Draw, Paint, Repaint

Thanks to Nick Parlante for much of this handout

Drawing

OOP GUI Drawing Theory

- Subclass off `JComponent` (lightweight) or `JPanel` (heavier)
    - `JComponent` does not have a background -- it's transparent plus whatever it draws in its own `paintComponent()`.
    - `JPanel` can have a colored or whatever background (like a canvas) plus whatever it draws in its `paintComponent()`

- Override `paintComponent()` -- draw within your current bounds (call `getWidth()` etc. to discover your bounds)

- Install your component in a window -- it draws itself

paintComponent(Graphics g)

- Notification sent to a `JComponent` when it should draw itself

- The `Graphics g` is a graphics context that, in some sense, represents the on-screen pixels. Drawing on `g` goes to the screen, eventually.

- Override to provide custom drawing code

- Call `super.paintComponent(g)`

- Call `getWidth()` etc. to see the current geometry -- see how big you are

- (0,0) is your upper-left corner -- draw yourself within your bounds

- Do not need to erase first -- the drawing canvas is already erased to a default state before `paintComponent()` runs

- The system automatically "clips" our drawing to our bounds rectangle. Drawing outside the bounds does not lay down any pixels.

Simple paintComponent Example

```
public void paintComponent(Graphics g) {
    super.paintComponent(g);   // does nothing in simple JComponent subclassess

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

    // draw a rect around the bounds of the component
    g.drawRect(0, 0, width-1, height-1);  // -1 since drawRect overhangs by one

    // draw a line from upper-left, to lower-right
    g.drawLine(0, 0, width-1, height-1);
}
```

See How Big You Are

- Send self `getWidth()`, etc. to see how big you are -- draw to fill that size.

- We draw in a reasonable way, even as the window resizes -- our width and height are different.

"Respond To" Draw Style

- You don't demand to draw, you respond -- drawing when the system says to draw, dealing with however many pixels the system says you have.

- In contrast to the "imperative" draw style you might be used to from C, where you just start drawing on your own schedule.

- Passive drawing works better in a windowing system in which **when** to draw is complex -- on scroll, on window resize, etc..

No Erase Needed

- Don't have to erase first -- the `Graphics` is already erased for us. We just draw ourselves on the pre-erased `Graphics`.

Graphics Object

- A drawing context object passed to you -- send it drawing commands to do drawing.

- The `Graphics` is in a default state when `paintComponent()` is called -- it does not have state from earlier paints.

- See the docs for the `Graphics` class

- With AWT, `Graphics` is a simple int-based 2d system.

- There is also a more sophisticated Java2D floating point (PDF like) drawing system, but we won't worry about that.

- `(0,0)` in the upper left hand corner
  - X extends to the right
  - Y extends down

- `g.drawRect(x, y, width, height)`
  - Draws the frame of a rectangle with its upper left at `(x,y)`
  - Extends past the given width and height by 1 on the right and bottom, so you frequently subtract 1 when calling this. I think they were trying to appease some mathematical elegance with this design, but in fact it was just stupid.

- `g.fillRect(x, y, width, height)`
  - Uses the current color to fill a colored rect of the given size. Does not overhang the size by one.

- `drawLine(x1, y1, x2,  y2)` -- draws a one pixel wide line between the points

- `drawString(String, x, y)`
  - Draws the string, with the lower left of the text line at x,y. Use the `Font` class to draw with different font sizing etc.

- `g.setColor(Color)`
  - Sets the color for subsequent drawing.
  - There are constants in the `Color` class such as `Color.BLACK`, `Color.GREEN`, etc.

- `Component.getGraphics()`
  - You probably never want to call this. The correct style is to use the `Graphics` passed in to `paintComponent()`

Constructor -- setPreferredSize()

- Each component can have a preferred, max, and min size set

- The layout manager consults these values when laying everything out, e.g. when `pack()` is called

- A good strategy is to call `setPreferredSize()` in the constructor -- that way the values are available later on when `pack()` or whatever is called,
  e.g. `setPreferredSize(new Dimension(400, 200))`

- Do not call `setWidth()`, `setHeight()` etc. ... the layout manager owns the actual current width, height etc.

Debugging paintComponent()

- Put a call to `g.drawRect(0,0,getWidth()-1, getHeight()-1)` at the start of your `paintComponent()` just to see where things are.

- If nothing at all shows up, make sure the component is not width or height 0, and has been added to the frame.

- Make sure the prototype is exactly right, not paintCompomnent() or something.

- Put a `println()` or `Toolkit.getDefaultToolkit().beep()` to see if `paintComponent()` is getting called at all

- While a Swing program is running, type *ctrl-shift-f1* to get a debugging printout dump of the complete nesting of components installed in the frame.

Face Example
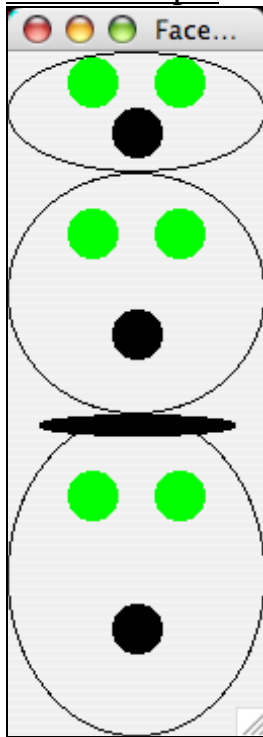

```
// Face.java
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import javax.swing.event.*;

/*
 Demonstrates a component that draws a face and uses
 the setter/repaint style.
*/
public class Face extends JComponent {
    // The "model" data for the face:
    // booleans and ints that control how it looks
    private boolean angry;
    private boolean hat;

    // Deltas added/subtracted from default eye/mouth size
    private int eyeDelta;
    private int mouthDelta;
```

```java
    private int mouthWidthDelta;

    Face(int width, int height) {
        setPreferredSize(new Dimension(width, height));

        angry = false;
        hat = false;
        eyeDelta = 0;
        mouthDelta = 0;
        mouthWidthDelta = 0;
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);

        // see how big we are
        int width = getWidth();
        int height = getHeight();

        // Angry -> draw red oval, otherwise plain outline
        if (angry) {
            g.setColor(Color.RED);
            g.fillOval(0, 0, width, height);
        } else {
            g.drawOval(0, 0, width - 1, height - 1);
        }

        int eyeY = height / 4; // place eyes 1/4 from top
        int mouthY = 2 * height / 3; // place mouth 2/3 from top

        // base eye size on width
        int eyeRadius = width / 10 + eyeDelta;

        // place the eyes at 1/3 and 2/3 from left
        int left = width / 3;
        int right = 2 * width / 3;

        // draw the eyes in green
        g.setColor(Color.GREEN);
        g.fillOval(left - eyeRadius, eyeY - eyeRadius,
                eyeRadius * 2, eyeRadius * 2);
        g.fillOval(right - eyeRadius, eyeY - eyeRadius,
                eyeRadius * 2, eyeRadius * 2);

        // draw the mouth
        int mouthRadius = width / 10 + mouthDelta;
        g.setColor(Color.BLACK);
        g.fillOval(width / 2 - mouthRadius - mouthWidthDelta,  // x
                mouthY - mouthRadius, // y
                (mouthRadius + mouthWidthDelta) * 2,  // width
                mouthRadius * 2);  // height

        // draw the hat last (on top)
        if (hat) {
            g.setColor(Color.BLACK);
            int midX = width / 2;
            int wide = 3*width/4;
            g.fillOval(midX - wide / 2, 0, wide, height / 15);
        }
    }
}
```

How Does a GUI Work?

- Objects in memory, storing state as strings, ints pointers, ...

- The System sends these objects `paintComponent()` and they draw themselves on screen

- The user clicks, types, ... the system maps theses events to notification messages sent to the objects. The objects react, changing their state, and ultimately drawing that new state on screen.

- In this way, it appears to the user that their actions changed what's on screen.
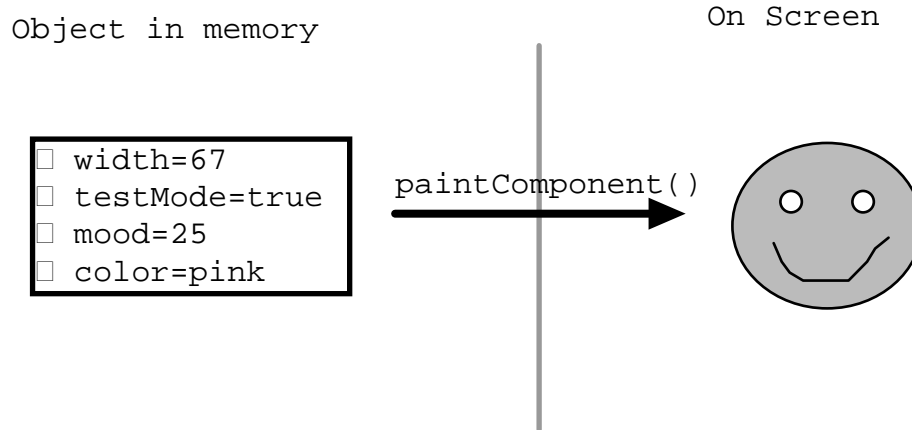
paintComponent() -- System Driven

- Components wait for system to tell them to draw, what size, etc.

- Do not just start drawing on your own, they way you would in a C program

paintComponent():
Object model -> pixels

- Object in memory -- has assorted ivars: int, boolean, String ... which collectively define its current state -- the "model" data.

- What `paintComponent()` does: looks at the model data and draw the pixels that represent that state. Does not change the model/state of the object.

- `paintComponent()` should treat the object as read-only -- just look at its state and render out pixels.

```
      Object in memory                    On Screen


        width=67
        testMode=true       paintComponent()
        mood=25
        color=pink
```

Repaint -- Request a Redraw

90% of drawing is automatic

- 90% of the time, drawing is initiated automatically.

- The programmers does not need to do anything at all -- the system notes these cases automatically and does the drawing...
    - Expose event-- a component used to be covered in the "z-order" stacking of components, but now is not and needs to be redrawn. Sometimes called a "damage" area on screen -- needs to be redrawn.
    - Resizing
    - Scrolling

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

Redraw request: component.repaint();

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

Asynchronous repaint()

- `component.repaint()` **does not do the drawing immediately**.

- The connection from `repaint()` to `paintComponent()` is indirect.

- `component.repaint()` adds a notation in the event queue that the component needs to be drawn in the near future.
- When the swing thread gets to the repaint request, it will, with some intermediate logic, send `paintComponent()` to that component.

Do Not call paintComponent()

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

"Up To Date" Repaint Model

- You can think of keeping the object state and its pixels on screen "in sync" -- redrawing the pixels when the state changes.
- Object Model
    - Each component stores model data: strings, pointers, booleans...
    - Some of the model affects the way the object appears on screen.
- Pixels
    - The pixels on screen are a function of the model
- Out of date
    - A change to the model makes the on-screen pixels **out of date** -- they are the pixels from an earlier `paintComponent()` with the old model state.
- State Change -> Repaint
    - When the object model changes, do a `repaint()` to trigger a `paintComponent()` using the new model.

Common repaint() Errors

- What happens if you forget to call `repaint()` .. the pixels show the old state. When you update the pixels for some other reason -- say resizing the window -- suddenly the right pixels appear.
- What happens if you call `repaint()` from `paintComponent()`? This is an error, but more subtle. It puts the swing thread in a sort of infinite loop, bogged down with constant useless drawing. Event handling will be sluggish. `paintComponent()` should just look at the current model and render out pixels. Avoid making it more complex since it is called in so many contexts.

Setter Repaint Pattern

- Since calls to `repaint()` tend to go with changes in the object's model data, it's natural to put them in the object's setter methods that change the model.

Angry Example

- There is an `angry` boolean ivar in the face -- when it is true, the face draws in red.
- `paintComponent()` looks at the value of `angry`, and draws in the smiley face in red if it is `true`. Otherwise, the color will be its default -- black.

```
// smiley -- draws in red if angry
public void paintComponent(Graphics g) {
    if (angry) {
        g.setColor(Color.RED);
        ...
```

- The `angry` boolean is controlled by a `setAngry()` setter. The `setAngry()` does a `repaint()` since the `angry` state is relevant to the appearance -- classic use of `repaint()` in a setter to trigger the redraw.

```
public void setAngry(boolean angry) {
    this.angry = angry;   // change object state
    repaint();            // then repaint to trigger a re-draw
```

```
    }
```

- Or to be a little slicker, we could detect if the new `angry` value is different from the old. A redraw is only required if it's really different.

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

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.

- Fish With hat
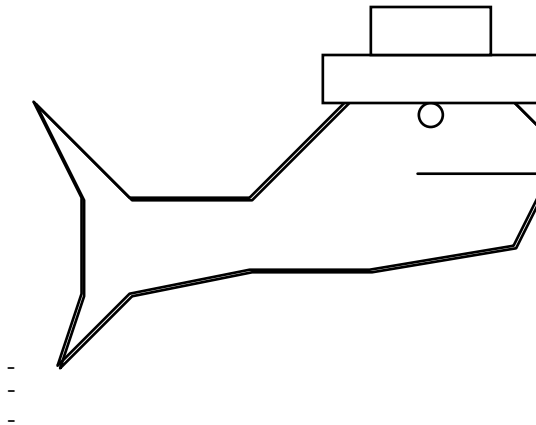
- Fish Without hat (the hat has been "erased")
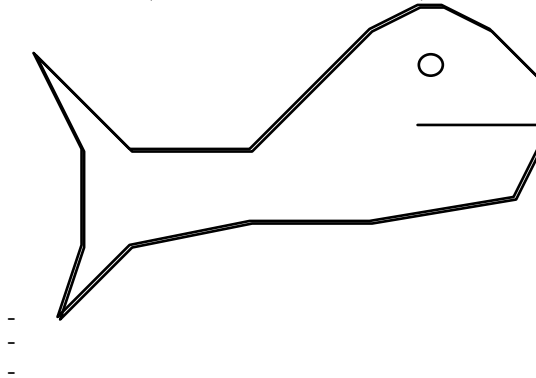
- Fish class...

```
- 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.

Easy for the Client

- By putting the logic in the setter, we make it easy for the client, putting the work on the implementation -- good OOP style.
  - Some state is relevant to the appearance and some is not, but the client should not need to know those details.
  - Do not make the client figure this out -- just hide the call to `repaint()` in the appropriate setters.
  - This is another example of the sort of asymmetry you tend to see in good client oriented design -- simple for the client, complex for the implementation.

Repaint-Face Example


```java
/*
 Classic setter -- changes state and sends repaint()
 to trigger a redraw.
*/
public void setHat(boolean hat) {
    this.hat = hat;
    repaint();
}

public boolean getHat() {
    return hat;
}


public void setAngry(boolean angry) {
    // Could add slight optimization to only repaint if it's
    // *really* a change.
    if (this.angry != angry) {
        this.angry = angry;
        repaint();
    }
}

public void setEyeDelta(int delta) {
```

```java
        eyeDelta = delta;
        repaint();
    }

    public void setMouthDelta(int delta) {
        mouthDelta = delta;
        repaint();
    }

    public void setMouthWidthDelta(int delta) {
        mouthWidthDelta = delta;
        repaint();
    }

    /*
     Creates a frame with a Face, wire its setters to some controls.
     */
    public static void main(String[] args) {
        JFrame frame = new JFrame("Face");

        Box box = Box.createVerticalBox();
        frame.add(box);
        box.setBackground(Color.lightGray);

        // Add a Face
        final Face face = new Face(100, 100);
        box.add(face);

        // Angry Checkbox
        final JCheckBox angerManagement = new JCheckBox("Angry");
        box.add(angerManagement);
        angerManagement.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                face.setAngry(angerManagement.isSelected());
            }
        });

        // Hat CheckBox
        final JCheckBox hat = new JCheckBox("Hat");
        box.add(hat);
        hat.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                face.setHat(hat.isSelected());
            }
        });

        // Eye slider
        final JSlider eyeSlider = new JSlider(-100, 100, 0); // (min, max, current)
        box.add(eyeSlider);

        // JSlider usese the "ChangeListener" listener interface
        eyeSlider.addChangeListener(new ChangeListener() {
            public void stateChanged(ChangeEvent e) {
                face.setEyeDelta(eyeSlider.getValue());
            }
        });

        // Mouth slider
        final JSlider mouthSlider = new JSlider(-100, 100, 0);
        box.add(mouthSlider);

        mouthSlider.addChangeListener(new ChangeListener() {
            public void stateChanged(ChangeEvent e) {
                face.setMouthDelta(mouthSlider.getValue());
            }
        });

        // Width slider
        final JSlider widthSlider = new JSlider(-100, 100, 0);
        box.add(widthSlider);

        widthSlider.addChangeListener(new ChangeListener() {
            public void stateChanged(ChangeEvent e) {
                // note: can access final stack var "widthSlider"
```

```
                    face.setMouthWidthDelta(widthSlider.getValue());
            }
        });

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
    }
}
```