CSE 143 Java

Introduction to Graphical Interfaces in Java:
AWT and Swing

Reading: Sec. 19.1-19.3, 19.5

Overview

- Roadmap
  - Today: introduction to Java Windows and graphical output
  - Future: event-driven programs and user interaction
- Topics
  - A bit of history: AWT and Swing
  - Some basic Swing components: JFrame and JPanel
  - Java graphics
- Reading:
  - Textbook: Ch. 19
  - Online: Sun Java Swing tutorial (particularly good for picking up details of particular parts of Swing/AWT as needed); Swing API javadoc web pages
    http://java.sun.com/docs/books/tutorial/uiswing/index.html

Graphical User Interfaces

- GUIs are a hallmark of modern software
- Hardly existed outside research labs until Mac's came along
  - Picked up by PC's later
- User sees and interacts with “controls” or “components” (sometimes called “widgets”)
  - menus
  - scrollbars
  - text boxes
  - check boxes
  - buttons
  - radio button groups
  - graphics panels
  - etc. etc.

Opposing Styles of Interaction

- “Algorithm-Driven”
  - When *program* needs information from user, it asks for it
  - Program is in control
  - Typical in non-GUI environments (examples: payroll, batch simulations)
- “Event Driven”
  - When *user* wants to do something, he/she signals to the program
    Moves or clicks mouse, types, etc.
  - These signals come to the program as “events”
  - Program is interrupted to deal with the events
  - User has more control
  - Typical in GUI environments

A Bit of Java History

- Java 1.0: AWT (Abstract Windowing Toolkit)
- Java 1.1: AWT with new event handling model
- Java 1.2 (aka Java 2): Swing
  - Greatly enhanced user interface toolkit built on top of AWT
  - Same basic event handling model as in Java 1.1 AWT
- Java 1.3, 1.4
  - Bug fixes and significant performance improvements; no major revolution
- Naming
  - Most Swing components start with J.
  - No such standard for AWT components

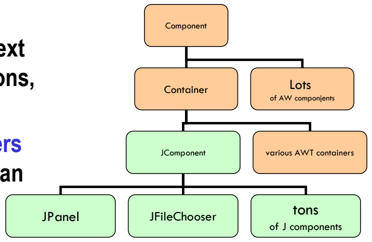
Bit o' Advice

- Use Swing whenever you can
- Use AWT whenever you have to

  (mostly to support older windows browsers that don't have the current Sun Java plugin)

Components & Containers

- Every GUI-related component descends from Component, which contains dozens of basic methods and fields common to all AWT/Swing component
- "Atomic" components: labels, text fields, buttons, check boxes, icons, menu items, …
- Some components are Containers – components like panels that can contain other nested subcomponents



Types of Containers

- Top-level containers: JFrame,JDialog, JApplet
  - Often correspond to OS Windows
- Mid-level containers: panels, scroll panes, tool bars, …
  - can contain certain other components
  - JPanel is best for general use
  - An Applet is a special kind of container
- Specialized containers: menus, list boxes, combo boxes...
- Technically, all J components are containers

JFrame – A Top-Level Window

- Top-level application window

```
JFrame win = new JFrame("Optional Window Title");
```

- Some common methods

```
setSize(int width, int height);     // frame width and height
setBackground(Color c);             // background color
show( );                            //make visible (for the first time)
repaint( );                         // request repaint after content change
setPreferredSize(Dimension d);      // default size for window; also can set min
                                    //      and max sizes
dispose( );                         // get rid of the window when done
```

JPanel – A General Purpose Container

- Commonly added to a window to provide a space for graphics, or collections of buttons, labels, etc.
- JPanels can be nested to any depth
- Many methods in common with JFrame (since both are ultimately instances of Component)

```
setSize(int width, int height);
setBackground(Color c);
setPreferredSize(Dimension d);
```

  - Bit o' advice: Can't find the method you're looking for?

    Check the superclass.

Adding Components to Containers

- Swing containers have a "content pane" that manages the components in that container

  [Differs from original AWT containers, which managed their components directly]

- To add a component to a container, get the content pane, and use its add method

```
JFrame jf = new JFrame( );
JPanel panel = new JPanel( );
jf.getContentPane( ).add(panel);
    or
Container cp = jf.getContentPane( );
cp.add(panel);
```

Non-Component Classes

- Not all classes are GUI components
- AWT
  - Color, Dimension, Font, layout managers
  - Shape and subclasses like Rectangle, Point, etc.
  - Graphics
- Swing
  - Borders
  - Further geometric classes
  - Graphics2D
- Neither AWT nor Swing
  - Images, Icons

Layout Managers

- What happens if we add several components to a container?
  - What are their relative positions?
- Answer: each container has a layout manager. Some kinds:
  - FlowLayout (left to right, top to bottom)
  - BorderLayout("center", "north", "south", "east", "west")
  - GridLayout (2-D grid)
  - GridBagLayout (makes HTML tables look simple); others
- Default LayoutManager for JFrame is BorderLayout
- Default for JPanel is FlowLayout

pack and *validate*

- Container state is "valid" or "invalid" depending on whether layout manager has arranged components since last change
- When a container is altered, either by adding components or changes to components (resized, contents change, etc.), the layout needs to be updated (i.e., the container state needs to be set to valid)
  - Swing does this automatically more often than AWT, but not always
- Common methods after changing layout
  - validate( ) – redo the layout to take into account new or changed components
  - pack( ) – redo the layout using the preferred size of each component

Layout Example

- Create a JFrame with a button at the bottom and a panel in the center

```
JFrame frame = new JFrame("Trivial Window"); //default layout: Border
JPanel panel = new JPanel( );
JLabel label = new JLabel("Smile!");
label.setHorizontalAlighment(SwingConstants.CENTER);
Container cp = frame.getContentPane( );
cp.add(panel, BorderLayout.CENTER);
cp.add(label, BorderLayout.SOUTH);
```

Graphics and Drawing

- Simple things like labels have suitable default code to paint themselves
- For more complex graphics, extend a suitable class and override the (empty) inherited method *paintcomponent* to draws its contents
  - (Different from AWT, where you overrode *paint* – don't do that in swing!)

```
public class Drawing extends JPanel {
...
    /** Repaint this Drawing whenever requested by the system */
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.setColor(Color.green);
        g.drawOval(40,30,100,100);
        g.setColor(Color.red);
        g.fillRect(60, 50, 60, 60);
    }
```

paintComponent

- Method paintComponent is called by the underlying system *whenever it needs the window to be repainted*
  - Triggered by window being move, resized, uncovered, expanded from icon, etc.
  - Can happen anytime – you don't control when
- If your code does something that requires repainting, call method repaint( )
  - Requests that paintComponent be called sometime in the future, when convenient for underlying system window manager

Painter's Rules

- *Always* override paintComponent( ) of any component you will be drawing on
  - Not necessary if you make simple changes, like changing background color, title, etc. that don't require a graphics object
- *Always* call super.paintComponent(g) to paint the background
- *Never* call paint( ) or paintComponent( ). Never means never!
  - This is a hard rule to understand. Follow it anyway.
- *Always* paint the entire picture, from scratch
- Don't create a Graphics object to draw with
  - only use the one given to you as a parameter of paintComponent( )
  - and, don't save that object to reuse later!
  - This rule is bent in advanced graphics applications

What Happens If You D ollow The Rules...

Classes Graphics and Graphics2D

- The parameter to *paintComponent* or *paint* is a graphics context where the drawing should be done
  - In Swing components, the parameter has static type Graphics, but dynamic type Graphics2D, a subclass of Graphics

    Cast it to Graphics2D if you want to use the newer, more sophisticated graphics operations
- More procedural-like interface than uwcse.GWindow (if you've used that)
  - Call Graphics methods to draw on the Graphics object

    [instead of creating new shape objects and adding them to the window]

Graphics 2D

- In the Graphics 2D package, many graphical objects implement the Shape interface
  - When possible, chose a Shape rather than a non-Shape
- Lots of methods available to draw various kinds of outline and solid shapes and control colors and fonts

Learning Graphics2D

- In reading and experimenting, focus on these classes:
  - JPanel (and ancestors)
  - (interface) Shape
  - Line2D
  - Polygon
  - Graphics2D, especially these methods:

    draw(Shape)

    draw(String, int, int)

    fill(Shape)

    setColor(Color)

    Avoid methods like drawLine, drawPolygon, etc.
- (But you can use the older AWT graphics if you want)

Roadmap

- Future: Events
  - User interaction
  - GUI components
- What to do
  - Start reading textbook chs. 19 and 20
  - Browse the Swing tutorial and Java Swing/AWT documentation from Sun to start to feel your way around