Swing 1

Introduction

Our first topic will be advanced use of MVC in Swing. We'll review basic Swing very quickly, so we can get to the advanced parts.

If you are not familiar with Swing, see the Swing track in the Java tutorial...
http://java.sun.com/docs/books/tutorial/uiswing/index.html

OOP/GUI Systems

CT hierarchy of library classes

RT system -- event dequeue

User event maps to Java message "notification" sent to the object
e.g. the "draw yourself " message paintComponent()

"Passive" style

Objects sit around waiting for the system to tell them something. They react to each notification: click, draw, etc.

Java GUIs

The AWT/Peer Debacle

"Peer" system

Each Java AWT object, such as Button or Panel, on the Java side has a "peer" written in C on the VM side. The AWT object and the peer collectively provide the implementation. Each VM has its own peer code particular to that vendor/OS.

Bug/Quirk Matching

Develop a client against the Microsoft AWT implementation. The client code develops quirks that match the bugs/quirks in the Microsoft AWT. The code then fails to work against, say, the Sun Solaris VM with its quirks.

Flaw

Many separate implementations trying to implement from a common set of docs **does not work** if the spec is complex. If the implementations are to be truly compatible with each other, they must be built from common source -- so called "bug for bug" compatible.

The new Swing/JFC approach...

"Lightweight" Swing objects -- no native peers

- Swing objects written entirely in Java
  - It's the **same** Java bytecode on Windows, Solaris, ...
- Layout Manager Portability
- Pluggable Look 'n' Feel
- Listener Event Model
- Serialized-GUI-Bean technology (Java 1.4)

JComponent

- Drawable
  - The superclass of all drawable, on screen things
- 227 public methods
  - Go read through the method documentation page for JComponent once (off the home page)
- Its Abstraction
  - How the geometry works
  - How components relate to each other
  - When what happens
- 1. Geometry Rect (location + size)
- 2. Containment
- 3. Drawing
- 4. Event handling
- Class Hierarchy
  - **JComponent** has two superclasses that are AWT classes -- (AWT) **Component**: (AWT) **Container**: JComponent
  - There are few times the AWT classes, intrude, but mostly we'll try to conceptually collapse everything down to JComponent.

Geometry Theory

- Size + Loc
  - Own co-ord sys w / origin (0,0) in the Upper Left
  - Extends out to getWidth() and getHeight()
- Local Co-ord System
  - "local coordinate system" -- remains constant, even as you get moved around.
  - You can always draw relative to your own local coordinate system.
- PreferredSize
  - The layout manager determines our exact size. Use setPreferredSize() to indicate your wishes to the layout manager.
- Parent = our container
- Layout Manager
  - Looks at the preferred size of everything, the size of the window, etc. and arranges (size+loc) of everything as best it can.
- setSize() **no**, setPreferredSize() **yes**

It is rarely the case that the size of component is set by client code that calls setSize().

Send getWidth, getHeight(), getSize(), getLocation(), getBounds()

To see where you are and draw within that (0,0) out to getWidth() getHeight()

You do not get to dictate your geometry -- the LayoutManager does

Geometry Methods

(Mostly inherited from Component)

Constructor

The initial component is size0 and has no parent

int getWidth(), getHeight()

Dimension getSize([Dimension]);

Like above, but get width / height in an object

int getX(), getY()

Get the location of the upper left of our coord system within our container.

Location getLocation([Point])

As above, but in an object

get / set PreferredSize(Dimension)

Rectangle getBounds([Rectangle])

location and size all at once

boolean contains(x,y)

boolean contains(Point)

setBounds(Rectangle -or- x,y,w,h)

Do not call this -- the layout manager is responsible for establishing the bounds

Likewise, do not call setSize()

getParent()

How To Draw

paintComponent(Graphics g)

Sent to the object when it should draw itself

Override to provide code for a component to draw itself

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

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

Passive

Note: passive -- you don't demand to draw, you respond -- drawing when the system says to draw

Best design: all drawing bottlenecks through paintComponent()

```
public void paintComponent(Graphics g) {
   // super.paintComponent(g);   // not necessary for simple cases

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

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

Graphics

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

(0,0)

In the upper left hand corner
X extends to the right
Y extends down

g.drawRect(x, y, width, height)

Draw rect 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 one

g.fillRect(x, y, w, h)

Uses the current color, does not overhang like drawRect()

drawLine(x1, y1, x2, y2)

drawString(String, x, y)

g.setColor(Color)

JFrame

Window

Represents one window

container = frame.getContentPane()

Gets the "content" part of the window where things go

container.setLayoutManager(...)

Install a layout manger which will arrange the components in the container.

container.add()

Add a component to the content.

frame.pack()

Size the frame depending, recursively, on the sizes of the things it contains.

frame.setVisible(true)

Initially the frame is not visible -- setVisible(true) brings it on screen

Layout Manager Theory

Like HTML -- policy, not exact pixels

1. Don't set explicit (pixel) sizes for things
2. The layout managers knows the "intent" (policy) of the layout

e.g. vertical list

3. The layout manager applies the intent to figure the correct size on the fly

Pro: the GUI can work, even though different platforms have fonts with slightly different metrics

Pro: re-sizing works (the policy

Pro: internationalization

Con: new paradigm, can be unwieldy, get used to it

Flow Layout

Arranges components left-right, top-down like text.

Box Layout

Can create a horizontal or vertical box

Border Layout

Main content in the center

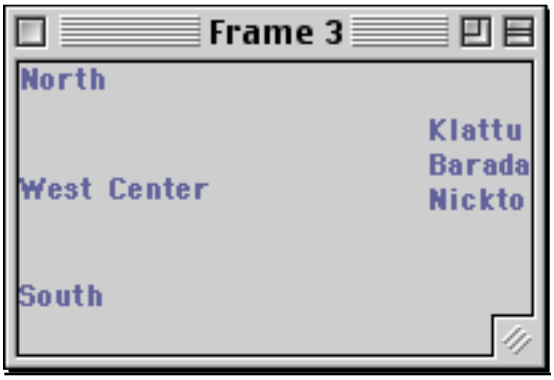
e.g. the spreadsheet cells

Decorate with 4 things around the outsize -- north, south, east, west

e.g. the controls around the spreadsheet cells

Resizing changes the size of the center

FirstFrame Example




```
// FirstFrame.java
/*
 Demonstrates bringing up a frame with the various
 layout managers.
*/
import java.awt.*;
import javax.swing.*;
import java.util.*;
import java.awt.event.*;

public class FirstFrame extends Object {
```

```
public static void main(String[] args) {
   // 1. Frame with flow layout
   JFrame frame = new JFrame("Frame 1");
   JComponent container = (JComponent) frame.getContentPane();
   container.setLayout(new FlowLayout());

   container.add(new JLabel("Hello World"));
   container.add(new JLabel("another label"));

   frame.pack();
   frame.setVisible(true);

   // By default, the frame just hides -- the following
   // actually deletes it on close...
   // frame.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);


   // -----
   // 2. Frame with box layout

   JFrame frame2 = new JFrame("Frame 2");
   JComponent container2 = (JComponent)frame2.getContentPane();

   // The Box layout make a vertical arrangement
   container2.setLayout(new BoxLayout(container2, BoxLayout.Y_AXIS));

   // add a few components
   container2.add(new JLabel("Homer"));
   container2.add(new JLabel("Marge"));
   container2.add(new JLabel("Lisa"));
   container2.add(new JLabel("Bart"));
   container2.add(new JLabel("Maggie"));

   frame2.pack();
   frame2.setVisible(true);


   // ----
   // 3. Frame with border layout
   JFrame frame3 = new JFrame("Frame 3");
   JComponent container3 = (JComponent)frame3.getContentPane();

   // Border layout
   // (the 6's are for inter-component spacing)
   container3.setLayout(new BorderLayout(6, 6));

   container3.add(new JLabel("North"), BorderLayout.NORTH);
   container3.add(new JLabel("West"), BorderLayout.WEST);
   container3.add(new JLabel("Center"), BorderLayout.CENTER);
   container3.add(new JLabel("South"), BorderLayout.SOUTH);

   // Create a little panel (box layout)
   JPanel panel = new JPanel();
   panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
   panel.add(new JLabel("Klattu"));
   panel.add(new JLabel("Barada"));
   panel.add(new JLabel("Nickto"));
```

```
      // Put it in the east
      container3.add(panel, BorderLayout.EAST);

      frame3.pack();
      frame3.setVisible(true);
   }
}
```

JComponent

(Details on the previous handout)

Drawable
Geometry
Containment
System arranges and tells to draw

Key Methods

Send to self:
- setPreferredSize() -- inform the layout manager of your pref
- getWidth()
- getHeight()

Override
- paintComponent() -- all drawing happens here

Graphics
- setColor()
- fillRect(x, y, width, height)
- drawRect(x, y, width*, height*) -- draws one pixel too large in width and height
- drawString(str, x, y)

Key Points

1. Layout manager arranges geometry
   - Use setPreferredSize() to influence the layout manager
2. Subclass off JComponent
3. Override paintComponent()
4. Use the passed in Graphics object

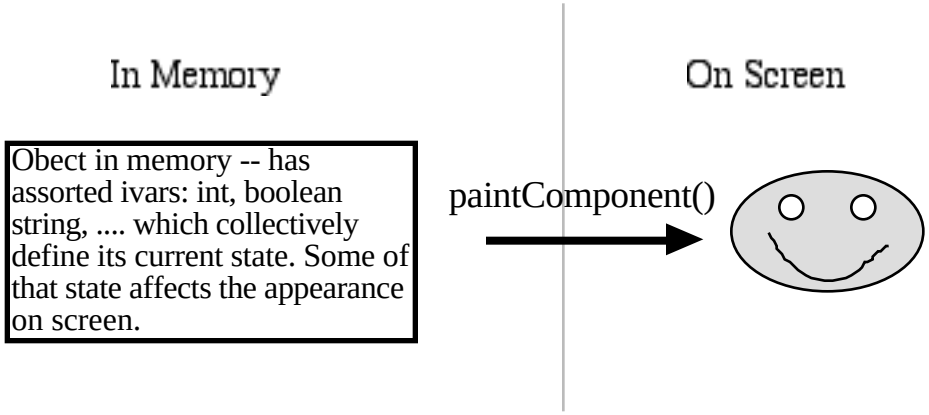
paintComponent() Features...

1.Passive

Wait for system to tell you to draw
drawRect() debug -- put a call to g.drawRect() at the start of your paintComponent() just too where things are.

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" 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. getGraphics -- NO

- Almost always incorrect to use this
- Only use if the assignment handout specifically says to
- Subverts 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, but now its not
  - Resizing
  - Scrolling

component.repaint() -- draw request

- Tell the system that the given component needs to be redrawn

Asynchronous

- Does not draw immediately
- Marks the given component for future redraw (pretty soon -- fraction of a second)
- The system calls paintComponent() on its schedule

"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
  - The change has made the on-screen representation out of date -- it is showing the result of 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 angry and draws accordingly

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

      if (angry) g.setColor(Color.red);
      g.drawRect(0, 0, getWidth()-1, getHeight()-1);
   }
```

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

```
{
   this.angry = angry;
   repaint();
}
```

- Better

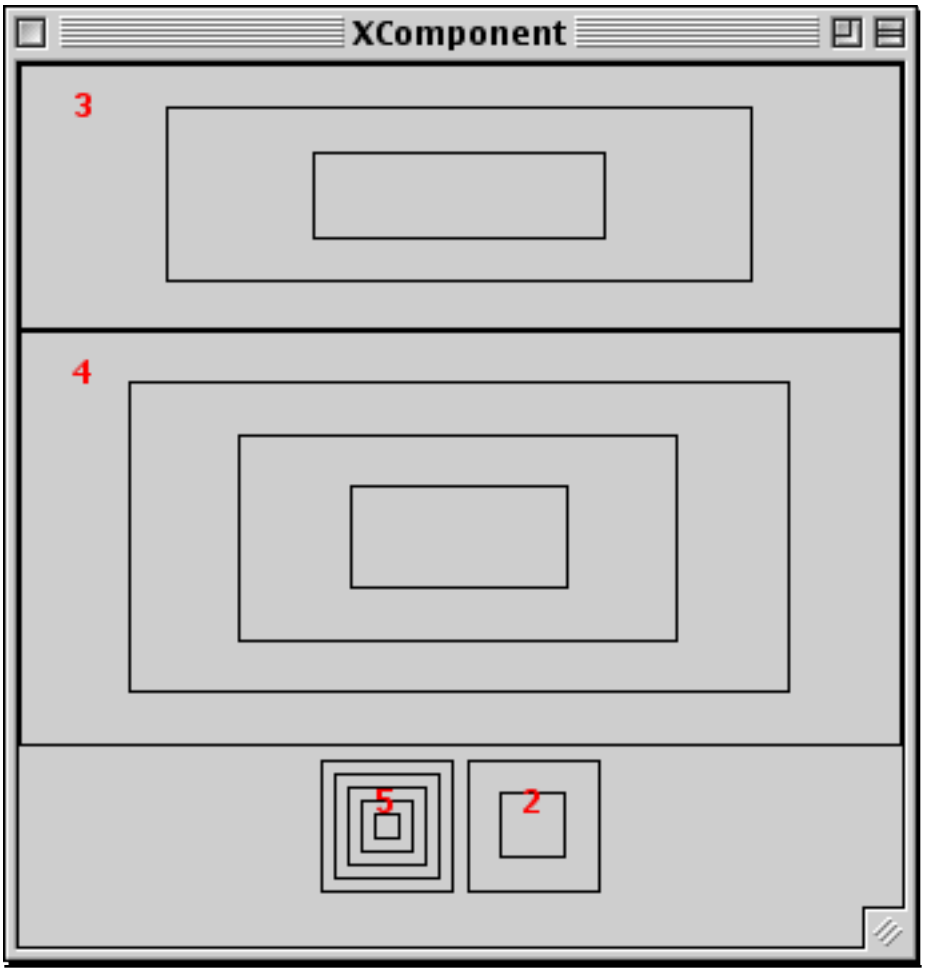
```
{
   if (this.angry != angry) {
      this.angry = angry;
      repaint();
   }
}
```

- Work for Client NO /
- Work for Utility 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 around -- don't be careless
- What if the code calls repaint() in paintComponent()?

XComponent Example



```
// XComponent.java
/*
 Simple class that demonstrates basic drawing,
 using repaint, and layout managers.
*/
import java.awt.*;
import javax.swing.*;
import java.util.*;

import java.awt.event.*;


import com.sun.java.util.collections.*;


class XComponent extends JComponent {
   private final static int MAX = 10;
   private int count;   // ranges 1..MAX

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

      count = 1;

      // Create and register a mouse listener
      addMouseListener( new MyMouseListener());
   }


   // This inner class gets mouse event notifications for us
   class MyMouseListener extends MouseAdapter {
      public void mousePressed(MouseEvent e) {
         upCount();
            //System.out.println("pressed  x:" + e.getX() + " y:" +e.getY());
      }
   }

   /*
    Increase the count.
    Call repaint() to signal a redraw --
    classic example: state change -> repaint
   */
   public void upCount() {
      count++;
      if (count>MAX) count=1;

      repaint();
   }


   /*
    Set the count.
    Does not respect MAX.
   */
   public void setCount(int count) {
      this.count = count;
      repaint();
```

```
    }

    /*
     Draw 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++) {
          // 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);
    }

    public static void main(String[] args) {
       JFrame frame = new JFrame("XComponent");
       JComponent container = (JComponent) frame.getContentPane();
       container.setLayout(new BorderLayout());

       // Put an XComponents in the NORTH and CENTER
       container.add(new XComponent(100,100), BorderLayout.NORTH);
       container.add(new XComponent(200, 200), BorderLayout.CENTER);

       // Create a little panel (flow layout)
       JPanel panel = new JPanel();
       panel.add(new XComponent(50, 50));
       panel.add(new XComponent(50, 50));

       // put it in the SOUTH
       container.add(panel, BorderLayout.SOUTH);

       frame.pack();
       frame.setVisible(true);

    }
}
```