Java GUI Programming

*Adapted from a lecture by Vangelis Metsis

CS 3354
Spring 2017
Jill Seaman

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
- Because GUIs are neat!

Java GUI example

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.

- **SWT + JFace**
  - Mixture of native widgets and Java rendering; created for Eclipse for faster performance

- Others
  - Apache Pivot, SwingX, JavaFX, ...

- **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
Basic Elements

- Components:
  - Button / List / Checkbox / Choice / TextField / Etc.
- Containers (subclass of Component):
  - Panel / Window / Dialog / Applet / Frame / Etc.
- Menu Components
  - Menu / Menu bar / Etc.
- Layout Managers
  - BorderLayout / GridLayout / Etc.
- Events
  - MouseEvent / MouseMotionEvent / ItemEvent / Etc.
- Graphics
  - Graphics / Image / Color / Font / FontMetrics / Etc.

AWT Components, Containers, and Layout Managers

Swing/AWT inheritance hierarchy
Component fields/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

Swing window example

**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!
**JFrame**

- Frame window has *decorations*
  - title bar
  - close box
  - provided by windowing system

- Basic code to create a frame:
  ```java
  JFrame frame = new JFrame();
  frame.pack();   // Fit frame to its contents
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  frame.setVisible(true);
  ```

**More JFrame**

- `public void setDefaultCloseOperation(int op)`
  Makes the frame perform the given action when it closes.
  - Common value passed: JFrame.EXIT_ON_CLOSE
  - Other possible values:
    DO_NOTHING_ON_CLOSE
    HIDE_ON_CLOSE
    DISPOSE_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.

**Example: SimpleFrameMain**

```java
import java.awt.*;
import javax.swing.*;

public class SimpleFrameMain {
    public static void main(String[] args) {
        SimpleFrame frame = new SimpleFrame("A Window");
        frame.pack();
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}

class SimpleFrame extends JFrame {
    public SimpleFrame(String title) {
        super(title);
        setSize(300, 200);
    }
}
```

**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.
  - E.g.: `contentPane.setLayout(new BorderLayout(0, 0));`
JFrame as container

A JFrame is a container. Containers have these methods:

- `public void add(Component comp)`
- `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.

Graphics and drawing

- 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.

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.
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)

---

Rules for painting

- Always override `paintComponent(g)` if you want to draw on a component.
- Always call `super.paintComponent(g)` first.
- **NEVER** 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, 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.

---

Event-driven programming

A style of coding where a program’s overall flow of execution is dictated by events.

- The program loads, then waits for user input events.
- As each event occurs, the program runs particular code to respond.
- The overall flow of what code is executed is determined by the series of events that occur.
- Contrast with application- or algorithm-driven control where program expects input data in a pre-determined order and timing
  - Typical of large non-GUI applications like web crawling, payroll, batch simulation

---

Event-driven programming

- The main body of the program is an event loop. Abstractly:

  ```java
  do {
    e = getNextEvent();
    process event e;
  } while (e != quit);
  ```

---


d0 {  
ed = getNextEvent();
P    process event e;
} while (e != quit);

1. User interacts with page
2. An "event" occurs
3. A piece of code runs in response
4. The page's appearance is updated/modified in some way as a result

---
Graphical events

- **event**: An object that represents a user's interaction with a GUI component; can be "handled" to create interactive components.

- **listener**: An object that waits for events and responds to them.
  - To handle an event, attach a listener to a component.
  - The listener will be notified when the event occurs (e.g. button click).

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)

Action events

- **action event**: An action that has occurred on a GUI component.
  - The most common, general event type in Swing. Caused by:
    - button or menu clicks,
    - check box checking / unchecking,
    - pressing Enter in a text field, ...
  - Represented by a class named `ActionEvent`
  - Handled by objects that implement interface `ActionListener`

Implementing a Listener (Observer)

```java
public class MyClass implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        code to handle the event;
    }
}
```

JButton and other graphical components have this method:

```java
/** Attaches the given listener to be notified of clicks and events that occur on this component. */
public void addActionListener(ActionListener al)
```

e.g.

```java
button.addActionListener(new MyClass());
```
Example: button

- Create a JButton and add it to a window
  - public JButton(String text) Creates a new button with the given string as its text.
  - public String getText() Returns the text showing on the button.
  - public void setText(String text) Sets button's text to be the given string.

- Create an object that implements ActionListener (containing an actionPerformed method)

- Add the listener object to the button's listeners

```java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ButtonDemo1 {
    public class ButtonListener implements ActionListener {
        private int nEvents = 0;

        public void actionPerformed(ActionEvent e) {
            nEvents++;
            System.out.println(e.getActionCommand() + " " + nEvents);
        }
    }

    public static void main(String[] args) {
        JFrame frame = new JFrame("Button Demo");
        frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);

        JButton button = new JButton("Hit me");
        button.setActionCommand("OUCH!");
        button.addActionListener(new ButtonListener());

        // Add button to the window and make it visible
        frame.add(button); frame.pack(); frame.setVisible(true);
    }
}
```

Program thread and UI thread

- The program and user interface run in concurrent threads.
- All UI actions happen in the UI thread – even when they execute callbacks to code like ActionListener, etc. defined in your program.
- Any updates to the user interface must happen on the event dispatch thread.
- Event handlers usually should not do a lot of work.
- If the event handler does a lot of computing, the user interface will appear to freeze up.

```java
import java.awt.*; // basic awt classes
import java.awt.event.*; // event classes
import javax.swing.*; // swing classes

public class ButtonDemo2 {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Button Demo");
        frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);

        JButton button = new JButton("Hit me");
        button.setActionCommand("OUCH!");

        // Create and register a new button listener to handle clicks
        button.addActionListener(new ActionListener() {
            int nEvents = 0; // number of events handled
            public void actionPerformed(ActionEvent e) {
                nEvents++;
                System.out.println(e.getActionCommand() + " " + nEvents);
            }
        });

        // Add button to the window and make it visible
        frame.add(button); frame.pack(); frame.setVisible(true);
    }
}
```
Program thread and UI thread

- If there's lots to do, the event handler should start a new thread or set a bit that the program thread will notice.
- Do the heavy work back in the program thread.
- When the heavy work finishes, the UI is notified to update the view.

Suppose we have a button that launches a series of database queries. We dutifully start up a new thread so that our queries won't block the user interface:

```java
JButton b = new JButton("Run query");
b.addActionListener(new ActionListener() {
    Thread queryThread = new Thread() {
        public void run() {
            runQueries();
        }
    }
    queryThread.start();
});
```

Application startup code

- There's one place where it's very easy to forget that we need SwingUtilities.invokeLater(), and that's on application startup.
- Our applications main() method will always be called by the main program thread that the VM starts up for us.
- If we have code to update the GUI there, it may interfere with the UI thread.
- The code that initializes our GUI must also take place in an invokeLater().

```java
public class MyApplication extends JFrame {
    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            MyApplication app = new MyApplication();
            app.setVisible(true);
        });
    }
    private MyApplication() {
        // create UI here: add buttons, actions etc
    }
}
```

But now, from our query thread, we want to update a progress bar or some other component showing the current progress to the user.

```
private void updateProgress(final int queryNo) {
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            statusLabel.setText("Query: " + queryNo);
        }
    });
}
```

Better implementation of ButtonDemo

```java
public class ButtonDemo3 extends JFrame {
    private ButtonDemo3(String title) {
        super(title);
        setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        Component c = this;
        JButton button = new JButton("Hit me");
        button.setActionCommand("OUCH!");
        button.addActionListener(new ActionListener() {
            int nEvents = 0;
            public void actionPerformed(ActionEvent e) {
                nEvents++;
                JOptionPane.showMessageDialog(c, e.getActionCommand() + "  " + nEvents);
            }
        });
        this.add(button); this.pack(); this.setVisible(true);
    }
    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            ButtonDemo3 app = new ButtonDemo3("Button Demo");
            app.setVisible(true);
        });
    }
}
```