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 and Unix later
- User sees and interacts with "controls" or "components"
  - 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
- "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

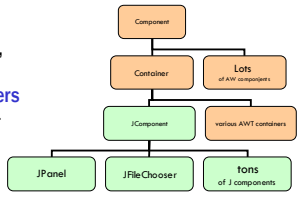
- Java 1.0: AWT (Abstract Windowing Toolkit)
- Java 1.1: AWT with new event handling model
- Java 1.2 (aka Java 2): Swing
  - Greatly enhanced user interface toolkit built on top of AWT
  - Same basic event handling model as in Java 1.1 AWT
- Java 1.3, 1.4
  - Bug fixes and significant performance improvements; 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
  - For classes that Swing doesn't have
  - For applets to run on systems without Java 1.2 or higher
    includes Windows systems unless Java Plug-in is installed

Components & Containers

- Every GUI-related component descends from Component
  - contains dozens of basic methods and fields common to all AWT/Swing components
  - "Atomic" components: labels, text fields, buttons, check boxes, icons, menu items, ...
- Some components are Containers
  - components that can contain other subcomponents
  - Can be nested as deep as needed



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...

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)
pack( );                            //resize the frame based on subcomponents
repaint( );                         // request repaint after content change
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**

Positioning Components in a Container

- What happens if we add several components to a container?
  - Want some control over position
  - Don't want them to overlap
  - Don't want them to overflow the container
- If user resizes the containers...
  - Would be nice if the components adjusted, too

Java Solution: Layout Managers

- Each container has a layout manager
  - Determines how the components are arranged
- Several common ones:
  - FlowLayout (left to right, top to bottom)
  - BorderLayout("center", "north", "south", "east", "west")
  - GridLayout (2-D grid)
  - GridBagLayout (makes HTML tables look simple)
  - many others
- Container state is "valid" or "invalid" depending on whether layout manager has arranged components since last change
- Default LayoutManager for JFrame is BorderLayout
- Default for JPanel is FlowLayout

pack and *validate*

- 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 (sub-)components
  - pack( ) – redo the layout using the preferred size of each (sub-) 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);
```

- Try it at home!
  - And you'll find it doesn't *quite* work...
  - Look at previous slides...

Graphics and Drawing

- The windows, panes, and other components supplied with Swing are sufficient for predefined GUI components
- For more complex graphics, extend a suitable class and override the (empty) inherited method *paintcomponent* that draws its contents

```
public class Drawing extends JPanel {
...
    /** Repaint this Drawing whenever requested by the system */
    public void paintComponent(Graphics g) {
      Graphics2D g2 = (Graphics2D) g;  //always do this for Swing components
      g2.setColor(Color.green);
      g2.drawOval(40,30,100,100);
      g2.setColor(Color.red);
      g2.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
  - In AWT days, you overrode paint( ). With Swing, it is best to leave paint alone and override paintComponent
- 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

Classes Graphics and Graphics2D

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

    so cast it to a 2D and use that
- Useful methods:

```
g.setColor( ); //sets of the color for the next thing drawn
g.draw(Shape); //draw the outline of a Shape
g.fill(Shape); //draw a Shape filled with color
```


Summer 2003 Project 3 Note

- In this project, you do not have to subclass a component and override *paintComponent*
- All of that set-up has been taken care of
- Instead, the AbstractVehicle and Background classes have method called *paint* with a similar role:

```
public abstract void draw(Graphics2D g);
```

- Override *draw* and follow the same rules as for *paintComponent*

Drawing Graphical Objects

- 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
  - setColor, setFont, drawArc, drawLine, fillPolygon, drawOval, fillRect, many others

Painter's Rules

- *Always* override *paintComponent*( ) of any Swing 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
- *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* or *Graphics2D* 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...

Preparing for Future Projects

- 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.

Summer 2003
Restating these Rules for Project 3

- Don't call *draw*
- Draw the whole icon (vehicle or background) from scratch each time *draw* is called
- Don't call *draw*
- Don't call *draw*

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

    Focus on the classes listed previously