Drawing 3

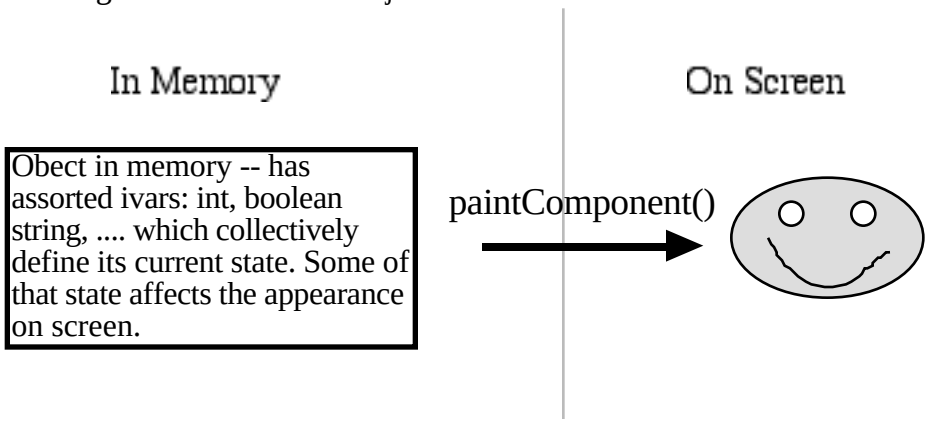
paintComponent() Features

1. Passive / System Driven

Wait for system to tell you to draw
drawRect() debugging -- put a call to g.drawRect() at the start of your paintComponent() just to see where things are.
Do not just start drawing on your own, they way you would in a 106a program

2. Object state -> pixels, read only

Look a the state of the object, and draw the pixels that represent that state. Do not change the state of the object.



3. Clipping

System sets a "clipping region" on the Graphics object before sending paintComponent(). By default, the clipping region is the bounds of the component, so the component does not draw outside its bounds.
Drawing outside the clipping region does not lay down any pixels.
System uses this to optimize the drawing in cases we will see later.
The component can be unaware of the clipping for basic uses -- e.g. g.drawRect() automatically only changes pixels inside the clipping region.

4. float / int drawing

Suppose you want to draw 10, equally spaced vertical lines
NO:

```
int dh = width / 10;
for (int i=0; i<10; i++) {
  int x = dh*i;
```

```
    g.drawSomething(x, ...
YES:
    Do calculations with floats, convert to int only at the last step
    double dh = ((double)width) / 10;
    for (int i=0; i<10; i++) {
      int x = Math.round(dh*i);
      g.drawSomething(x, ...                  / / convert to int at the last step
```

5. component.getGraphics() -- NO

Almost always incorrect to use this
Only getGraphics() if the assignment handout specifically says to
getGraphics() goes against the system / paintComponent paradigm

Repaint

90% of drawing is automatic

90% of drawing is automatic, you don't do anything at all -- the system notes these cases automatically and does the drawing...
- Expose event-- something used to be covered in the "z-order" stacking of components, but now is not
- Resizing
- Scrolling

We need repaint for the cases where the system does not automatically know that the components needs to be redrawn.

component.repaint() -- draw request

Send the repaint() message to tell the system that the given component needs to be redrawn. In other words, repaint() will cause the system to redraw the component.

Asynchronous

Repaint() does not cause the drawing immediately.
The connection from repaint() to paintComponent() is indirect.
Repaint() simply notes that given component needs a redraw in the system's private data structure.
A fraction of a second later, the draw thread in the system will notice that the component needs to be drawn, and will initiate calling paintComponent()

Do Not call paintComponent()

It is almost never correct to call component.paintComponent()
Instead, call component.repaint(), and the system will schedule a paintComponent() to happen soon

"Up To Date" Repaint Model

Object State

Each object in memory has lots of state : strings, pointers, booleans...

Some of that state affects the way the object appears on screen.

Relevant Change

When state that affects the appearance is changed, a repaint() is required.

Out of date

A change to the object state makes the on-screen representation out of date -- it is showing the pixels from a paintComponent() with the old state.

e.g. Repaint Setter Style

boolean angry

Suppose the smiley face has an angry boolean.

paintComponent() looks at the value of the angry ivar, and draws in red if it is true

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

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

The setter does a repaint() since the angry state is relevant to the appearance

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

Better

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

Work for Client NO / Work for receiver YES

Some state is relevant to the appearance and some is not.

Do not make the client figure this out -- just hide the call to repaint() in the appropriate setters.

Repaint Bugs

Tempting to sprinkle repaint() calls all around -- don't be careless

What if the code calls repaint() in paintComponent()?

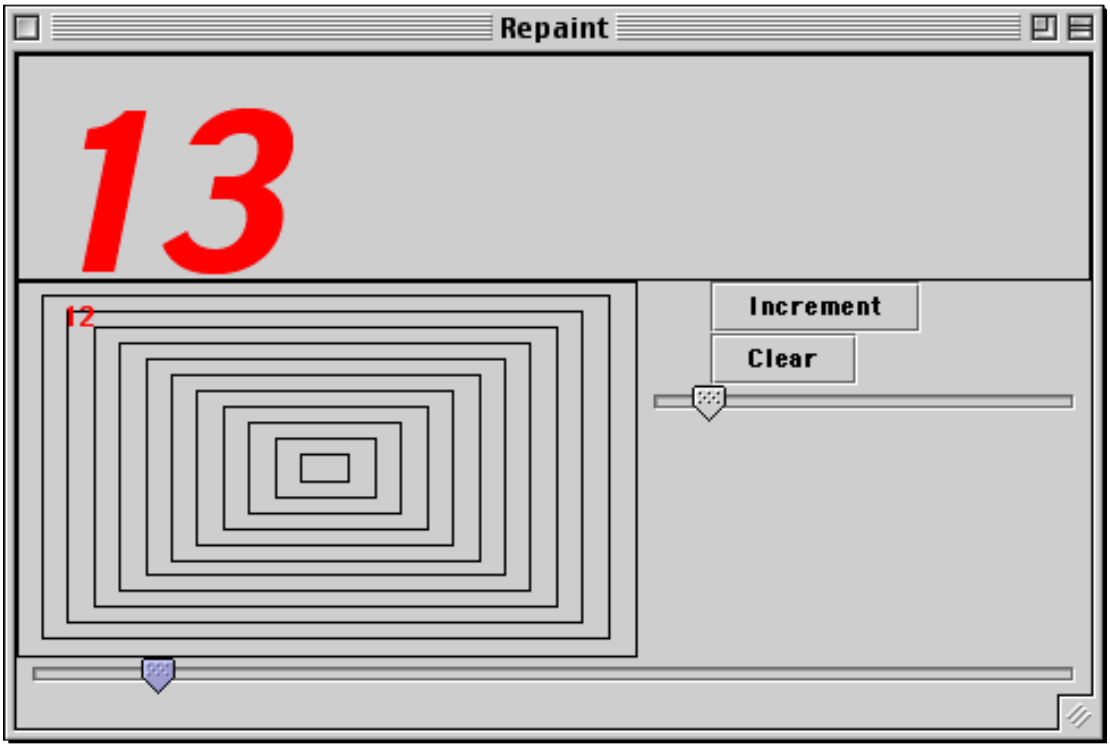
More Listeners - JSlider

JSlider -- component with min / max / current int values
Listener implements StateChangeListener interface -- the notification is called stateChanged(ChangeEvent e)
Use e.getSource() to get a pointer to the source object

final var trick

Inner classes can see ivars of outer object
Inner classes **cannot** see stack vars from where they are created.
However, inner classes **can see** "**final**" **stack vars** from where they are created -- use this to communicate the value of a stack var to the inner class.
Use "Outer.this" to refer to the this pointer of the outer object. Necessary in some cases if the compiler cannot distinguish that a ivar should be available from the outer object.

Repaint Example



/ / Widget.java

```
/*
 Stores a single number.
 Draws that number with a large font.
 Simple example of setter/repaint style.
*/
import java.awt.*;
import javax.swing.*;
import javax.swing.event.*;
import java.awt.event.*;


public class Widget extends JComponent {
   private int count;

   // static: one variable shared by all instances
   // aka "singleton" pattern
   private static Font font = null;

   public Widget(int width, int height) {
      super();
      setPreferredSize(new Dimension(width, height));

      count = 0;
   }

   /*
    Typical setter -- calls repaint() to alert the
    system that we need to be redrawn.
   */
   public void setCount(int newCount) {
      if (newCount!=count) {
         count = newCount;
         repaint();
      }
   }


   public void increment() {
      setCount(count+1);
   }

   /*
    Draw ourselves with a big font (see the Font class).
   */
   public void paintComponent(Graphics g) {
      // typical "debug rect" around our bounds just to have
      // something show up
      g.drawRect(0, 0, getWidth()-1, getHeight()-1);

      // trick: lazy evaluation of font object
      if (font==null) font = new Font("DIALOG", Font.ITALIC, 96);

      g.setFont(font);
      g.setColor(Color.red);
      g.drawString(Integer.toString(count), 4, getHeight()-4);
```

```
   }

}
```

/ / Boxer.java

```
/*
 Simple component draws a number of boxes.
*/
import java.awt.*;
import javax.swing.*;
import java.util.*;

import java.awt.event.*;

class Boxer extends JComponent {
   private int count;   // number of boxes to draw

   Boxer(int width, int height) {
      super();
      setPreferredSize(new Dimension(width, height));

      count = 1;
   }


   /*
    Increases the count.
   */
   public void increment() {
      setCount(count+1);
   }


   /*
    Sets the count.
   */
   public void setCount(int count) {
   // note: tricky case of param and ivar with same name
      if (this.count != count) {
         this.count = count;
         repaint();
      }
   }

   /*
    Draws the series of 1..count rectangles
   */
   public void paintComponent(Graphics g) {
      //super.paintComponent(g); // not necessary

      int width = getWidth();
      int height = getHeight();

      for (int i=0; i<count; i++) {
         // note: do the running i/count computation with floats
```

```
         // 0/10 1/10 2/10 ...
         int rx = (int) ((float)width*i/(2*count));
         int ry = (int) ((float)height*i/(2*count));

         // 5/5 4/5 3/5...
         int rWidth = (int) ((float)width*(count-i))/count;
         int rHeight = (int) ((float)height*(count-i))/count;

         g.drawRect(rx, ry, rWidth-1, rHeight-1);
      }

      g.setColor(Color.red);
      g.drawString(Integer.toString(count), 20, 20);
   }

}
```

/ / Repaint.java

```
/*
 Demonstrates a frame containing a widget and boxer with
 controls wired up to them.
*/
import java.awt.*;
import javax.swing.*;
import javax.swing.event.*;
import java.awt.event.*;


public class Repaint extends JFrame {
   private Widget widget;
   private Boxer boxer;

   public Repaint() {
      super("Repaint");

      JComponent container = (JComponent) getContentPane();

      // Put in a border layout
      container.setLayout(new BorderLayout());

      // Put a widget in the north
      widget = new Widget(100, 100);
      container.add(widget, BorderLayout.NORTH);


      JButton button;


      // Create a vertical box, put it in the east, put controls in it
      Box box = Box.createVerticalBox();
      container.add(box, BorderLayout.EAST);

      button = new JButton("Increment");
      box.add(button);
      button.addActionListener(
```

```
   new ActionListener() {
      public void actionPerformed(ActionEvent e) {
         // note: we can access ivars of our "outer" object
         widget.increment();
      }
   }
);

button = new JButton("Clear");   // note: re-using "button" var
box.add(button);
button.addActionListener(
   new ActionListener() {
      public void actionPerformed(ActionEvent e) {
         widget.setCount(0);
      }
   }
);

// note: store slider as "final" variable so inner class
// can see it
final JSlider slider = new JSlider(0, 100, 0);  // (min, max, current)
box.add(slider);

slider.addChangeListener(
   new ChangeListener() {
      public void stateChanged(ChangeEvent e) {
         // note: can access final stack var "slider"
         widget.setCount(slider.getValue());
      }
   }
);


// Create boxer in center with slider in south
final Boxer b = new Boxer(200,200);
container.add(b, BorderLayout.CENTER);

JSlider slider2 = new JSlider(0, 100, 0);
container.add(slider2, BorderLayout.SOUTH);

slider2.addChangeListener(
   new ChangeListener() {
      public void stateChanged(ChangeEvent e) {
         // note: another way to get a pointer to the source
         JSlider s = (JSlider) e.getSource();
         b.setCount(s.getValue());
      }
   }
);


// Quit on window close
addWindowListener(
   new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
         System.exit(0);
      }
   }
```

```
      );
      // in 1.2 this works: frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      pack();
      setVisible(true);

   }

   public static void main(String[] args) {
      new Repaint();
   }
}
```

Control Strategies...

1. Listener

The way we've done things so far.
Get notifications from the button, slider, etc. at the time of the change

2. Poll

Another technique -- do not listen to the control. Instead, check the control's value at the time of your choosing
e.g. checkbox.isSelected()
Avoid having two copies of the control's state -- just use the one copy in the control itself.
Polling does not work if you need to repaint on control change.

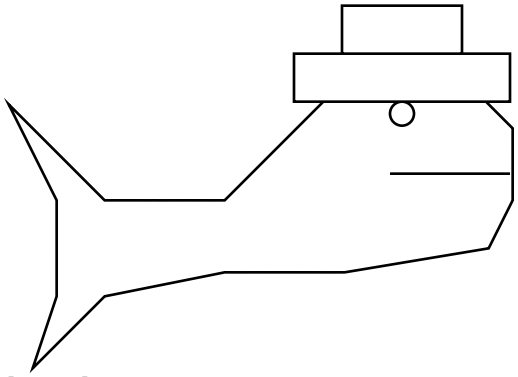
Erasing

We don't actively erase things.

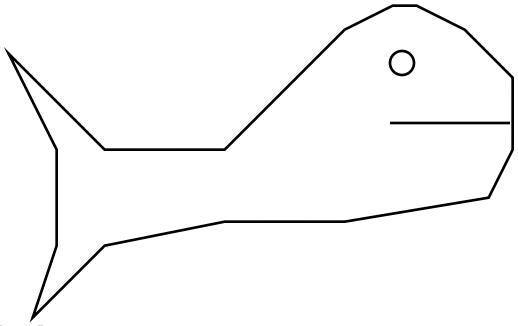
To "erase" something, we just don't draw it in paintComponent(), and so it disappears

When calling paintComponent(), the system starts with an erased canvas, and draws the components back to front. To make something disappear -- just don't draw it.

With hat



Without hat



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.