Why lightweight?

- **1.1 AWT, widgets have window-system specific peers**
  - Button defers operation to ButtonPeer, Choice defers operation to ChoicePeer
    - this allows window-specific functionality and native look-and-feel
    - each peer is “heavyweight” state, and application/applet will not work until and unless peers have been allocated and are ready --- the function `addNotify()` is used to notify peers, if you override it, call `super.addNotify()`
  - Peers are good and bad
    - functionality for free (cut and paste)
    - limited to least common denominator across platforms
- **Possible to create lightweight/no peer component in 1.1**
  - widgets in 1.2/Swing set are lightweight
How does drawing work? Rendering lightweights

- **Drawing is done in a Graphics object**
  - all Components have associated Graphics object
    - g = getGraphics()
    - g.setColor(...);
    - g.dispose();

- **Using a clipping region can make drawing faster**
  - outside of clip nothing is drawn
  - clipping regions can be combined: union/intersection
  - no real need to use clips, but lightweights clip

Clipping region, drawing restricted here, what’s the coordinate system?
Relative to clip or to graphics?
If clip is (80,40,50,25) what does g.fillRect(10,10,20,20) do?
Drawing Lightweight Components

- The component is drawn in a clip region of its container’s Graphic object
  - coordinates are relative to the clip: (0,0) is upper left
  - use coordinates of lightweight/sprite to set the component’s location or bounding rectangle
    - see `Component.setLocation()`
    - see `Component.getBounds()`
    - see `Component.getPreferredSize()`

- Container’s paint method must call `super.paint(g)` to have lightweights rendered
  - super/Container tracks all child widgets, and draws them all by calling their `paint(g)` functions (what Z-order?)

- see `LightWeightBouncy.java` example
Design for Scooter, Design for re-use

● What is the right thing to do for a queue class that dequeues the first “good” item, not necessarily the first item?
  ➤ Do you design/implement a class? A re-usable class?
  ➤ What are the tradeoffs in use and re-use?

● What Java data structures could be used for the underlying storage?
  ➤ See the PredQueue interface for how the class behaves
  ➤ use Vector, Array, List (1.2), build-your-own, differences?
  ➤ What criteria used in designing implementation?

● What other possibilities are there?
  ➤ Queue of queues?
  ➤ Something else?
DOOM ideas apropos Scooter

- One idea: write as much test code as production code
  - all Java classes should have a main that can be used to test the class
  - differentiate between testing classes, packages, project
- Iterate over a working program, grow the software
  - iterations are incremental, build on code, analysis, and use-cases (how the classes/program is used/behaves)
  - when you add functionality you must try to throw away code! (Martin Fowler, UML distilled, throw 10% away)
- With each iteration, revisit the design, replan, rethink
  - don’t slip an iteration, slip what functions in the iteration
  - test, test, test, integrate
Refactoring (see UML Distilled, Martin Fowler)

- **Software Entropy**
  - program starts off in a well-design state, as more functionality is added the program loses structure
  - scale problems, sometimes small designs/programs don’t scale appropriately
  - don’t build on a structure that wasn’t meant to be built on, refactor the program

- **Redesign**
  - redesign/rework is problematic and painful: the (small) program works
  - reduce pain of redesign, refactor the program

- **Small steps, rename variables, factor out code, make new classes, but DO NOT ADD FUNCTIONALITY**