

Drawing

Q: How to get Java GUI on screen?

Q: How do you get some GUI components on screen?

A: Create a window (also known as a "frame") object. Install components in it (labels, buttons, ...). The system will manage the window and components, sending them notifications as user events happen (clicking, typing). The various components will draw themselves and handle events as they wish.

OOP GUI Systems

1. Classes

Hierarchy of classes for common problems -- drawing, controls, windows, scrolling, ...

2. System: Event -> Notifications

There is a background system that manages the basic bookkeeping and orchestration of windows and events. AKA "the system"

"User events" -- clicking, typing, ...

The system manages a queue of user events as they happen (realtime), and dispatches them one at a time to objects as "notification" messages

3. Instantiate Library Classes

Many tasks are as simple as constructing and installing system classes -- windows, buttons, etc.

This is the pretty easy -- requires some reading of the library class docs

Pull a library object "off the shelf"

4. Subclass Library Classes

To introduce custom behavior, subclass off a library class and use overriding to insert custom code

This is a trickier programming problem -- you need some understanding of the superclass in order to do the override "in harmony" with the superclass.

e.g. Subclass off button so it beeps when clicked -- keep the standard button code for drawing, mouse tracking, etc., but just add this one variation.

Java Swing GUI

AWT vs. Swing/JFC

AWT

Lame, broken, first iteration GUI Objects

AWT drawing uses "native peers" -- creating an AWT button creates a native peer (Unix, Mac, Win32) button to put on screen, and then tries to keep the AWT button and the peer in sync. This proved to be a bad strategy!

Swing

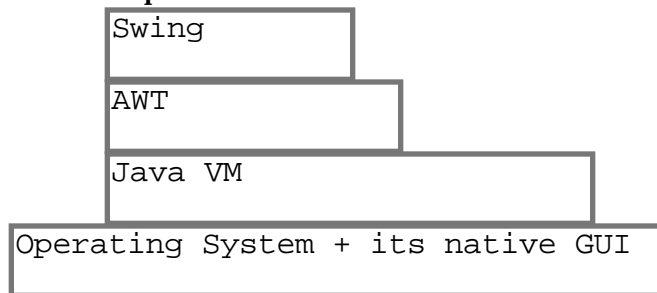
Also known as JFC

Implemented in Java -- the swingall.jar is the java code for swing -- it is the **same** swingall.jar running on all platforms.

Built on the AWT primitives, but done right

10x more classes, depth, and functionality than AWT

Has pluggable look-and-feel feature where buttons, etc can look like the ones for that platform.



AWT vs. Swing classes

Some old AWT classes are still used, but mostly we will use the modern Swing versions.

e.g. AWT Component is the superclass of JComponent

Theme: Things Draw Themselves

We will have objects that draw themselves -- labels, buttons, etc.

The system sends components "draw yourself" notifications as needed

Theme: Layout Manager

A "layout manager" will arrange the size and position of the things on screen.

For now, we'll ignore the layout manager

JComponent

The superclass of things that draw on screen.

Defines the basic notions of geometry and drawing -- details below

JLabel

Built in class that displays a little text string

```
new JLabel("Hello there");
```

JFrame

A single window

Has a "content pane" JComponent that contains all components in the frame

Send `frame.getContentPane()` to get the content pane

By default, closing a frame just hides it. See the code below so that closing a frame actually quits the application

Content Pane / Layout Manager

Use the `add()` message to add components to the content pane.

Content pane uses a "Layout Manager" to size and position its components

First Frame Example

A simple subclass of JFrame that puts 3 labels in its content pane.



FirstFrame Code

```
// FirstFrame.java
/*
 * Demonstrates bringing up a frame with some labels.
 */
import java.awt.*;
import javax.swing.*;
import java.util.*;
import java.awt.event.*;

public class FirstFrame extends JFrame {
    public FirstFrame(String title) {
        super(title); // superclass ctor takes frame title

        // Get content pane -- contents of the window
        JComponent content = (JComponent) getContentPane();

        // Set to use the "flow" layout
        // (controls the arrangement of the components in the content)
        content.setLayout(new FlowLayout());

        // Background color is a property of all components --
        // set it to white
        content.setBackground(Color.white);
    }
}
```

```

// Use add() to install components
content.add(new JLabel("Hello World."));
content.add(new JLabel("Another Label."));
content.add(new JLabel("Klaatu Barada Nikto!"));

// Force the frame to size/layout its components
pack();
setVisible(true);

// By default, the frame just hides when closed.
// The following causes program exit (with Java 1.2)
// frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);

// This is the old way to cause program exit on frame close
addWindowListener(
    new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
    }
);
}

public static void main(String[] args) {
    new FirstFrame("First Frame");
}
}

```

JComponent

JComponent Basics

Drawable

- The superclass of all drawable, on screen things
- Has a size and position on screen -- a "bounds" rectangle
- Draws itself, within its bounds

227 public methods

- Go read through the method documentation page for JComponent once (off the home page)

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.

Component Location/Size

Size + Loc

- Each JComponent has its own co-ord system with the origin (0,0) in the upper left corner

The "bounds" of the component is the rectangle with its upper left corner at (0,0) and extending out to component.getWidth() and component.getHeight(), with x growing to the right and y growing down

Local Co-ord System

The co-ord system of the component is not changed as the component moves around.

The component draws relative to its own local coordinate system with (0, 0) at the upper left

Parent container

The "parent" is the container that a component is in. The parent is itself a component.

The "location" of a component is the position of its upper-left corner in the co-ord system of the parent

PreferredSize

The layout manager determines the component size and location. Use setPreferredSize() to indicate your wishes to the layout manager. You can also set min and max sizes that the layout manager will try to respect.

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.

Send setSize() no, send 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()

Send these messages to determine the size and location of the component (essentially, its bounds rectangle): (0,0) out to getWidth() getHeight().

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

Geometry Methods

(Mostly inherited from Component)

Constructor

The initial component is size0 and has no parent

int getWidth(), getHeight()

Return the size of the component

Dimension getSize([Dimension]);

Like above, but get width/height in an object (potentially more slow)

int getX(), getY()

Get the location of the upper left of our co-ord system within our container (in the co-ord system of the container)

Location getLocation([Point])

As above, but in an object

get/set PreferredSize(Dimension)

Get or set the preferred size, which the layout manager uses when sizing and arranging components. The "Dimension" object encapsulates a width and height.

Rectangle getBounds([Rectangle])

Returns the current bounds in a Rectangle object

boolean contains(x,y), boolean contains(Point)

Test if the component bounds include the given point

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

You probably do not want to call this -- the layout manager is responsible for establishing the bounds

Likewise, do not call `setSize()`

`getParent()`

Get a pointer to the parent component

Drawing

OOP GUI Drawing Theory

Subclass off `JComponent`

Override `paintComponent()` -- draw within the bounds of the component

Install your components in a window -- they draw themselves

paintComponent(Graphics g)

Sent to a `JComponent` when it should draw itself

Override to provide custom drawing code

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

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

paintComponent Example

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

    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.

Passive Draw Style

Note: passive -- 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.

Graphics Object

A drawing context object 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)`

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)`

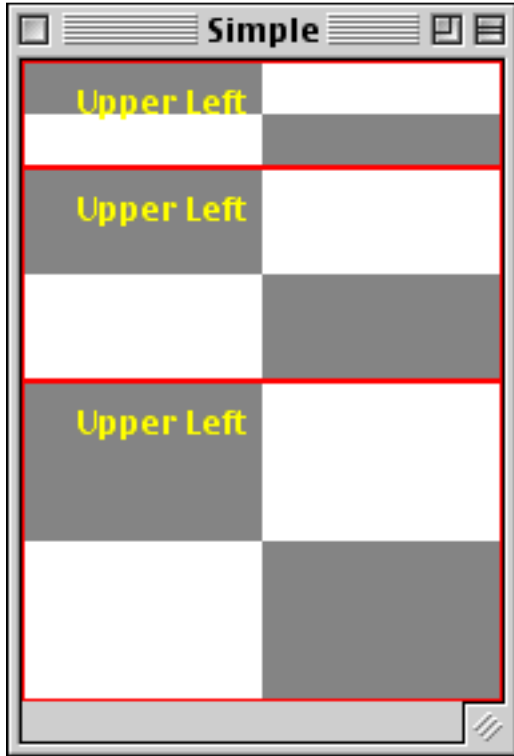
Sets the color for subsequent drawing.

There are constants in the Color class such as `Color.black`, `Color.green`, etc.

`Component.getGraphics()`

You probably never want to call this. Use the Graphics passed in to `paintComponent()`

MyComponent Example



```
// MyComponent.java
class MyComponent extends JComponent {
    MyComponent(int width, int height) {
```

```

    super(); // reminder that we have a super ctor

    // Set the preferred size -- used by the layout mgr
    setPreferredSize(new Dimension(width, height));
}

/**
Fills a gray rect in the upper left and lower right quarters.
Draws a string and a red frame at the bounds.

Typical paint component:
-see how big you are
-draw within your bounds
*/
public void paintComponent(Graphics g) {
    // super.paintComponent(g); // not necessary for simple cases

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

    // compute midpoint
    int midX = width/2;
    int midY = height/2;

    // draw two filled gray rects
    g.setColor(Color.gray);
    g.fillRect(0, 0, midX, midY);
    g.fillRect(midX, midY, width-midX, height-midY);

    // add a string at (20, 20) -- relative to our own origin
    g.setColor(Color.yellow);
    g.drawString("Upper Left", 20, 20);

    // draw a yellow rect frame at our bounds
    g.setColor(Color.red);
    g.drawRect(0, 0, width-1, height-1); // -1 for drawRect
}

public static void main(String[] args) {
    FirstFrame.main(null);

    JFrame frame = new JFrame("Simple");

    // Get the content area of the frame
    JComponent content = (JComponent) frame.getContentPane();
    content.setBackground(Color.white);

    // The Box layout makes a vertical arrangement.
    // Its components grow and shrink with the window
    content.setLayout(new BoxLayout(content, BoxLayout.Y_AXIS));

    // add a few components
    content.add(new MyComponent(180, 40));
    content.add(new MyComponent(140, 80));
    content.add(new MyComponent(120, 120));

    // This causes everything to get laid out

```



```

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

    // frame.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
}
}

```

Layout Managers

Layout Manager Theory

Like HTML -- policy, not exact pixels

1. Don't set explicit (pixel) sizes or positions 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: window re-sizing works (the layout manager policy guides how it fits components in to the new window size)
- Pro: internationalization -- layouts can adjust as the widths required for labels and buttons change for different languages
- Con: new paradigm, can be unwieldy when you just want to say where things are.

Future: an improved layout paradigm is coming in Java 1.4

Flow Layout

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

Box Layout

Aligns components in a line -- either vertically or horizontally

Can install a box layout into an existing JComponent

```
comp.setLayout(new BorderLayout(comp, BorderLayout.Y_AXIS));
```

Or, can create a "Box" component. There are convenience methods

`Box.createVerticalBox()` and `Box.createHorizontalBox()` that return a Box component. However, Box is not a JComponent, so the `setLayout()` technique on a JComponent above is preferable.

Use `Box.createVerticalStrut(pixels)` to create a little spacer component that be added to the box between components.

Border Layout

Main content in the center

e.g. the spreadsheet cells

Window size changes mostly go to the center

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

e.g. the controls around the spreadsheet cells

2nd parameter to `add()` controls where things go

```
border.add(comp, BorderLayout.CENTER); // add comp to center
```

Nested JPanel

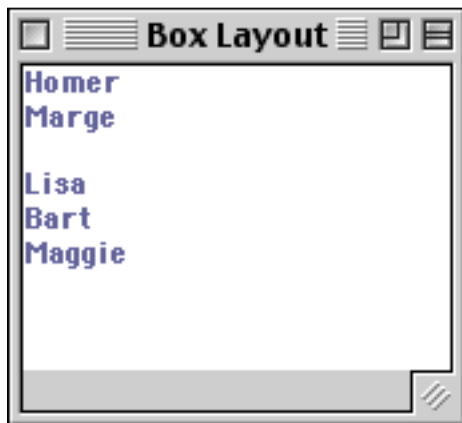
JPanel is a simple component that you can put other components in

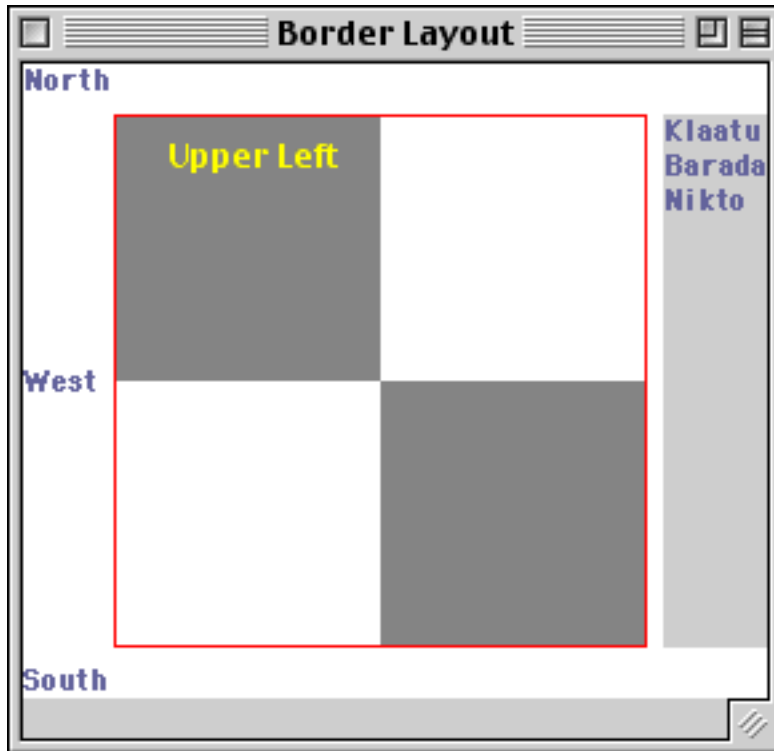
Use to group other components -- put them both in a JPanel, and put the JPanel where you want

e.g. group a label with a control

e.g. set the layout of the panel to vertical box, put lots of buttons in it, put the panel in the EAST of a border layout

Layout Example





```
// Layouts.java
/*
 * Demonstrates some basic layouts.
 */
import java.awt.*;
import javax.swing.*;
import java.util.*;
import java.awt.event.*;

public class Layouts {

    public static void main(String[] args) {
        // ----
        // 1. Flow Layout
        // Flow layout arranges Left-right top-bottom, like text
        JFrame frame1 = new JFrame("Flow Layout");
        JComponent content = (JComponent) frame1.getContentPane();
        content.setLayout(new FlowLayout());

        // Background color is a property of all components --
        // here I it to white, so it looks better in the handouts
        content.setBackground(Color.white);

        // Use add() to install components
        content.add(new JLabel("Hello World."));
        content.add(new JLabel("Another Label."));
        content.add(new JLabel("Klaatu Barada Nikto!"));

        // Force the frame to size/layout its components
        frame1.pack();
        frame1.setVisible(true);
    }
}
```

```

// ----
// 2. Box Layout
JFrame frame2 = new JFrame("Box Layout");
JComponent content2 = (JComponent)frame2.getContentPane();
content2.setBackground(Color.white);

// The Box layout make a vertical arrangement
content2.setLayout(new BorderLayout(content2, BorderLayout.Y_AXIS));

// add a few components
content2.add(new JLabel("Homer"));
content2.add(new JLabel("Marge"));

// add a little spacer
content2.add(Box.createVerticalStrut(12));

content2.add(new JLabel("Lisa"));
content2.add(new JLabel("Bart"));
content2.add(new JLabel("Maggie"));

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

// 3. Border Layout + nested box panel
JFrame frame3 = new JFrame("Border Layout");
JComponent content3 = (JComponent)frame3.getContentPane();
content3.setBackground(Color.white);

// Border layout
// (the 6's are for inter-component spacing)
content3.setLayout(new BorderLayout(6, 6));

// Add labels around the edge
content3.add(new JLabel("North"), BorderLayout.NORTH);
content3.add(new JLabel("West"), BorderLayout.WEST);
content3.add(new JLabel("South"), BorderLayout.SOUTH);

// Add a MyComponent in the center
content3.add(new MyComponent(200, 200), BorderLayout.CENTER);

// Create a little panel (box layout)
// with some labels. Nest it into the EAST
// (we'll use this strategy to arrange buttons
// around our main content)
JPanel panel = new JPanel();
panel.setLayout(new BorderLayout(panel, BorderLayout.Y_AXIS));
panel.add(new JLabel("Klaatu"));
panel.add(new JLabel("Barada"));
panel.add(new JLabel("Nikto"));

content3.add(panel, BorderLayout.EAST);

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

```