Java Graphics & GUIs
(and Swing/AWT libraries)

CSE 331
Software Design & Implementation

Why study GUIs?

- Learn about *event-driven programming* techniques
- Practice learning and using a large, complex API
- A chance to see how it is designed and learn from it:
  - design patterns: model-view separation, callbacks, listeners, inheritance vs. delegation
  - refactoring vs. reimplementing an ailing API
- Because GUIs are neat!

*Caution:* There is way more here than you can memorize.
- Part of learning a large API is "letting go."
- First, learn the fundamental concepts and general ideas.
- Then, look things up as you need them
- Don’t get bogged down implementing eye candy
References

Today: Java graphics and Swing/AWT class libraries
Only an introduction! Also see
• Sun/Oracle Java tutorials
  http://docs.oracle.com/javase/tutorial/uiswing/index.html
• Extra slides, on class website
• Core Java vol. I by Horstmann & Cornell
• If you have another favorite, use it

Next lecture:
Event-driven programming and user interaction
Outline

Organization of the Swing/AWT library

Graphics and drawing
  Repaint callbacks, layout managers, etc.

Handling user events

Building GUI applications
  MVC, user events, updates, &c
Java GUI libraries

- **Swing**: the main Java GUI library
  - *Benefits*: Features; cross-platform compatibility; OO design
  - Paints GUI controls itself pixel-by-pixel
    - Does not delegate to OS’s window system

- **Abstract Windowing Toolkit (AWT)**: Sun's initial GUI library
  - Maps Java code to each operating system's real GUI system
  - *Problems*: Limited to lowest common denominator (limited set of UI widgets); clunky to use.

- Advice: Use Swing. You occasionally have to use AWT (Swing is built on top of AWT). Beware: it’s easy to get them mixed up.
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
   Also called *controls* in many other languages
   Examples: button, text box, label

**container**: A component that hosts (holds) components
   Examples: panel, box
# Components

<table>
<thead>
<tr>
<th>JButton</th>
<th>JCheckBox</th>
<th>JRadioBox</th>
<th>JLabel</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="OK" /></td>
<td><img src="image" alt="Check" /></td>
<td><img src="image" alt="Radio" /></td>
<td><img src="image" alt="Image and Text" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>JTextField</th>
<th>JSlider</th>
<th>JToolBar</th>
<th>JMenuBar, JMenu, JMenuItem</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Years: 30" /></td>
<td><img src="image" alt="Frames Per Second" /></td>
<td><img src="image" alt="Images" /></td>
<td><img src="image" alt="Menu Items" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>JComboBox</th>
<th>JList</th>
<th>JTable</th>
<th>JTree</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Pig" /></td>
<td><img src="image" alt="January" /></td>
<td><img src="image" alt="FirstName" /></td>
<td><img src="image" alt="Music" /></td>
</tr>
<tr>
<td>Bird</td>
<td>February</td>
<td>Last Name</td>
<td>Classical</td>
</tr>
<tr>
<td>Cat</td>
<td>March</td>
<td>Favorite Food</td>
<td>Beethoven</td>
</tr>
<tr>
<td>Dog</td>
<td>April</td>
<td></td>
<td>Brahms</td>
</tr>
<tr>
<td>Rabbit</td>
<td></td>
<td></td>
<td>Mozart</td>
</tr>
<tr>
<td>Pig</td>
<td></td>
<td></td>
<td>Jazz</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Rock</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Component & container classes

Every GUI-related class descends from Component

Containers can hold nested subcomponents

“Atomic” components: labels, text fields, buttons, check boxes, icons, menu items...

Component

Container

Jcomponent

Lots of AWT components

Various AWT containers

Jpanel

JFileChooser

Tons of Jcomponents
<table>
<thead>
<tr>
<th>Component (AWT)</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Window</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Frame</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JFrame (Swing)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JDialog</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Container</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JComponent (Swing)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JButton</td>
<td>JColorChooser</td>
<td>JFileChooser</td>
</tr>
<tr>
<td>JComboBox</td>
<td>JLabel</td>
<td>JList</td>
</tr>
<tr>
<td>JMenuBar</td>
<td>JOptionPane</td>
<td>JPanel</td>
</tr>
<tr>
<td>JPopupMenu</td>
<td>JProgressBar</td>
<td>JScrollPane</td>
</tr>
<tr>
<td>JScrollPane</td>
<td>JSlider</td>
<td>JSpinner</td>
</tr>
<tr>
<td>JSplitPane</td>
<td>JTabbedPane</td>
<td>JTable</td>
</tr>
<tr>
<td>JToolBar</td>
<td>JTree</td>
<td>JTextArea</td>
</tr>
<tr>
<td>JTextField</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
### Component fields (actually properties)

Each has a `get` (or `is`) accessor and a `set` modifier. Examples: `getColor`, `setFont`, `isVisible`, ...

<table>
<thead>
<tr>
<th>name</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>background</td>
<td>background color behind component</td>
</tr>
<tr>
<td>border</td>
<td>border line around component</td>
</tr>
<tr>
<td>enabled</td>
<td>whether it can be interacted with</td>
</tr>
<tr>
<td>focusable</td>
<td>whether key text can be typed on it</td>
</tr>
<tr>
<td>font</td>
<td>font used for text in component</td>
</tr>
<tr>
<td>foreground</td>
<td>foreground color of component</td>
</tr>
<tr>
<td>height, width</td>
<td>component's current size in pixels</td>
</tr>
<tr>
<td>visible</td>
<td>whether component can be seen</td>
</tr>
<tr>
<td>tooltip text</td>
<td>text shown when hovering mouse</td>
</tr>
<tr>
<td>size, minimum / maximum / preferred size</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
  – Can be used by themselves, but usually as 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 J-components are containers
JFrame – top-level window

Graphical window on the screen
Typically holds (hosts) other components
Common methods:

- `JFrame(String title)` – constructor, title optional
- `setSize(int width, int height)` – set size
- `add(Component c)` – add component to window
- `setVisible(boolean v)` – make window visible or not. Don’t forget this!
Example

SimpleFrameMain.java
More JFrame

- **public void setDefaultCloseOperation(int op)**
  Makes the frame perform the given action when it closes.
  - Common value passed: `JFrame.EXIT_ON_CLOSE`
  - If not set, the program will never exit even if the frame is closed.

- **public void setSize(int width, int height)**
  Gives the frame a fixed size in pixels.

- **public void pack()**
  Resizes the frame to fit the components inside it snugly.
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

```java
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 superclass(es)

Some new methods. Particularly useful:

```java
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, top to bottom) – default for JPanel
- **BorderLayout** (“center”, “north”, “south”, “east”, “west”) – default for JFrame
- **GridLayout** (regular 2-D grid)
- others... (some are incredibly complex)

The first two should be good enough for now....
Place components in a *container*; add the container to a frame.

- **container**: An object that stores components and governs their positions, sizes, and resizing behavior.
Once all the components are added to their containers, do this to make the window visible:

```java
pack();
setVisible(true);
```

`pack()` figures out the sizes of all components and calls the 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
Sizing and positioning

How does the programmer specify where each component appears, how big each component should be, and what the component should do if the window is resized / moved / maximized / etc.?

- **Absolute positioning** (C++, C#, others):
  Programmer specifies exact pixel coordinates of every component.
  - "Put this button at (x=15, y=75) and make it 70x31 px in size."

- **Layout managers** (Java):
  Objects that decide where to position each component based on some general rules or criteria.
  - "Put these four buttons into a 2x2 grid and put these text boxes in a horizontal flow in the south part of the frame."
**JFrame as container**

A `JFrame` is a container. Containers have these methods:

- **public void `add`**(Component comp)
  ```java
  public void add(Component comp, Object info)
  ```
  Adds a component to the container, possibly giving extra information about where to place it.

- **public void `remove`**(Component comp)

- **public void `setLayout`**(LayoutManager mgr)
  Uses the given layout manager to position components.

- **public void `validate`()**
  Refreshes the layout (if it changes after the container is onscreen).
Preferred sizes

• Swing component objects each have a certain size they would "like" to be: Just large enough to fit their contents (text, icons, etc.).
  – This is called the *preferred size* of the component.
  – Some types of layout managers (e.g. `FlowLayout`) choose to size the components inside them to the preferred size.
  – Others (e.g. `BorderLayout`, `GridLayout`) disregard the preferred size and use some other scheme to size the components.
FlowLayout

public FlowLayout()

• treats container as a left-to-right, top-to-bottom "paragraph".
  – Components are given preferred size, horizontally and vertically.
  – Components are positioned in the order added.
  – If too long, components wrap around to the next line.

myFrame.setLayout(new FlowLayout());
myFrame.add(new JButton("Button 1"));

– The default layout for containers other than JFrame (seen later).
public BorderLayout()

- Divides container into five regions:
  - **NORTH** and **SOUTH** regions expand to fill region horizontally, and use the component's preferred size vertically.
  - **WEST** and **EAST** regions expand to fill region vertically, and use the component's preferred size horizontally.
  - **CENTER** uses all space not occupied by others.

```java
myFrame.setLayout(new BorderLayout());
myFrame.add(new JButton("Button 1"), BorderLayout.NORTH);
```

- This is the default layout for a JFrame.
GridLayout

public GridLayout(int rows, int columns)

- Treats container as a grid of equally-sized rows and columns.
- Components are given equal horizontal / vertical size, disregarding preferred size.
- Can specify 0 rows or columns to indicate expansion in that direction as needed.
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`
   Method in `JComponent` that draws the component
In `JLabel`’s case, it draws the label text
Example

SimplePaintMain.java
Many methods to draw various lines, shapes, etc., ...

Can also draw images (pictures, etc.). Load the image file into an `Image` object and use `g.drawImage(...)`:

- In the program (not in `paintComponent`):
  ```java
  Image pic =
  Toolkit.getDefaultToolkit()
      .getImage(image path);
  ```
- Then in `paintComponent`:
  ```java
  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 a object interface – create instances of `Shape` like `Line2D`, `Rectangle2D`, etc., and add these to the `Graphics2D` object

Parameter to `paintComponent` is always `Graphics2D`. Can always cast it to that class. `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!!!*
  - When the window is first made visible, and whenever after that it is needed
- *Corollary:* `paintComponent` must *always* be ready to repaint – regardless of what else is going on
  - You have no control over when or how often – 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)
Example

FaceMain.java
How repainting happens

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!
Rules for painting – Obey!

- Always override `paintComponent(g)` if you want to draw on a component.
- Always call `super.paintComponent(g)` first.
- **NEVER** call `paintComponent` yourself. That means ABSOLUTELY POSITIVELY NEVER!!!
- 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, permanently side-effect it, etc. 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 you aren’t there yet.
What’s next – and not

Major topic next time is how to handle user interactions
    Key idea: 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)