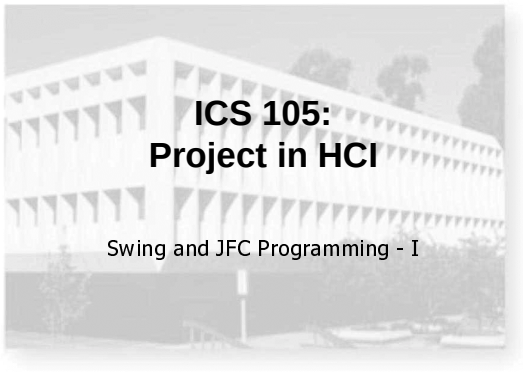


ICS 105: Project in HCI

Swing and JFC Programming - I

midterm

- not bad overall
  - a couple of questions caused consistent confusion
  - 27 and up is an A
  - 20 and up is a B
- review answers
- and now back to our regularly scheduled broadcast…

java, awt, Swing, Java2D…

- java UI delivered piece by piece
  - awt first for basic graphics and UI
  - later, Swing on top of awt
  - then Java2D
  - then Swing on Java2D
- package confusion
  - java.awt, java.awt.event
  - javax.swing, javax.swing.event, …

swing design goals

- AWT used a "peer-ed" design
  - each AWT object had a peer in the native UI system
    - AWT button proxy for a button object in Windows, Mac, etc
    - essentially, AWT is a Java *binding* for the native system
  - this is cumbersome and complex
    - each window system has its own quirks, but Java needs to be portable across platforms
    - little control over the actual UI behaviour

swing design goals

- swing does as much as possible itself
  - completely portable
    - no longer lowest common denominator
    - implementation simpler, too
  - complete control
    - can add features over the underlying system
      - e.g. antialiasing throughout the interface
    - look and feel

frames, panels and panes

- every app is rooted in a top-level window
  - in swing, this is a "frame"
  - class JFrame (everything begins with J in swing)
  - frames correspond to windows
- frames have multiple "panes"
  - normally, we're concerned with the ContentPane
  - we can add further nested panes
    - e.g. JTabbedPane for tabbed windows

frames, panels and panes

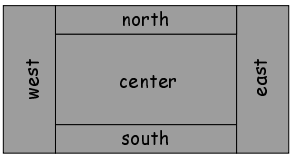
- panels
  - most user interface objects live in panels (JPanel)
  - panels are the basic unit of layout
- layout
  - delegation model for layout
    - panel itself doesn't deal with layout
    - each panel has a LayoutManager object
      - panel.setLayout(…)
    - swing provides multiple LayoutManager classes
      - each class implements a different layout policy
      - which class you instantiate determines panel layout

panels and layout

- FlowLayout
  - most primitive layout
    - basically stops things from being on top of each other
    - lays objects out left to right, top to bottom

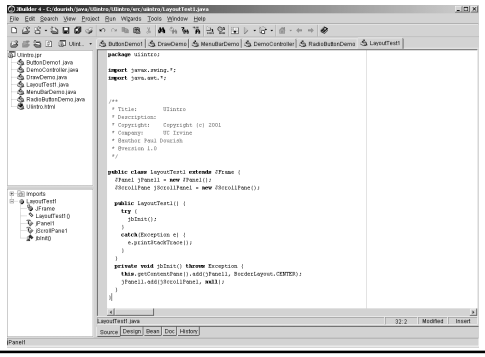
panels and layout

- BorderLayout



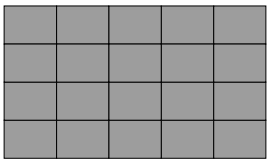
  - panel.add(Component, BorderLayout.NORTH)
  - you don't need to have all these components
    - use BorderLayout to associate some objects with borders of the application

panels and layout



panels and layout

- GridLayout



- GridLayout(int rows, int cols)
  - objects added top to bottom, left to right
  - can't span cells

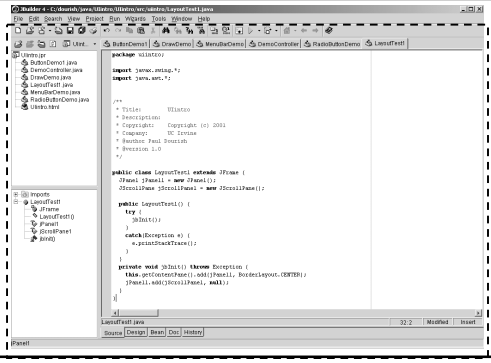
panels and layout

- GridBagLayout
  - like GridLayout, but more flexible
  - can span cells horizontally and vertically
  - very powerful
    - but therefore very complex
    - horrendous programming interface
      - often best left to GUI builders and auto-programming

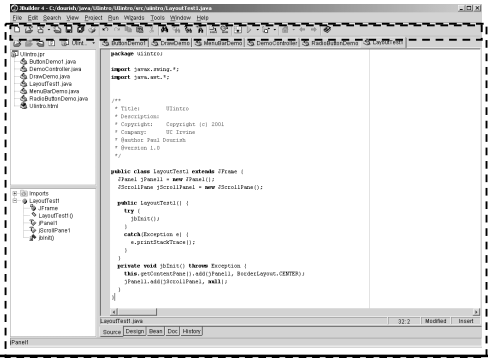
frames, panels and panes

- typically use multiple nested panels
  - panels capture different areas of functionality
  - different panels may have different layouts
    - different policies
    - different navigation mechanisms

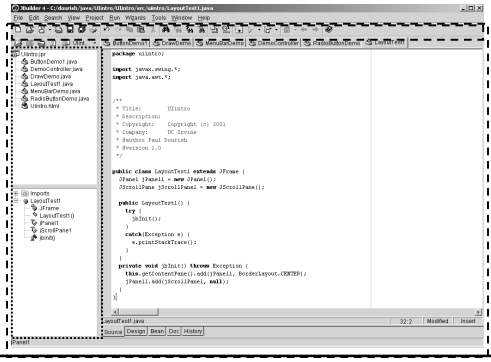
frames, panes and panels



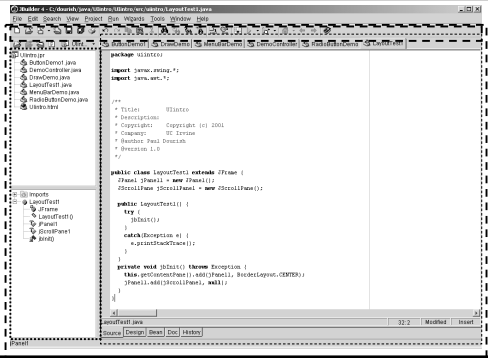
frames, panes and panels



frames, panes and panels

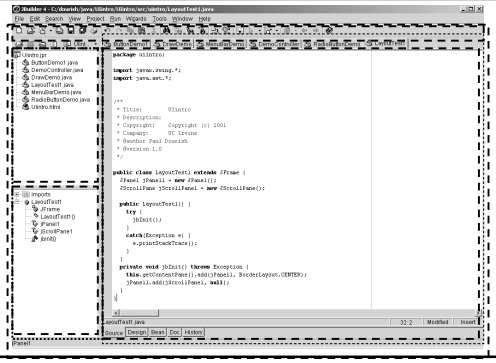


frames, panes and panels



frames, panes and panels

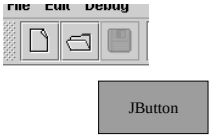
frames, panes and panels



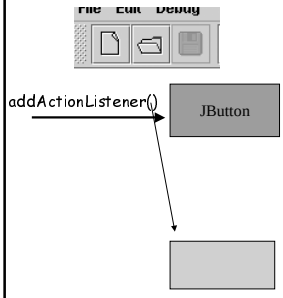
swing event model

- event-based programming model
  - user interface actions generate events
  - events delivered to objects that express interest
- the swing approach
  - every object has a set of listeners
    - different listeners for different sorts of events
  - listeners are objects interested in events
    - note – this is OBJECT BASED
      - listeners are objects
      - listeners are associated with particular objects

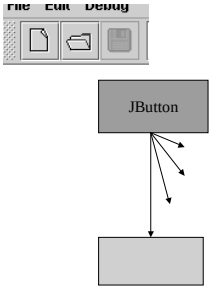
swing event model



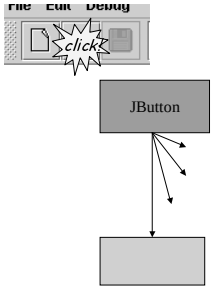
swing event model



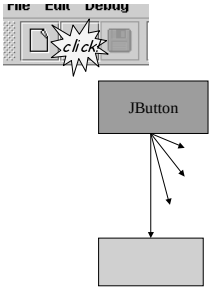
swing event model



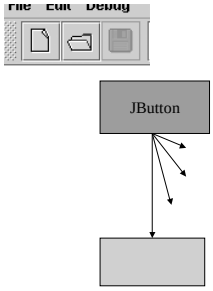
swing event model



swing event model



swing event model



actionPerformed(ActionEvent)

swing event model

- listeners
  - three features
    - the listener need to declare the right interfaces
      - each type of listener has an associated interface
        - » e.g. ActionListener, MouseMotionListener
      - java.awt.event.*;
    - the listener needs to be attached
      - via addXXXListener()
    - the listener needs to handle the event
      - implement the methods specified in the interface
        - » public void actionPerformed(ActionEvent)

swing event model

- event classes
  - ActionEvent
    - actionPerformed – e.g. button pressed, item selected
  - MouseEvent
    - mouse buttons or mouse movement
      - entered or exited, pressed, clicked, moved, dragged…
  - KeyEvent
  - WindowEvent
  - ContainerEvent
  - and others…

swing event model

- everything is object-based
  - events are objects
    - Event class describes generic event
    - each event object contains specific details
      - event.getSource() is the object that generated it
      - MouseEvent.getX() and MouseEvent.getY() for locations
  - listeners are objects
    - listeners can retain state to collect history
  - sources are objects
    - that is, you attach listeners to specific objects
      - *this* button or *that* button but not *all* buttons

swing idioms

- adapters
  - sometimes you only need a subset of events
    - e.g. MouseListener defines five methods
      - mouseClicked, mouseEntered, mouseExited, mousePressed, mouseReleased
    - often not interested in all of them
      - care about clicks but not presses, or entry/exit
  - for each XXXListener, there's an XXXAdapter
    - null methods for all events
    - inheriting from the adapter means you only have to supply the methods you're interested in

swing idioms

- listeners often use anonymous inner classes
  - inner classes are defined inside other classes
  - anonymous inner classes are
    - unnamed
    - defined in-line

```
foo.addActionListener(listenerObject)
```

swing idioms

- listeners often use anonymous inner classes
  - inner classes are defined inside other classes
  - anonymous inner classes are
    - unnamed
    - defined in-line

```
foo.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent ae) {
    System.out.println("The action was performed!");
  }
});
```

swing idioms

- listeners often use anonymous inner classes
  - inner classes are defined inside other classes
  - anonymous inner classes are
    - unnamed
    - defined in-line
  - saves a lot of overhead
    - defining a separate class for just a single method
    - listeners are often just "glue" and forward activation
    - pollute the namespace

basic drawing

- drawing isn't a Swing function
  - provides UI objects, drawing is lower level
  - still necessary, though
    - not everything on the screen is a Swing component
    - creating new objects with new visual features
- the Graphics object
  - this.getGraphics() returns an instance of Graphics
  - Graphics supports most simple drawing operations
    - drawLine, drawRectangle, drawRoundRect, drawText…

drawing example

- simple drawing application
  - open a window
  - listen for mouse events
    - mouse down – start of drawing
    - mouse dragged – draw lines