Building a Swing GUI

- Consider the following “SwingApplication” (courtesy of Sun’s Java Swing Tutorial)
- Every app defines a hierarchy of components
  - “component” = “widget”

Creating components

- Just call the constructors!

  ```
  frame = new JFrame (...);
  button = new JButton (...);
  label = new JLabel (...);
  panel = new JPanel (...);
  ...
  ```

- All we’ve done is create the data structures
- Still need to:
  - add components to container
  - lay out container

Adding components

- Add components to top-level container... typically, to `content pane`

  ```
  frame.getContentPane().add (button);
  ```
- Add components to intermediate containers

  ```
  JPanel panel = new JPanel (...);
  panel.add (button); // and more...
  frame.getContentPane().add (panel);
  ```
Laying out components

- We could just specify absolute positioning — i.e., exact window coordinates

- Why is this not (typically) a good idea?

Laying out components

- Problems with absolute positioning (cont.)
  - components designed for a specific look-and-feel or font size
  - components designed for a specific language

- Solution: Layout managers!
  - layout manager = algorithm for positioning and sizing of GUI components
  - Swing’s LayoutManager interface
    - each Component has:
      - public Dimension getPreferredSize(); // desired size
      - public Dimension getMinimumSize(); // smallest desired size
      - public Dimension getMaximumSize(); // largest desired size
  - managers use this info to compute layouts
  - Caveat: “Layout managers can respect or ignore as much or as little of this information as they see fit” (!)
## Layout managers

### BorderLayout (the default)

```java
JPanel pane = new JPanel();
pane.setLayout(new BorderLayout());  // not really needed
...
pane.add (buttonNorth, BorderLayout.NORTH);
pane.add (buttonCenter, BorderLayout.CENTER);
...
```

### BorderLayout (cont.)

- can’t add twice in the same place

```java
contentPane.add (buttonNorth1, BorderLayout.NORTH);
contentPane.add (buttonNorth2, BorderLayout.NORTH);
// this second add() puts the second button on top of the first!
...
```

- can add spacing with the constructor

```java
JPanel pane = new JPanel();
pane.setLayout (new BorderLayout (5, 20));  // xGap, yGap
...
```

###BoxLayout: across or up/down

```java
private void addButton(String text, Container container) {
    JButton button = new JButton(text);
    button.setAlignmentX(Component.CENTER_ALIGNMENT);
    container.add(button);
}

public BoxWindow() {
    Container contentPane = getContentPane();
    contentPane.setLayout(new BoxLayout(contentPane, BoxLayout.Y_AXIS));
    addButton("Button 1", contentPane);
    addButton("2", contentPane);
    addButton("Button 3", contentPane);
    addButton("Long-Named Button 4", contentPane);
    addButton("Button 5", contentPane);
    ...
}
```

### GridLayout: equal-sized grid of panels

```java
Container contentPane = getContentPane();
contentPane.setLayout(new GridLayout(0, 2));  // grid layout with 2 columns; doesn’t specify number of rows!
contentPane.add(new JButton("Button 1"));
contentPane.add(new JButton("2"));
contentPane.add(new JButton("Button 3"));
contentPane.add(new JButton("Long-Named Button 4"));
contentPane.add(new JButton("Button 5"));
```
Layout managers

• **FlowLayout**: flows the rows, you know!

```
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
contentPane.add(new JButton("Button 1"));
contentPane.add(new JButton("Button 2"));
contentPane.add(new JButton("Long-Named Button 4"));
contentPane.add(new JButton("Button 5"));
```

• **GridBagLayout**: more flexible grid
  - useful! — check out the tutorial online

```
Container contentPane = getContentPane();
contentPane.setLayout(new GridBagLayout());
contentPane.add(new JButton("Button 1"));
contentPane.add(new JButton("Button 2"));
contentPane.add(new JButton("Button 3"));
contentPane.add(new JButton("Long-Named Button 4"));
contentPane.add(new JButton("Button 5"));
```

• **CardLayout**: changing components

```
Container contentPane = getContentPane();
contentPane.setLayout(new CardLayout());
contentPane.add(new JButton("Button 1"));
contentPane.add(new JButton("Button 2"));
contentPane.add(new JButton("Button 3"));
contentPane.add(new JButton("Long-Named Button 4"));
contentPane.add(new JButton("Button 5"));
```

Layout extras

• **Spacing components out**
  - create space with rigid boxes
  - create space with “glue” (bad name!)

```
pane.add(Box.createRigidArea(new Dimension(0,5)));
```

```
without horizontal glue
with horizontal glue
```

```
container.add (firstComponent);
container.add (Box.createHorizontalGlue());
container.add (secondComponent);
```

• **Absolute positioning**

```
Container contentPane = getContentPane();
contentPane.setLayout(null);
b1 = new JButton("one");
contentPane.add(b1);
b2 = new JButton("two");
contentPane.add(b2);
b3 = new JButton("three");
contentPane.add(b3);
Insets insets = contentPane.getInsets();
b1.setBounds(25 + insets.left, 5 + insets.top, 75, 20);
b2.setBounds(55 + insets.left, 35 + insets.top, 75, 20);
b3.setBounds(150 + insets.left, 15 + insets.top, 75, 30);
```
Example: SwingApplication

• High-level view

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

public class SwingApplication {
    public Component createComponents() {
        ...
    }
    public static void main(String[] args) {
        ...
    }
}
```

Example: SwingApplication

• main()

```java
public Component createComponents() {
    ...
    JFrame frame = new JFrame("SwingApplication");
    SwingApplication app = new SwingApplication();
    Component contents = app.createComponents();
    frame.getContentPane().add(contents, BorderLayout.CENTER);
    frame.pack();
    frame.setVisible(true);
}
```

Example: SwingApplication

• createComponents()

```java
public Component createComponents() {
    final JLabel label = new JLabel(labelPrefix + "0 ");
    JButton button = new JButton("I'm a Swing button!");
    label.addActionListener(button);
    JPanel pane = new JPanel();
    pane.setBorder(BorderFactory.createEmptyBorder(30,30,10,30));
    pane.setLayout(new GridLayout(0, 1));
    pane.add(button);
    pane.add(label);
    return pane;
}
```

Class exercise: MyLayout

• Consider this window and what it looks like when resized...
• How would you write the Swing code?
Example: MyLayout2

- How can we structure this layout?

<table>
<thead>
<tr>
<th>North</th>
<th>Center</th>
<th>South</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up</td>
<td></td>
<td>Down</td>
</tr>
<tr>
<td>Go</td>
<td>with</td>
<td>the</td>
</tr>
<tr>
<td>flow</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Model-View-Controller Architecture

- What defines a component?
  - In Swing (and similar frameworks), a component has three crucial elements:
    - Model: what data is associated with component
    - View: how the component is displayed on-screen
    - Controller: how the component responds to user interaction / events

Example: the scrollbar

<table>
<thead>
<tr>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>min = 0</td>
</tr>
<tr>
<td>max = 255</td>
</tr>
<tr>
<td>value = 87</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>View</th>
</tr>
</thead>
<tbody>
<tr>
<td>Controller</td>
</tr>
<tr>
<td>mouse click on end</td>
</tr>
<tr>
<td>mouse click on bar</td>
</tr>
<tr>
<td>mouse drag on scroller</td>
</tr>
</tbody>
</table>
Model-View-Controller Architecture

- What are the MVC elements for a...
  - Menu?
  - Toolbar?

Model-View-Controller Architecture

- All three elements are interdependent!
  - Why is this breakdown useful?
    - multiple components can be tied to same model
    - models can have different "look & feel"s

Handling GUI events

- The user does something – an event occurs.
  - How do you handle the event?
  - Wait, what events are we talking about?

  User clicks a button, presses Return while typing in a text field, or chooses a menu item
  User closes a frame (main window)
  User presses a mouse button while the cursor is over a component

User moves the mouse over a component
Component becomes visible
Component gets the keyboard focus
Table or list selection changes

Programming the GUI

- Sequential/procedural programming
  - your program is (almost) always in control

  yourFoo() → yourSubFoo() → systemFoo()
  yourFoo() ← yourSubFoo() ← systemFoo()

  - for user input, the program dictates when/how, and the user responds

  > ls */*.java_
Programming the GUI

- **Sequential/procedural programming (cont.)**
  - the good points
    - easy to think about: one event, then the next, ...
    - easy to design and represent with well-known models
    - easy to program with today’s programming languages
  - the big bad point
    - program dictates when/how user must respond

Fine for the programmer...
Typically not great for the user

---

Ways to handle events

- **“Macintosh-style” event handling**
  - decide what the event is
  - figure out what window it goes to
  - handle the event

```c
void handleEvent (event)
{
    switch (event->type)
    {
    case MOUSE_CLICK:
        window = event->window;
        << deal with click >>
        case ...:
    }
}
```

NB: Pseudocode!
Ways to handle events

- Object-oriented event handling
  - the way Java does it, naturally
  - based on the OOP component hierarchy
    - define event-handling methods for components
    - components can (of course) inherit these methods from parent components
  - also based on the Model-View-Control Architecture
  - now let’s look at this in detail...

Swing sources and listeners

- Event sources generate events
- Event listeners respond to them

<table>
<thead>
<tr>
<th>Event Source</th>
<th>Event Listener</th>
</tr>
</thead>
<tbody>
<tr>
<td>User clicks a button, presses Return while typing in a text field, or chooses a menu item</td>
<td>ActionListener</td>
</tr>
<tr>
<td>User closes a frame (main window)</td>
<td>WindowListener</td>
</tr>
<tr>
<td>User presses a mouse button while the cursor is over a component</td>
<td>MouseListener</td>
</tr>
<tr>
<td>User moves the mouse over a component</td>
<td>MouseMotionListener</td>
</tr>
</tbody>
</table>

ActionEvent

- A class with three methods:

```java
String getActionCommand ();
int getModifiers ();
String paramString ();
```

- We won’t use these methods at the moment, but the keep the class in mind!

 ActionListener

- An interface with a single method:

```java
public interface ActionListener {
    void actionPerformed (ActionEvent e);
}
```

- We implement the interface as follows:

```java
public class MyClassThatListens ... implements ActionListener {
    ... void actionPerformed (ActionEvent e) { ... }
    ...
}
```

- Review: How is an interface different from a class?
The 3-step program to handling events

1. Code that implements the listener class

   ```java
   public class MyClass implements ActionListener
   {
   ...
   }
   ```

2. Code that implements the listener methods

   ```java
   public void actionPerformed(ActionEvent e) {
   ...
   }
   ```

3. Code that registers the listener to a source

   ```java
   component.addActionListener(instanceOfMyClass);
   ```

Example: ButtonTest

- Beep when the user clicks the button

   ```java
   import javax.swing.*;
   import java.awt.*;
   import java.awt.event.*;
   public class ButtonTest
   {
      public static class MyActionListener implements ActionListener
      {
         public void actionPerformed(ActionEvent e)
         {
            Toolkit.getDefaultToolkit().beep();
         }
      }
      public static void main (String[] args)
      {
         JFrame frame = new JFrame("Program");
         JButton button = new JButton("Click Me");
         button.addActionListener(new MyActionListener());
         ...
      }
   }
   ```

KeyEvent

- Another event class, like ActionListener
- Has many methods, including:

   ```java
   char getKeyChar ();  // returns character typed
   int getModifiers (); // returns typed modifiers (shift, ctrl, etc.)
   ```

KeyListener

- Another interface, like ActionListener
- Includes several methods:

   ```java
   public interface KeyListener
   {
   public void keyTyped(KeyEvent e);  // press+release
   public void keyPressed(KeyEvent e); // press
   public void keyReleased(KeyEvent e); // release
   }
   ```
Example: KeyTest

• **MyKeyListener class**

```java
class MyKeyListener implements KeyListener {
    public void keyTyped(KeyEvent e) {
        char c = e.getKeyChar();
        KeyTest.windowPrint(String.valueOf(c));
    }
    public void keyPressed(KeyEvent e) {}
    public void keyReleased(KeyEvent e) {}
}
```

**do we need these?**

Example: KeyTest

• **KeyTest class**

```java
class KeyTest {
    public static void main(String[] args) {
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                buttonAction();
            }
        });
    }
}
```

Example: KeyTest2

• The last example defined a full class for a new KeyListener in the usual way.
• It’s prim & proper, but gets bulky. (logorrheic, for you word buffs out there)
• A quicker way of saying the same thing

```java
button.addActionListener(new KeyListener() {
    public void keyTyped(KeyEvent e) {
        char c = e.getKeyChar();
        windowPrint(String.valueOf(c));
    }
    public void keyPressed(KeyEvent e) {}
    public void keyReleased(KeyEvent e) {}
});
```

Our old friend “ClickMe”

```java
class ClickMe {
    public static void main(String[] args) {
        initialize();
        JButton button = new JButton("Click Me!");
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                buttonAction();
            }
        });
        << create and lay out frame >>
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);}
        });
    }
}
```
WindowListener

• Another listener interface
• Includes lots o’ methods:

```java
public interface WindowListener {
    public void windowActivated(WindowEvent e);
    public void windowClosed(WindowEvent e);
    public void windowClosing(WindowEvent e);
    public void windowDeactivated(WindowEvent e);
    public void windowDeiconified(WindowEvent e);
    public void windowIconified(WindowEvent e);
    public void windowOpened(WindowEvent e);
}
```

WindowAdapter

• Implements the WindowListener interface with null (do-nothing) methods

```java
public class WindowAdapter implements WindowListener {
    public void windowActivated(WindowEvent e) {  }
    public void windowClosed(WindowEvent e) {  }
    public void windowClosing(WindowEvent e) {  }
    public void windowDeactivated(WindowEvent e) {  }
    public void windowDeiconified(WindowEvent e) {  }
    public void windowIconified(WindowEvent e) {  }
    public void windowOpened(WindowEvent e) {  }
}
```

Adapters vs. Interfaces

• The first is fine; the second is not!

```java
public class MyListener implements KeyListener, WindowListener {
    public void keyPressed(KeyEvent e) { ... }
    public void keyReleased(KeyEvent e) { ... }
    public void windowActivated(WindowEvent e) { ... }
    public void windowClosed(WindowEvent e) { ... }
}
```

```java
public class MyListener extends KeyAdapter, WindowAdapter {
    public void keyPressed(KeyEvent e) { ... }
    public void keyReleased(KeyEvent e) { ... }
    public void windowActivated(WindowEvent e) { ... }
    public void windowClosed(WindowEvent e) { ... }
}
```

Again, there is no multiple inheritance in Java!

Research Question of the Day

• How easy is it to think in terms of event-driven programming?
• Here’s an idea: Can we teach children to “program” with “events”?
  - what type of events?
  - what representation for the program?
Research Question of the Day

• Montemayor et al. (2002) have explored “programming physical environments”
• Some objects kids might use:
  - hand for “input”
  - light and sound box for “output”
  - magic wand to enable “programming”


Research Question of the Day

• How to program an “event” (e.g.):
  - put the hand and light near each other
  - using the wand, tap the hand, then the light
  - when “running”, the hand now switches the light
• What’s the technology behind this?
  - actually, not much — they use a “Wizard of Oz” approach to recording and running programs
    • wizard = experimenter watching behind one-way mirror
    • wizard notes relationships between objects, then controls light/sound when appropriate
    • kids can get frustrated with this “technology”!!

Research Question of the Day

• Some interesting results
  - kids enjoyed the “programming” as part of their storytelling
  - biggest difficulty: understanding difference between programming and “running”
    • magic wand helped to discriminate
    • but the fact that output devices reacted during programming (e.g., light came on) blurred the line
  - when frustrated, kids revert to what they know best — telling stories verbally
    • like adults, they’ll ignore technology if it’s not useful!