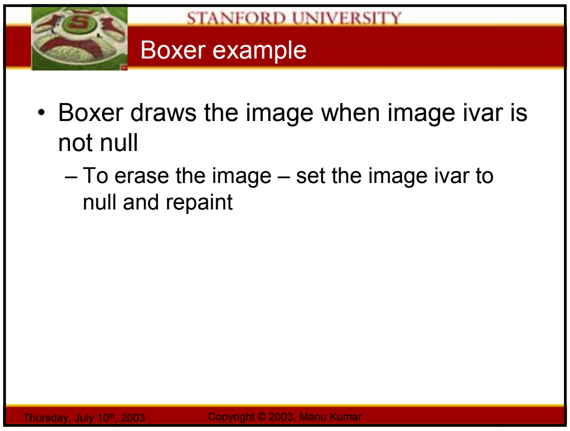
- Fish without a hat



The Fish class…

```
void paintComponent() {
  // draw fish body
  if (hasHat) // draw the hat
}
void setHat(boolean hat) {
  hasHat = hat;
  repaint();
}
```

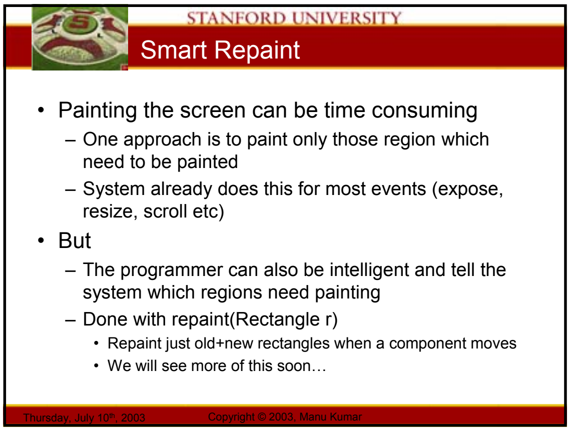
- Scenario: fish.hasHat is true. Send fish.setHat(false) -- the hat disappears



STANFORD UNIVERSITY

Boxer example

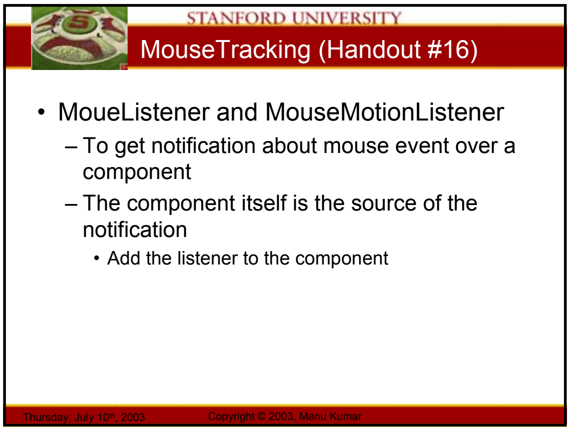
- Boxer draws the image when image ivar is not null
  - To erase the image – set the image ivar to null and repaint



STANFORD UNIVERSITY

Smart Repaint

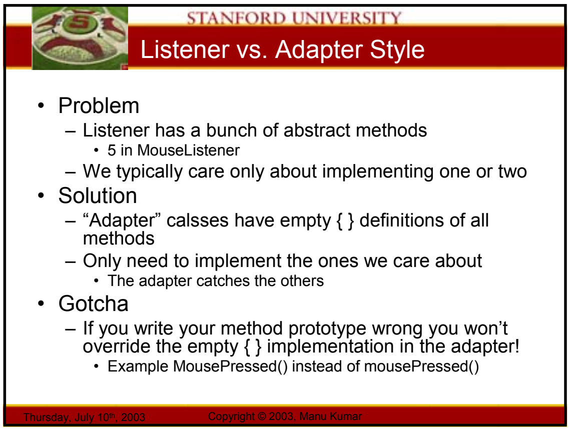
- Painting the screen can be time consuming
  - One approach is to paint only those region which need to be painted
  - System already does this for most events (expose, resize, scroll etc)
- But
  - The programmer can also be intelligent and tell the system which regions need painting
  - Done with repaint(Rectangle r)
    - Repaint just old+new rectangles when a component moves
    - We will see more of this soon…



STANFORD UNIVERSITY

MouseTracking (Handout #16)

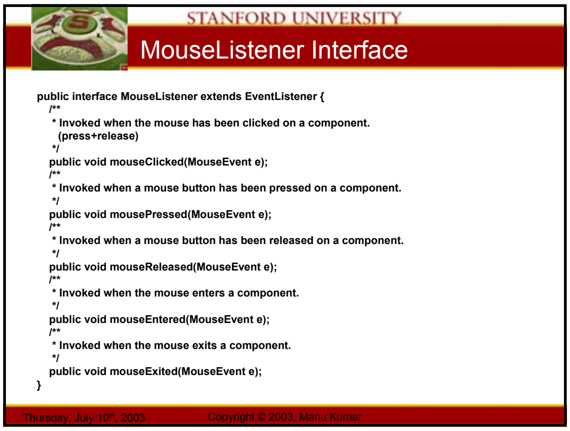
- MoueListener and MouseMotionListener
  - To get notification about mouse event over a component
  - The component itself is the source of the notification
    - Add the listener to the component



STANFORD UNIVERSITY

Listener vs. Adapter Style

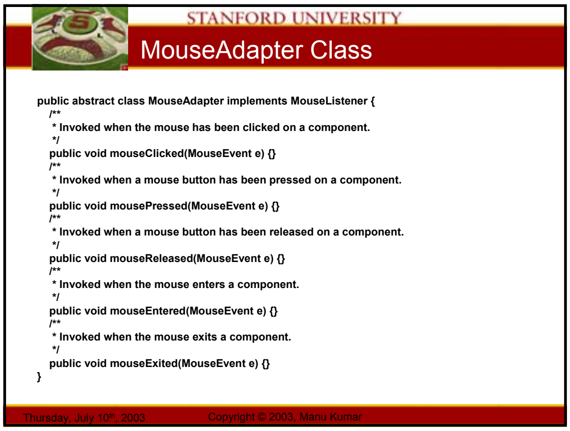
- Problem
  - Listener has a bunch of abstract methods
    - 5 in MouseListener
  - We typically care only about implementing one or two
- Solution
  - “Adapter” calsses have empty { } definitions of all methods
  - Only need to implement the ones we care about
    - The adapter catches the others
- Gotcha
  - If you write your method prototype wrong you won’t override the empty { } implementation in the adapter!
    - Example MousePressed() instead of mousePressed()



STANFORD UNIVERSITY

MouseListener Interface

```
public interface MouseListener extends EventListener {
    /**
     * Invoked when the mouse has been clicked on a component.
       (press+release)
     */
    public void mouseClicked(MouseEvent e);
    /**
     * Invoked when a mouse button has been pressed on a component.
     */
    public void mousePressed(MouseEvent e);
    /**
     * Invoked when a mouse button has been released on a component.
     */
    public void mouseReleased(MouseEvent e);
    /**
     * Invoked when the mouse enters a component.
     */
    public void mouseEntered(MouseEvent e);
    /**
     * Invoked when the mouse exits a component.
     */
    public void mouseExited(MouseEvent e);
}
```



STANFORD UNIVERSITY

MouseAdapter Class

```
public abstract class MouseAdapter implements MouseListener {
    /**
     * Invoked when the mouse has been clicked on a component.
     */
    public void mouseClicked(MouseEvent e) {}
    /**
     * Invoked when a mouse button has been pressed on a component.
     */
    public void mousePressed(MouseEvent e) {}
    /**
     * Invoked when a mouse button has been released on a component.
     */
    public void mouseReleased(MouseEvent e) {}
    /**
     * Invoked when the mouse enters a component.
     */
    public void mouseEntered(MouseEvent e) {}
    /**
     * Invoked when the mouse exits a component.
     */
    public void mouseExited(MouseEvent e) {}
}
```

Press: MouseListener

- How does a component handle a mouse press?

```
component.addMouseListener(new MouseAdapter() {
        public void mousePressed(MouseEvent e) {
          // called when mouse button first pressed on component
        }
        });
```

Motion: MouseMotionListener

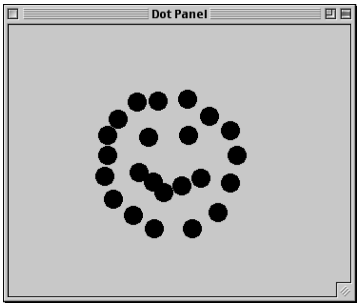
- How does a component detect a mouse movement?

```
component.addMouseMotionListener(new MouseMotionAdapter() {
     public void mouseDragged(MouseEvent e) {
       // called as mouse is dragged, after initial click
     }
     });
```

Delta rule for mouse motion

- Cannot use absolute coordinates for mouse movement!
  - Setting the position to the actual mouse coordinated may result is weird movements
- Correct approach
  - Get the current coordinates
  - Compare to the last known coordinates
    - Compute the delta
  - Apply the delta to the position of the object
- Test-case
  - A click-release with no motion should not change any state in a correct implementation of relative mouse tracking

DotPanel Example



DotPanel Example Code

- Code walkthrough…
  - DotPanel.java

Summary

- Today
  - Inner Classes
    - Review
    - Inner/Nested Class Example
  - Anonymous Inner Classes
  - Listener model
    - Button Listener Example
  - Repaint
  - Mouse Tracking
- Assigned Work
  - HW 2: Java Draw
    - Due before midnight on Wednesday, July 23rd, 2003
    - Start early!!