The plan

Today: introduction to Java graphics and Swing/AWT libraries

Then: event-driven programming and user interaction

None of this is comprehensive – only an overview and guide to what you should expect to be out there

– Some standard terminology and perspective

Credits: material taken from many places; including slides and materials by Ernst, Hotan, Mercer, Notkin, Perkins, Stepp; Reges; Sun/Oracle docs & tutorial; Horstmann; Wikipedia; others, folklore, …

Why study GUIs?

• Er, because graphical user interfaces are pretty common (duh 😊)
  – And it’s fun!

• Classic example of using inheritance to organize large class libraries
  – The best (?) example of OOP’s strengths

• Work with a huge API – and learn how (not) to deal with all of it

• Many core design patterns show up: callbacks, listeners, event-driven programs, decorators, façade

What not to do…

• Don’t try to learn the whole library: There’s way too much

• Don’t memorize – look things up as you need them

• Don’t miss the main ideas, fundamental concepts

• Don’t get bogged down implementing eye candy

Main topics to learn

Organization of the AWT/Swing library
  – Names of essential widgets/components

Graphics and drawing
  – Repaint callbacks, layout managers, etc.

Handling user events

Building GUI applications
  – MVC, user events, updates, …
A very short history (1)

Java’s standard libraries have supported GUIs from the beginning

Original Java GUI: AWT (Abstract Window Toolkit)
  – Limited set of user interface elements (widgets)
  – Mapped Java UI to host system UI widgets
  – Lowest common denominator
  – “Write once, debug everywhere”

A very short history (2)

Swing: Newer GUI library, introduced with Java 2 (1998)

Basic idea: underlying system provides only a blank window
  – Swing draws all UI components directly
  – Doesn’t use underlying system widgets

Not a total replacement for AWT: Swing is implemented on top of core AWT classes and both still coexist

Use Swing, but deal with AWT when you must

GUI terminology

window: A first-class citizen of the graphical desktop
  – Also called a top-level container
  – Examples: frame, dialog box, applet

component: A GUI widget that resides in a window
  – Called controls in many other languages
  – Examples: button, text box, label

container: A component that hosts (holds) components
  – Examples: frame, applet, panel, box

Some components...

Component and container classes

• Every GUI-related class descends from Component, which contains dozens of basic methods and fields
  – Examples: getBounds, isVisible, setForeground, ...
• “Atomic” components: labels, text fields, buttons, check boxes, icons, menu items...
• Many components are containers – things like panels (JPanel) that can hold nested subcomponents

Swing/AWT inheritance hierarchy
Component properties

Zillions. Each has a **get** (or **is**) accessor and a **set** modifier. Ex: **getColor**, **setFont**, **isVisible**, …

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>background</td>
<td>Color</td>
<td>background color behind component</td>
</tr>
<tr>
<td>border</td>
<td>Border</td>
<td>border line around component</td>
</tr>
<tr>
<td>enabled</td>
<td>boolean</td>
<td>whether it can be interacted with</td>
</tr>
<tr>
<td>focusable</td>
<td>boolean</td>
<td>whether key text can be typed on it</td>
</tr>
<tr>
<td>font</td>
<td>Font</td>
<td>font used for text in component</td>
</tr>
<tr>
<td>foreground</td>
<td>Color</td>
<td>foreground color of component</td>
</tr>
<tr>
<td>height, width</td>
<td>int</td>
<td>component's current size in pixels</td>
</tr>
<tr>
<td>visible</td>
<td>boolean</td>
<td>whether component can be seen</td>
</tr>
<tr>
<td>tooltip text</td>
<td>String</td>
<td>text shown when hovering mouse</td>
</tr>
<tr>
<td>size, minimum / maximum / preferred size</td>
<td>Dimension</td>
<td>various sizes, size limits, or desired sizes that the component may take</td>
</tr>
</tbody>
</table>

Types of containers

- **Top-level containers:** **JFrame**, **JDialog**, …
  - Often correspond to OS windows
  - Usually a “host” for other components
  - Live at top of UI hierarchy, not nested in anything else
- **Mid-level containers:** panels, scroll panes, tool bars
  - Sometimes contain other containers, sometimes not
  - **JPanel** is a general-purpose component for drawing or hosting other UI elements (buttons, etc.)
- **Specialized containers:** menus, list boxes, …
- **Technically, all JComponents are containers**

**JFrame** – top-level window

- Graphical window on the screen
- Typically holds (hosts) other components
- Common methods:
  - **JFrame(String title)**: constructor, title optional
  - **setDefaultCloseOperation(int what)**
    - What to do on window close
    - **JFrame.EXIT_ON_CLOSE** terminates application
  - **setSize(int width, int height)**: set size
  - **add(Component c)**: add component to window
  - **setVisible(boolean b)**: make window visible or not

**Example**

*SimpleFrameMain.java*

**JPanel** – a general-purpose container

- Commonly used as a place for graphics, or to hold a collection of button, labels, etc.
- Needs to be added to a window or other container: `frame.add(new JPanel(...))`
- **JPanels** can be nested to any depth
- Many methods/fields in common with **JFrame** (since both inherit from **Component**)
  - Advice: can’t find a method/field? Check the superclasses
- A particularly useful method:
  - **setPreferredSize(Dimension d)**

**Containers and layout**

- What if we add several components to a container?
  - How are they positioned relative to each other?
  - Answer: each container has a **layout manager**
Layout managers

Kinds:
- FlowLayout (left to right [changeable], top to bottom)
  - Default for JPanel
  - Each row centered horizontally [changeable]
- BorderLayout ("center", "north", "south", "east", "west")
  - Default for JFrame
  - No more than one component in each of 5 regions
  - (Of course, component can itself be a container)
- GridLayout (regular 2-D grid)
- Others... (some are incredibly complex)

FlowLayout and BorderLayout should be good enough for now...

pack()

Once all the components are added to their containers, do this to make the window visible:
pack();
setVisible(true);

pack() figures out the sizes of all components and calls the container’s layout manager to set locations in the container
  – (recursively as needed)

If your window doesn't look right, you may have forgotten pack()

Example

SimpleLayoutMain.java

Graphics and drawing

So far so good – and very boring…

What if we want to actually draw something?
  – A map, an image, a path, …?

Answer: Override method paintComponent
  – Components like JLabel provide a suitable paintComponent
    that (in JLabel’s case) draws the label text
  – Other components like JPanel typically inherit an empty
    paintComponent and can override it to draw things

Note: As we’ll see, we override paintComponent but we don’t call it

Example

SimplePaintMain.java

Graphics methods

Many methods to draw various lines, shapes, etc., …

Can also draw images (pictures, etc.):
  – In the program (not in paintComponent):
    • Use AWT’s “Toolkit” to load an image:
      Image pic =
      Toolkit.getDefaultToolkit().
      getImage(file-name (with path));
  – Then in paintComponent:
    g.drawImage(pic, ...);
### Graphics vs Graphics2D

Class `Graphics` was part of the original Java AWT
- Has a procedural interface:
  - `g.drawRect(...)`, `g.fillOval(...)`, ...

Swing introduced `Graphics2D`
- Added an object interface – create instances of `Shape` like `Line2D`, `Rectangle2D`, etc., and add these to the `Graphics2D` object

Actual parameter to `paintComponent` is always a `Graphics2D`
- Can always cast this parameter from `Graphics` to `Graphics2D`
- `Graphics2D` supports both sets of graphics methods
- Use whichever you like for CSE 331

### So who calls `paintComponent`? And when??

- **Answer:** the window manager calls `paintComponent` whenever it wants! (a callback!)
  - When the window is first made visible, and whenever after that some or all of it needs to be repainted
- **Corollary:** `paintComponent` must always be ready to repaint regardless of what else is going on
  - You have no control over when or how often
  - You must store enough information to repaint on demand
- If "you" want to redraw a window, call `repaint()` from the program (not from `paintComponent`)
  - Tells the window manager to schedule repainting
  - Window manager will call `paintComponent` when it decides to redraw (soon, but maybe not right away)
  - Window manager may combine several quick `repaint()` requests and call `paintComponent()` only once

### Example

**FaceMain.java**

### How repainting happens

**program**

```
repaint()
```

**window manager (UI)**

```
paintComponent(g)
```

It’s worse than it looks!

Your program and the window manager are running concurrently:
- Program thread
- User Interface thread

Do not attempt to mess around – follow the rules and nobody gets hurt!

### Crucial rules for painting

- Always override `paintComponent(g)` if you want to draw on a component
- Always call `super.paintComponent(g)` first
- **NEVER, EVER, EVER** call `paintComponent` yourself
- Always paint the entire picture, from scratch
- Use `paintComponent`’s `Graphics` parameter to do all the drawing. **ONLY** use it for that. Don’t copy it, try to replace it, or mess with it. It is quick to anger.
- **DON’T** create new `Graphics` or `Graphics2D` objects

Fine print: Once you are a certified™ wizard, you may find reasons to do things differently, but that requires deeper understanding of the GUI library’s structure and specification

### What’s next – and not

**Major topic for next lecture is how to handle user interactions**
- We already know the core idea: it’s a big-time use of the observer pattern

Beyond that you’re on your own to explore all the wonderful widgets in Swing/AWT.
- Have fun!!
- (But don’t sink huge amounts of time into eye candy)