CSE 331

Event-driven Programming and Graphical User Interfaces (GUIs) with Swing/AWT

slides created by Marty Stepp
http://www.cs.washington.edu/331/
Why learn GUls?

- 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:
  - model-view separation
  - design patterns
  - refactoring vs. reimplementing an ailing API
- Because GUls are neat!

*Caution*: There is way more here than you can memorize.
- Part of learning a large API is "letting go."
- You won't memorize it all; you will look things up as you need them.
- But you can learn the fundamental concepts and general ideas.
Java GUI History

• **Abstract Windowing Toolkit (AWT):** Sun's initial effort to create a set of cross-platform GUI classes. *(JDK 1.0 - 1.1)*
  - Maps general Java code to each operating system's real GUI system.
  - *Problems:* Limited to lowest common denominator; clunky to use.

• **Swing:** A newer GUI library written from the ground up that allows much more powerful graphics and GUI construction. *(JDK 1.2+)*
  - Paints GUI controls itself pixel-by-pixel rather than handing off to OS.
  - *Benefits:* Features; compatibility; OO design.
  - *Problem:* Both exist in Java now; easy to get them mixed up; still have to use both in various places.
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 logical grouping for storing components.
  - examples: panel, box
## Components

<table>
<thead>
<tr>
<th>JComponent</th>
<th>JComponent</th>
<th>JComponent</th>
<th>JComponent</th>
</tr>
</thead>
<tbody>
<tr>
<td>JButton</td>
<td>JCheckBox</td>
<td>JRadioButton</td>
<td>JLabel</td>
</tr>
<tr>
<td>OK</td>
<td>Check</td>
<td>Radio</td>
<td>Image and Text</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JTextField</td>
<td>JSlider</td>
<td>JToolBar</td>
<td>Text-Only Label</td>
</tr>
<tr>
<td>Years: 30</td>
<td>Frames Per Second</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JComboBox</td>
<td>JList</td>
<td>JMenuBar, JMenu, JMenuItem</td>
<td></td>
</tr>
<tr>
<td>Pig</td>
<td>January</td>
<td>A Menu</td>
<td>A text-only menu item</td>
</tr>
<tr>
<td>Bird</td>
<td>February</td>
<td>Another Menu</td>
<td>Both text and icon</td>
</tr>
<tr>
<td>Cat</td>
<td>March</td>
<td></td>
<td>A radio button menu item</td>
</tr>
<tr>
<td>Dog</td>
<td>April</td>
<td></td>
<td>A check box menu item</td>
</tr>
<tr>
<td>Rabbit</td>
<td></td>
<td></td>
<td>A submenu</td>
</tr>
<tr>
<td>Pig</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JColorChooser</td>
<td>JFileChooser</td>
<td>JTable</td>
<td>JTree</td>
</tr>
<tr>
<td>Swatches</td>
<td>Open</td>
<td></td>
<td>Music</td>
</tr>
<tr>
<td>HSB</td>
<td>Look in: C:</td>
<td></td>
<td>Classical</td>
</tr>
<tr>
<td>RGB</td>
<td></td>
<td></td>
<td>Beethoven</td>
</tr>
<tr>
<td></td>
<td>emacslib</td>
<td></td>
<td>Brahms</td>
</tr>
<tr>
<td></td>
<td>host-news</td>
<td></td>
<td>Mozart</td>
</tr>
<tr>
<td></td>
<td>java</td>
<td></td>
<td>Jazz</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Rock</td>
</tr>
</tbody>
</table>
Swing inheritance hierarchy

- Component  (AWT)
  - Window
    - Frame
      - JFrame  (Swing)
      - JDialog
  - Container
    - JComponent  (Swing)
      - JButton
      - JComboBox
      - JMenuBar
      - JPopupMenu
      - JScrollPane
      - JSplitPane
      - JToolBar
      - JTextField
      - JColorChooser
      - JFileChooser
      - JLabel
      - JOptionPane
      - JList
      - JComboBox
      - JFileChooser
      - JPanel
      - JProgress_Bar
      - JScrollPane
      - JSpinner
      - JScrollPane
      - JTable
      - JTextField
      - ...
## Component properties

- Each has a **get** (or **is**) accessor and a **set** modifier method.
- **examples:** `getColor`, `setFont`, `setEnabled`, `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>
JFrame

*a graphical window to hold other components*

- `public JFrame()`
  `public JFrame(String title)`
  Creates a frame with an optional title.
  
  - Call `setVisible(true)` to make a frame appear on the screen after creating it.

- `public void add(Component comp)`
  Places the given component or container inside the frame.
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.
JButton

a clickable region for causing actions to occur

• 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.
import java.awt.*; // Where is the other button?
import javax.swing.*;

public class GuiExample1 {
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(new Dimension(300, 100));
        frame.setTitle("A frame");

        JButton button1 = new JButton();
        button1.setText("I'm a button.");
        button1.setBackground(Color.BLUE);
        frame.add(button1);

        JButton button2 = new JButton();
        button2.setText("Click me!");
        button2.setBackground(Color.RED);
        frame.add(button2);

        frame.setVisible(true);
    }
}
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."
Containers and layout

- 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.
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.

*Buttons at preferred size:*

*Not preferred size:*
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.

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

▪ This is the default layout for a JFrame.
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.
Event Listeners
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).

![Diagram of event handling](image)
Event-driven programming

- **event-driven programming**: A style of coding where a program's overall flow of execution is dictated by events.
  - Rather than a central "main" method that drives execution, the program loads and 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, not a pre-determined order.
Event hierarchy

- EventObject
  - AWTEvent (AWT)
    - ActionEvent
    - TextEvent
    - ComponentEvent
      - FocusEvent
      - WindowEvent
      - InputEvent
        - KeyEvent
        - MouseEvent
  - WindowEvent
  - InputEvent
  - KeyEvent
  - MouseEvent

- EventListener
  - AWTEventListener
  - ActionListener
  - TextListener
  - ComponentListener
  - FocusListener
  - WindowListener
  - KeyListener
  - MouseListener

import java.awt.event.*;
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`
public class name implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        code to handle the event;
    }
}

- JButton and other graphical components have this method:
  - public void addActionListener(ActionListener al)
    Attaches the given listener to be notified of clicks and events that occur on this component.
Nested classes

• **nested class**: A class defined inside of another class.

• **Usefulness:**
  - Nested classes are hidden from other classes (encapsulated).
  - Nested objects can access/modify the fields of their outer object.

• Event listeners are often defined as nested classes inside a GUI.
Nested class syntax

// enclosing outer class
public class name {
  ...

  // nested inner class
  private class name {
    ...
  }
}

- Only the outer class can see the nested class or make objects of it.
- Each nested object is associated with the outer object that created it, so it can access/modify that outer object's methods/fields.
  - If necessary, can refer to outer object as `OuterClassName.this`
Static inner classes

// enclosing outer class
public class name {
    ...

    // non-nested static inner class
    public static class name {
        ...
    }
}

- Static inner classes are not associated with a particular outer object.
- They cannot see the fields of the enclosing class.
- Usefulness: Clients can refer to and instantiate static inner classes:
  Outer.Inner name = new Outer.Inner(params);
public class MyGUI {
    private JFrame frame;
    private JButton stutter;
    private JTextField textfield;

    public MyGUI() {
        ...
        stutter.addActionListener(new StutterListener());
    }
    ...

    // When button is clicked, doubles the field's text.
    private class StutterListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            String text = textfield.getText();
            textfield.setText(text + text);
        }
    }
}