GUI Programming and Event-driven Programming

Slides due to Michael Ernst, University of Washington

Announcements

- CHECK YOUR GRADES
  - Quiz 1-7, HW 1-6, Exam 1-2 now all in Submitty
- HW8 due today
- HW9 out tonight
  - A GUI Interface for your path finding algorithm

Outline of Today’s Class

- Organization of the Java Swing/AWT library
  - Components and containers
  - Layout managers
  - Graphics and drawing
- Events
  - Event objects
  - Event listeners
- Anonymous inner classes
- Interaction between UI and program threads

Why Study GUIs?

- Practice design patterns and concepts
  - Model View Controller (i.e., Observer), Composite
  - Callbacks, inheritance vs. delegation
- Learn about event-driven programming
- Practice learning and using a large API
- There is way more than you can memorize
  - First, learn fundamentals and general ideas
  - Then, look things up as you need them!
  - Don’t get bogged down implementing eye candy

Aside: Callbacks

- A callback occurs when library code calls a user-defined method

```java
class InstrumentedHashSet extends HashSet {
    private int addCount = 0;
    public InstrumentedHashSet(Collection c) {
        super(c);
    }
    public boolean add(Object o) {
        addCount++;
        return super.add(o);
    }
    public boolean addAll(Collection c) {
        addCount += c.size();
        return super.addAll(c);
    }
    public int getAddCount() { return addCount; }
}
```

```java
InstrumentedHashSet s = new InstrumentedHashSet();
System.out.println(s.getAddCount()); // 0
s.addAll(Arrays.asList("One","Two");
System.out.println(s.getAddCount()); // Prints?
```
References

- Sun/Oracle Java Swing tutorial: http://docs.oracle.com/javase/tutorial/uiswing/index.html
- Core Java vol. I by Horstmann and Cornell
- Other…

Java GUI Libraries

- Swing: the main Java GUI library
  - Paints GUI components itself pixel-by-pixel
  - Does not delegate to the OS window system
  - Benefits: expanded set of widgets and features, cross-platform compatibility, OO Design
- Abstract Windowing Toolkit (AWT): Sun’s initial GUI library
  - Maps Java code to each OS’s windowing system
  - Problems: limited set of widgets, clunky to use

GUI Terminology

- window: A first-class citizen of the graphical desktop. E.g., frame
- component: A GUI widget that resides in a window. E.g., button, text box, label
- container: A component that holds components. What design pattern is this? E.g., panel, box

Components

- The Component Hierarchy
  - Component (AWT, java.awt.Component)
  - Container (AWT, java.awt.Container)
    - Window
      - JFrame (Swing)
    - JWindow
  - JComponent (Swing, java.swing.JComponent)
    - JButton
    - JColorChooser
    - JFileChooser
  - JComponent (Swing, java.swing.JComponent)
    - JComboBox
    - JLabel
    - JList
  - JComponent (Swing, java.swing.JComponent)
    - JMenuBar
    - JOptionPane
    - JPanel
    - JPopupMenu
    - JProgressBar
    - JScrollPane
Component Properties

- Each property has a `get` (or `is`) accessor and a `set` modifier. E.g., `setFont`, `isVisible`
- Example properties
  - `background` – color behind component
  - `border` - border line around component
  - `enabled` – whether it can be interacted with
  - `focusable` – whether key text can be typed on it
  - `font` – font used for text in component
  - Etc.

Containers

- Windows are top-level containers: `JFrame`, `JDialog`
  - Live at the top of UI hierarchy, not nested
  - Can be used by themselves, but usually as a host for other components
- Mid-level containers: `JPanel`, `JToolBar`
  - Sometimes contain other components, sometimes not
  - `JPanel` is a general-purpose component for drawing or hosting other UI elements
- Specialized containers: menus, list boxes...

JFrame – Top-level Container (Window)

- Graphical window on the screen
- Typically holds other components
- Commonly used methods:
  - `JFrame(String title)` – title optional
  - `setSize(int width, int height)`
  - `add(Component c)` – add component to window
  - `setVisible(boolean v)` – don’t forget this!
- Example:
  - `www.cs.rpi.edu/~milanova/csci2600/handouts/SimpleFrameMain.java`

JFrame

- `setDefaultCloseOperation(int o)` – makes the frame perform the given action when it closes
  - Common value: `JFrame.EXIT_ON_CLOSE`
  - If not set, program will never exit even if frame is closed. Don’t forget this!
- `pack()` – resizes frame to fit components

JPanel – a General Purpose Container

- Used to group other containers: a place for graphics, or to hold buttons, labels, etc.
- Must be added to a frame or other container
  - `frame.add(new JPanel(...));`
- `JPanels` can be nested at any depth
- Many methods in common with `JFrame`. Why?
- Some new methods
  - E.g., `setPreferredSize(Dimension d)`

Layout Manager – positions components in container

- Each container has a layout manager
- `FlowLayout` (left to right, top to bottom) - default for `JPanel`
- `BorderLayout` (“center”, “north”, “south”, “east”, “west”) – default for `JFrame`
### Layout Manager – Positions Components in Container

- **GridLayout** (a 2D grid):
- **BoxLayout**
- Other... Some are very complex

### Layout Managers

- **Sizing and positioning**
- **Absolute positioning (C++, C#, other)**
  - Programmer specifies exact pixel coordinates of every component. E.g., “Put this button at (x=15, y=75) and make it 70x31 pixel in size”
- **Layout managers (Java):**
  - Objects that decide where to position each component based on some general rules or criteria. E.g., “Put these four buttons into a 2x2 “grid” and put these text boxes in a “horizontal flow” in the “south” part of the frame”

### Preferred Sizes

- Swing component objects all 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 (some dimension of) the preferred size and use some other scheme to size components

### JFrame – the top-level container

- **add(Component c)**,
- **add(Component c, Object info)**
  - Add component to the container, possibly giving extra info about where to place it:
  - ```java
  frame.add(new JButton("Button 1"), BorderLayout.NORTH);
  ```
- **remove(Component c)**
- **setLayout(LayoutManager mgr)**
- **validate()**
  - validate refreshes the layout (if it changes after container is onscreen). Time-consuming.
**General Structure of GUI Application**

- Place components in a container (**JPanel**) then add container to frame (**JFrame**).
- Container stores components and governs their positions, sizes and resizing behavior.

**Graphing and Drawing**

- What if we want to draw something? An image, a path?
- Answer: Extend **JPanel** and override method **paintComponent**.
- Method in **JComponent** that draws the component.
- Example:
  - [www.cs.rpi.edu/~milanova/csci2600/handouts/SimplePaintMain.java](http://www.cs.rpi.edu/~milanova/csci2600/handouts/SimplePaintMain.java)

**Graphics vs. Graphics2D**

- **Graphics** is part of the original Java AWT.
  - Has procedural interface: e.g., \( g \text{.drawRect}(...) \)
- Swing introduced **Graphics2D**:
  - Added an OO interface: create instance of **Shape**, e.g., **Line2D**, **Rectangle2D**, etc. Then call draw with respective arg: \( g \text{.draw}(\text{Shape } s) \)
- Argument passed to **paintComponent** is always a **Graphics2D**. Can always cast it to that class. **Graphics2D** supports both sets of graphics methods. Use whichever you like.

**Who Calls **paintComponent**?**

- The window manager calls **paintComponent** whenever it wants!!!
  - When the window is first made visible and whenever after that it is needed. **Never call paintComponent yourself!**
- Thus, **paintComponent** must always be ready to repaint – must store all information needed.
- If you must redraw a window, call **repaint()**:
  - Tells the window manager to schedule repainting.
  - Window manager will call **paintComponent** when it decides to redraw (soon, but not right away).

**Graphics Methods**

- Many methods to draw various lines, shapes.
- Can also draw image (pictures, etc.). Load the image file into an Image object and use \( g \text{.drawImage}(...) \).
- Example:
  - In the program (maybe not in **paintComponent**):
    - \( \text{Image image} = \text{ImageIO.read(file)} \)
  - Then in **paintComponent**:
    - \( g \text{.drawImage(image, ...)} \)
Rules for Painting. Obey!

- Always override `paintComponent(Graphics)` if you want to draw a component.
- Always call `super.paintComponent(g)` first from your `paintComponent(...)` method. Why?
- **NEVER** call `paintComponent` yourself.
- Always paint the entire picture from scratch.

Fine print: once you are certified™ wizard you may find reasons to do things differently, but for now follow the rules.

Outline of Today’s Class

- Organization of the Java Swing/AWT library
  - Components and containers
  - Layout managers
  - Graphics and drawing
- Events
  - Event objects
  - Event listeners
  - Anonymous inner classes
  - Interaction between UI and program threads

Event-driven Programming

- A style of programming where flow of execution is dictated by events
  - Program loads, then waits for user input events
  - As event occurs, program runs code to respond
  - Flow of what code is executed is determined by the series of events that occur
- In contrast, application- or algorithm-driven control expects input in well-defined places
  - Typical for large non-GUI applications

Event-driven Programming

- Main body of the program is an event loop. Like this:
  ```java
do {
    e = getNextEvent();
    // process event e;
} while (e != quit)
```

GUI Events and Listeners

- **event**: an object that represents the user’s interaction with a GUI component; event can be “handled”
- **listener**: an object that waits for events and handles them
  - To handle an event, attach a **listener** to a component
  - The listener will be **notified** when the event (e.g., button click) occurs
- What design pattern is this?
Kinds of GUI Events

- Mouse: move/drag/click, mouse button press/release
- Keyboard: key press/release, sometimes with modifiers like shift/control/alt
- Touchscreen: finger tap/drag
- Joystick, drawing tablet, other device inputs
- Window resize/minimize/restore/close
- Network activity or file I/O (start, done, error)
- Timer interrupt (including animations)

Event Hierarchy

```
import java.awt.event.*

EventObject
  import java.awt.event.*
  ActionEvent
    TextEvent
    KeyEvent (Keyboard)
    MouseEvent (Mouse move/drag, etc.)
```

Event Objects

- Event object contains information about the event
  - Info about GUI component that triggered the event
  - Other information depending on the event. E.g.,
    - ActionEvent – text string from a text selection box
    - MouseEvent – mouse coordinates
  - action event: an action that has occurred on a GUI component. The most common Event type in Swing
    - Button or menu clicks
    - Check box checking/unchecking, etc.
  - Represented by class ActionEvent
  - Handled by objects that implement interface ActionListener

Events are Handled by Listeners

- Implementing a listener
  ```
  public class MyListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
      // code to handle event here
    }
  }
  ```
  JButton and other graphical components have this method:
  ```
  /** Attaches a given listener to be notified of clicks and events that occur on this component. */
  public attachActionListener(EventListener al)
  ```

EventListener Hierarchy

```
import java.awt.event.*

EventListener
  import java.awt.event.*
  AWTEventListener
  ActionListener
    has actionPerformed(…)
  TextListener
  ComponentListener
  FocusListener
  WindowListener
  KeyListener
    has keyPressed(…)
  MouseListener
    has mouseClicked(…)
```

Event Listener Hierarchy

- When an event occurs, the appropriate method specified in the interface is called
  - When an action event (e.g., a button click) occurs, actionPerformed gets called on the attached (i.e., registered) Listeners

- An event object is received as parameter to the event listener method
  ```
  actionPerformed(ActionEvent e)
  ```
**Decoding Events**

- A single button listener can handle several buttons. How to tell which is which?

- **An ActionEvent has a `getActionCommand` method** that returns (for a button) the "action command" string. Default is the **button name**. It is better to set to a specific string, which will remain the same even if UI (and button name) changes.

- E.g.: `...csci2600/handouts/ButtonDemo1.java`

---

**Aside: Nested Classes**

- Event listeners are often defined as nested classes, mostly as **inner classes** inside a GUI.

- Nested class: A class defined in another class
  - static nested class
  - non-static nested class (known as **inner classes**)

- Inner classes are hidden from other classes

- Inner objects can access the instance fields and methods of their outer object

---

**Nested Class Syntax**

```java
public class OuterClass {
    ...
    // Static nested class
    // Same as top-level class, nested for convenience
    static private class NestedClass { ...

    // Instance nested class (inner class)
    // A different class for each instance of Outer
    private class InnerClass { ...
```
Anonymous Inner Class Syntax

```java
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        doSomething();
    }
});
```

Outline of Today’s Class

- Organization of the Java Swing/AWT library
  - Components and containers
  - Layout managers
  - Graphics and drawing
- Events
  - Event objects
  - Event listeners
- Anonymous inner classes
- Interaction between UI and program threads

Program thread and UI thread

- Your program and the UI run in concurrent threads
- All UI actions happen in the UI thread
  - Including callbacks: `paintComponent, actionPerformed`
  - After event handling, you may call `repaint()` if `paintComponent` needs to run
  - Do not try to draw anything from inside the event handler!

Working in the UI Thread

- Event handlers should not do a lot of work
  - If event handler does a lot of work, interface will appear to freeze up. Why?
    - If there is a lot to do, the event handler should set a bit that the program thread will notice. Then do the heavy work back in the program thread