OOP/GUI Libraries

Common OOP application
GUI's have a lot of standard, repeated behavior, so they are a common and natural domain to apply OOP techniques.

Swing Goals

Off-The-Shelf Classes
Package up common objects -- buttons, windows, menus, tables, ... -- so they are available "off the shelf" for GUI programmers.

Portable
Once GUI code is written to Swing, it can be run without even a recompile on other Java platforms.

RT World

RT Objects
RT arrangement of objects with pointers to each other. The RT arrangement forms a hierarchy which is different from the CT hierarchy below.

Library Objects ("Off The Shelf" objects)
Custom Objects

View Hierarchy
In particular, there is RT "View Hierarchy" defined by the nesting of drawable things in other drawable things -- can be drawn as a tree with the window at the root and its contained objects as children. This hierarchy is different from the CT class hierarchy.
RT World Drawing

JFrame

JPanel

Smiley (2 instances of custom class)

RT View Hierarchy Drawing

The run-time "parent" nesting of components on screen.

JFrame

JPanel

Smiley

Smiley

CT World — Class Hierarchy

CT hierarchy

CT tree of many classes making massive use of inheritance to define behavior. The logical CT class hierarchy is different from the RT arrangement hierarchy above.

Classes built into the library

JComponent (basic drawable)
JFrame
JLabel
JTable -- an entire 2-d table of drawable things

Custom Subclasses

Subclass off the built ins to customize
Smiley -- custom subclass of JComponent

Class Hierarchy Drawing

The compile-time arrangement of classes -- note that this hierarchy is separate from the view hierarchy above.
1. Three Varieties of Classes

**Built-In Objects**
which you instantiate but do not subclass -- required little understanding

**Custom subclasses**
Subclass off a built in class to provide specialization -- e.g. Smiley
subclassed off JComponent above
Subclassing off a built in class is not a trivial operation -- requires some
understanding of the superclass.
Repeat: subclasses correctly requires some understanding of the superclass
-- the subclass must fit within the superclass design logic

**Behind the scenes "facilitator"**
objects such as the event dequeuer -- require little understanding -- they
send you messages or do other bookkeeping behind the scenes.

2. "Responds To"

**Passive**
Objects respond to messages sent by "the system"-- objects rarely initiate
actions.

**Foo() Override**
If your class wants to respond to the Foo() message -- it overrides Foo() to
define its behavior for that message.
3. Event -> Message Mapping

User Events
Realtime "user" events such as clicking get mapped to OOP messages sent by the background event dispatch objects.

`paint()`
notification to draw yourself

`actionPerformed()`
notification that a button has been clicked
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
Fore/Back color
Font
Z-order overlap
Double-buffering
Transparency
Autoscrolling
4. Event handling
drawing
mouse
keyboard

5. Other
Having a border drawn at the edge
Accessibility for handicapped
Tooltips

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
Location is the location or our origin expressed in the coord system of our parent.

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

**Layout Manager**

**Defer**

Let the Layout Manager figure out the size and position of a component.
The component should send itself getWidth() etc. to see where it is
The component should use setPreferredSize() to set a size suggestion for the layout manager before layout happens.

**BorderLayout**

5 regions -- north, south, east, west, and center. Center is the main, resizeable content area, and the others are decorations around it.

**FlowLayout**

Left-right top-down arrangement, like text.

**BoxLayout**

A linear aligned arrangement -- horizontal or vertical.
Painting

`paintComponent(Graphics g)`
Sent to the object when it should draw itself
Override to provide code for a component to draw itself
Call `getSize()` etc. to see the current geometry
Note: passive -- you don't demand to draw, you respond
Best design: all drawing bottlenecks through `paintComponent()`

Geometry Methods
(Almost inherited from Component)

Constructor
The initial component is size0 and has no parent

`Dimension getSize([Dimension]);`
Our height and width

`get/set PreferredSize(Firstname)`

`Location getLocation([Point])`

`Rectangle getBounds([Rectangle])`
location and size in parent co-ord system

`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

`getHeight(), getWidth()`

`getParent()`

Drawing

`get/set`

`Foreground/Background(Color)`

`get/set Font`
setSize(Dimension -or- w,h)
Do not call, layout manager is responsible for establishing the bounds
setEnabled(boolean)
setVisible(boolean)
paintComponent(Graphics)
Override to customize how this draws itself
paint() deals with the border etc. -- leave it alone

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

drawRect(x, y, width, height)
Extends past the given width and height by 1 on the right and bottom, so
you frequently subtract one

fillRect(x, y, w, h)
Uses the current color, does not overhang like drawRect()

drawLine(x1, y1, x2, y2)

drawString(String, x, y)

setForeground(Color)

Simple Component Example

import java.awt.*;
import javax.swing.*;
import java.util.*;
import java.awt.event.*;

import java.awt.event.*;
A very simple component subclass example.

```java
class MyComponent extends JComponent {
    MyComponent(int width, int height) {
        super();
        setPreferredSize(new Dimension(width, height));
    }

    // Draw a rectangle around the component
    // (ignoring insets for now)
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        Dimension size = getSize();
        g.drawRect(0, 0, size.width-1, size.height-1);
        // -1 since drawRect overhangs by one
        // could use getWidth()
        g.drawString("Binky", 20, 20);
    }

    public static void main(String[] args) {
        JFrame frame = new JFrame("Simple"); // makes a window
        // place to add components
        JComponent container = (JComponent) frame.getContentPane();
        container.setLayout(new FlowLayout());
        // add the components
        container.add(new MyComponent(50, 50));
        container.add(new MyComponent(40, 20));
        container.add(new MyComponent(20, 40));
        // lay everything out
        frame.pack();
        frame.setVisible(true);
        // frame.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
    }
}
```
The Repaint System

Automatic Repaint Cases

There are many cases where "the system" realizes a component needs to be redrawn and so does the repaint() itself. These probably account for 90% of the cases — manual calls to repaint() are somewhat rare and we'll deal with them later.

1. First Brought on Screen
2. "Exposed" event
   You were covered by something, but now that something is gone.
3. Resized
   The layout manager has resized/repositioned you, so you need to be redrawn.
4. Scrolled
   You were inside a scroller. It scrolled to change what part of you is exposed, so now that part needs to be redrawn. This is essentially a special case of the "exposed" event.

The Point

The system deal with a great many situations automatically. Most often, all you need to do is respond to paintComponent() -- the system is figuring the when and where for you.

The Repaint System

Who / When

The system needs to compute when to draw what components

Layering

This is complicated by the fact that the components overlap each other

Region Based Drawing

"Update Region"

Stores the 2-d region on screen of what needs to be redrawn

Draw Thread

The draw thread notices when the update region is non-empty
1. Computes the intersection of the update region with components
2. Messages those components to draw themselves
3. Clears the update region

Component wants to redraw
It does not call its paint() directly
It adds its rectangle to the update region with a call to repaint()
The paint() notification will be sent sometime soon

Get Used To It
Region based drawing is the only way to go, so just get used to it.
Get rid of those imperative neurons.

Coalescing
Efficient
Smart about combining multiple update requests to be serviced by one paint cycle

Overlap
Transparent Objects
Z-Order

How To Redraw
1. 90% Automatic
Most of the cases, the system just gets -- all you need to do is respond to paintComponent()
2. 10% Manual Refresh
Sometimes a component needs to be redrawn for reasons unknown to the system
Send a repaint() message to that component, and its rect will be added to the update region.
Do not do this casually -- only add repaint() in the few cases where it is necessary.

"Synchronization" 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.

In Memory

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

On Screen

- paintComponent()

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.

```java
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.

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

Better

```java
{  
    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.

Component.move example

What area?
What needs to be repainted if I move a component from one location to another?

Region Based
Redraw is expressed in terms of regions
The region where the component is now needs to be redrawn
The region where the component used to be also needs to be redrawn -- there may have been something underneath it, or we at least need to draw the background.

Swing Source code
Just for fun, look at Component.move() -- there's a lot of stuff, followed by this little snippet

```java
// Repaint the old area ...
parent.repaint(oldParentX, oldParentY, oldWidth, oldHeight);
// ... then the new (this area will be collapsed by
// the ScreenUpdater if they intersect).
repaint();
```

paintComponent() Bottleneck

Advantage
The repaint-driven / update-region system also has the advantage of bottlenecking all draw code through one place (paintComponent()). Yet another example of the never have two copies of anything rule.

e.g. Angry drawing
The code for drawing the smiley is all in paintComponent(), so it is always consistent with itself.
Sometimes it is called because of a setAngry()
Sometimes it is called because of an expose event
It all goes through the same place, so it's always consistent.

Widget Example

```java
import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;
import java.awt.event.*;
import java.util.*;
```
A simple subclass of JComponent that does a little drawing.
Can be given a string at construction time which it draws.

```java
class Widget extends JComponent {
    private String string;

    public Widget(String string) {
        super();
        this.string = string;

        // If the string is empty, just use a rectangular shape
        // otherwise try to be wide enough to contain the string's width
        // (a better approach would be to measure the string's width
        // in pixels with the actual font in use)
        if (string.length() == 0) setPreferredSize(new Dimension(60, 20));
        else setPreferredSize(new Dimension(string.length() * 10, 20));
    }

    public Widget() {
        this(""");
    }

    /*
    Simple paintComponent() example.
    Draw ourselves with our current size relative
    to our upper-left (0,0) origin. Draw the string
    vertically centered.
    */
    public void paintComponent(Graphics g) {
        super.paintComponent(g);

        g.setColor(Color.red);
        // Note: we draw relative to (0,0), and ask
        // our superclass what our size is
        g.drawRect(0, 0, getWidth()-1, getHeight()-1);

        // Draw the string 7 pixels below
        // the vertical center
        g.drawString(string, 4, getHeight()/2+7);

        // System.out.println("paint!");
    }
}
```

Layout Examples

/*
Demonstrate common layouts.
*/
public class Layout {
    public static void flow() {
        JFrame frame = new JFrame("Flow");
        Container container = frame.getContentPane();

        container.setLayout(new FlowLayout(FlowLayout.LEFT, 6, 6));
    }
}
// Flow the components left-right, line by line -- like text
// Rearranges the components, but does not resize or overlap them
// The "6" is the h and v inter-component spacing

container.add(new Widget("Hello"));
container.add(new Widget("There"));
container.add(new JLabel("Flow"));
container.add(new Widget());
container.add(new JLabel("Layout"));

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

public static void border() {
    JFrame frame = new JFrame("Border");
    Container container = frame.getContentPane();

    container.setLayout(new BorderLayout(6, 6));
    // The NORTH SOUTH EAST WEST CENTER layout
    // (BorderLayout is actually the default)
    // Resizes components as it resizes
    container.add(new JLabel("East"), BorderLayout.EAST);
    container.add(new Widget("West"), BorderLayout.WEST);
    container.add(new Widget("CENTER"), BorderLayout.CENTER);
    container.add(new Widget("North"), BorderLayout.NORTH);
    container.add(new JLabel("South"), BorderLayout.SOUTH);

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

public static void box() {
    JFrame frame = new JFrame("Box");
    Container container = frame.getContentPane();

    // A convenience method in Box creates the container in one step
    Box box = Box.createVerticalBox();
    container.add(box);

    box.add(new JLabel("Box"));
    box.add(new JLabel("Layout"));
    box.add(Box.createVerticalStrut(10)); // create a little spacer
    box.add(new Widget("Homer"));
    box.add(new Widget("Bart"));
    box.add(new Widget("Lisa"));

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

public static void combo() {
    JFrame frame = new JFrame("Combo");
    Container container = frame.getContentPane();

    container.setLayout(new BorderLayout(6, 6));
}
// Use panels to make groups with vertical axis
// Then put the panels in the WEST and CENTER
JPanel panel1 = new JPanel();
panel1.setLayout(new BoxLayout(panel1, BoxLayout.Y_AXIS));
panel1.add(new JLabel("Box * 2"));
panel1.add(new JLabel("Layout"));
panel1.add(new Widget());

JPanel panel2 = new JPanel();
panel2.setLayout(new BoxLayout(panel2, BoxLayout.Y_AXIS));
panel2.add(new JButton("Compile"));
panel2.add(Box.createRigidArea(new Dimension(0,10)));
panel2.add(new JButton("Run"));
panel2.add(Box.createRigidArea(new Dimension(0,10)));
panel2.add(new JButton("Panic"));

container.add(panel1, BorderLayout.WEST);
container.add(panel2, BorderLayout.CENTER);
((JPanel)container).setBorder(BorderFactory.createEmptyBorder(10,10,10,10));
frame.pack();
frame.setVisible(true);