Repaint

How Does a GUI Work?

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

The System sends these objects paintComponent() and they drawthemslves 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 draw that new state on screen.

In this way, it appears to the user that their actions changed what's on screen.

paintComponent() -- System Driven

Wait for system to tell you to draw, what size, etc.

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

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.

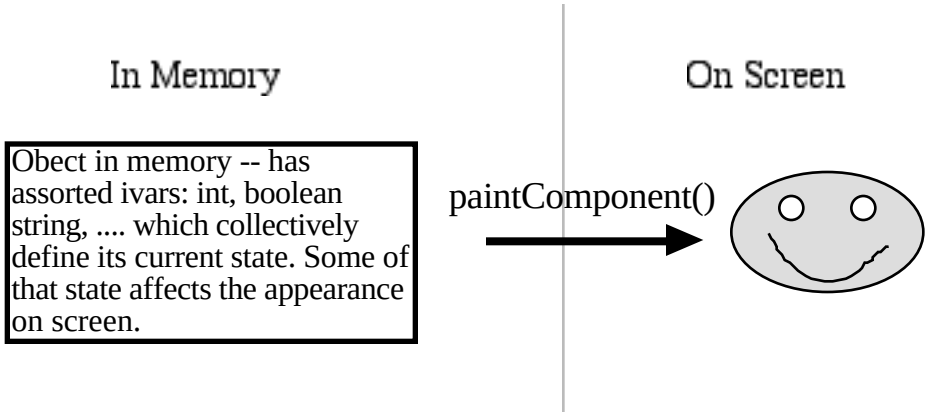
Put a Toolkit.getDefaultToolkit().beep() to see if paintComponent() is getting called at all

While a Swing program is running, type ctrl-shift-f1 to get a debugging printout dump of the installed components

paintComponent()

Object state -> pixels, read only

What paintComponent() does: look a the state of the object, and draw the pixels that represent that state. Do not change the state of the object.



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

Asynchronous

component.repaint() **does not do the drawing immediately**.
The connection from repaint() to paintComponent() is indirect.
component.repaint() causes the system to note in its event queue that given component needs a redraw.
A fraction of a second later, the system draw thread will dequeue the draw request and ultimately send paintComponent() to the component.

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

You can think of keeping the object state and its pixels on screen "in sync" -- redrawing the pixels when the state changes.

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.

Pixels

The pixels on screen are a function of the object state

Out of date

A change to the object state makes the on-screen pixels **out of date** -- they are the pixels from an earlier paintComponent() with the old object state.

State Change -> Repaint

When the object state changes, do a repaint() to trigger a paintComponent() using the new object state.

Setter Repaint Pattern

Since repaints() tend to go with changes in the objects state, it's natural to put them in the object's setter methods.

Face Repaint Example

Suppose we have a Face component that draws itself as a smiley face by default. There is an "angry" boolean ivar -- when it is true, the face draws in red.

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

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

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

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;
   repaint();
}
```

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.

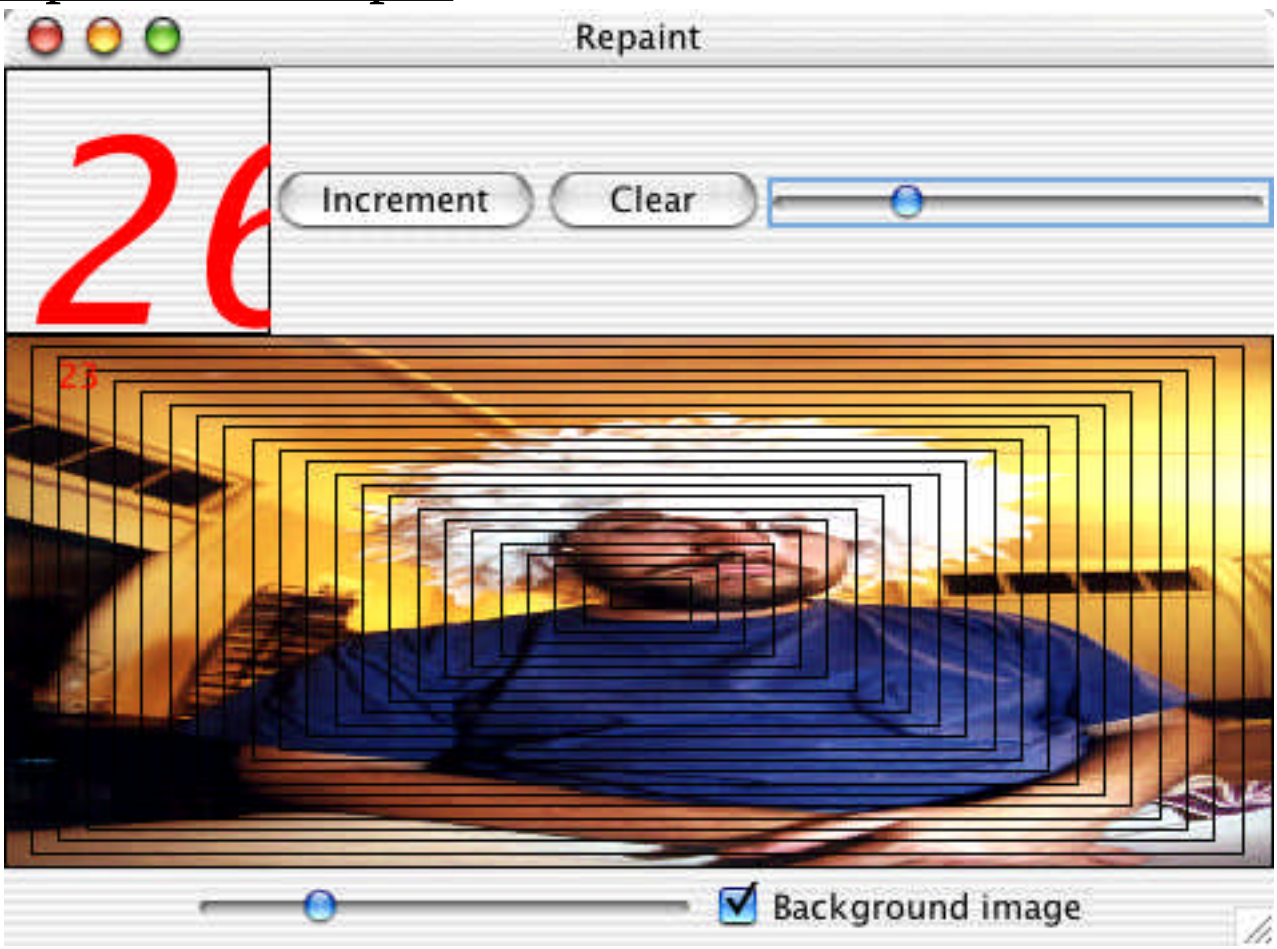
```
public void setAngry(boolean angry)
{
   if (this.angry != angry) {
      this.angry = angry;
      repaint();
   }
}
```

Work for Client NO / Work for receiver implementation YES
- 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 Bugs

It can be tempting to sprinkle repaint() calls all around -- resist the temptation. Figure out the one or two places where repaint() is really needed.
With needless repaint() calls, the app will still appear to function, but it will be slower than necessary.
What if the paintComponent() calls repaint()?

Repaint Example



/ / Widget.java

```
// Widget.java
/*
 A component that stores a number and draws it
 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
       // note: it's a static
      if (font==null) font = new Font("DIALOG", Font.ITALIC, 96);
```

```
      g.setFont(font);
      g.setColor(Color.red);
      // draw near the bottom
      g.drawString(Integer.toString(count), 4, getHeight()-4);
   }
```

/ / Boxer.java

```
// Boxer.java
/*
 Simple component that stores a count, and draws that number
 of boxes within its bounds.
 Also can store an Image object -- if non-null, draws the image
 behind the boxes.
 Demonstrates setter/repaint style.
*/
import java.awt.*;
import javax.swing.*;

class Boxer extends JComponent {
   private int count;   // number of boxes to draw
    private Image image;    // image to draw (may be null)

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

      count = 1;
      image = null;
   }


   /*
    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();
      }
   }


   /*
    Installs an image for us to draw, or null
    to not draw an image.
   */
```

```
   public void setImage(Image image) {
       this.image = image;
       repaint();
   }

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

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

      // If the image is present, draw it first (behind everything)
      if (image != null) {
          // drawImage() will scale the image to whatever size we say
          g.drawImage(image, 0, 0, width, height, this);
      }

      // Draw the series of rectangles
      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

```
// Repaint.java
/*
 Creates 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;
   private Image image;
```

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


   // Put in a border layout
   JComponent content = (JComponent) getContentPane();
   content.setLayout(new BorderLayout());


   // Make a north horizontal box layout
   JPanel north = new JPanel();
   north.setLayout(new BoxLayout(north, BoxLayout.X_AXIS));
   content.add(north, BorderLayout.NORTH);

     // Put in a widget and some controls for it
      widget = new Widget(100, 100);
   north.add(widget);

   JButton button;

   button = new JButton("Increment");
   north.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
   north.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)
   north.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 controls in a south panel
```

```
   final Boxer boxer = new Boxer(200,200);
   content.add(boxer, BorderLayout.CENTER);

   JPanel south = new JPanel();
   content.add(south, BorderLayout.SOUTH);

   JSlider slider2 = new JSlider(0, 100, 0);
   south.add(slider2);

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

   final JCheckBox imageMode = new JCheckBox("Background image");
   south.add(imageMode);
   imageMode.addActionListener(
      new ActionListener() {
         public void actionPerformed(ActionEvent e) {
             if (imageMode.isSelected()) {
                 // load image if not loaded already
                 if (image==null) image = loadImage("jeffsad.jpg");

                 boxer.setImage(image);
             }
             else boxer.setImage(null);
         }
      }
   );

   setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   pack();
   setVisible(true);

}


/*
 This is the standard code to load an image --
 it's complicated because getImage() returns
 before the image is actually loaded. The ImageObserver
 stuff is required to wait for the image to
 actually load. Bad design: making the obvious case
 hard on the client to support some advanced feature.

 In Java1.4, just use ImageIO.readImage(new File(filename))
*/
public Image loadImage(String filename) {
   // Give us an Image object pointing to the given file
   // (does not load the image synchronously)
   Image image = Toolkit.getDefaultToolkit().getImage(filename);

   // This incantation causes the loading of the image
```

```
      // to actually happen -- we block while it does.
      // If the image is not available, this just
      // falls through silently.
      MediaTracker tracker = new MediaTracker(this);
      tracker.addImage(image, 0);
      try{
         tracker.waitForID(0);
      }
      catch (InterruptedException e) {
         e.printStackTrace();
      }

      return(image);
   }
```

Drawing Misc.

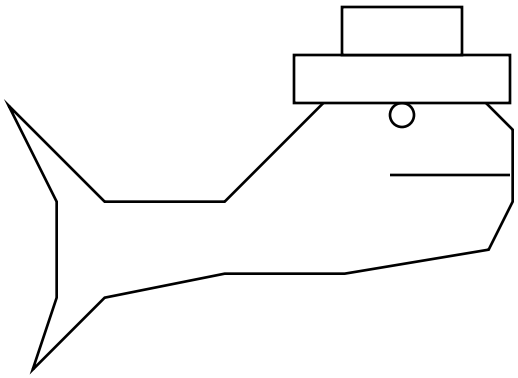
Erasing

We don't actively erase things.

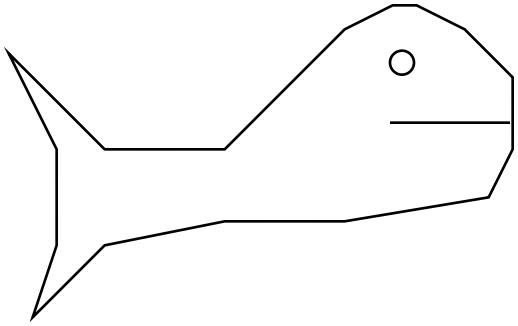
To "erase" something, we just don't draw it in paintComponent(), and so it disappears.

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.

Boxer Example -- the Boxer draws the image if the image ivar is not null. To erase the image, set the image ivar to null and repaint. This triggers a paintComponent() which, since the image ivar is null, doesn't draw the image, and so it disappears.

Smart Repaint

Smart repaint = repaint just the rectangle of the component that needs to be redrawn, not the entire component or window bounds.

This makes the following draw cycle faster, so we get faster, smoother drawing. This can be quite a speedup, if the smart repaint rectangle is significantly smaller.

There's a version of the repaint() method that takes a rectangle argument, and just repaints that rectangle (rather than the component bounds) -- component.repaint(x, y, width, height)

e.g. -- just repaint the old+new rectangles when a component moves.

The system gets this right automatically when moving components around with, say, a JPanel. See the setBounds() source code -- repaints just the old+new regions.