Programming Languages and Techniques (CIS1200)

Lecture 36

Swing IV: Adapters, Paint revisited, Design Patterns

Advanced Java

Chapter 31

Announcements

- Final Program Due: (92 points)

  Monday, December 9th at 11:59pm

  - Submit zipfile online, submission *only* checks if your code compiles
  - IntelliJ is **strongly recommended** for this project
  - You may distribute your game (after the deadline) if you do not use any of our code
- Grade based on demo with your TA during/after reading days
  - Grading rubric on the assignment website
  - Recommendation: don't be too ambitious.
- *NO LATE SUBMISSIONS PERMITTED*

CIS 1200 Final Exam

- Tuesday, December 17th 12:00-2:00 PM
  - Meyerson Hall B1 — Last Names A – M
  - Fagin Auditorium — Last Names N – Z

- Students who need accommodations should schedule their exams (ASAP) through the Weingarten Center

- Review Session / Mock exam
  - 2 hour mock exam followed by 2 hour review
  - (The review session will be recorded)
  - Location and Time – TBA
  - Look for details on Ed

Exam Preparation

- *Comprehensive* exam covering the entire course:
  - *Ideas* from OCaml material (but no need to write OCaml)
  - All Java material
    (emphasizing material since midterm 2)
  - All course content
    - *except:* Bonus Lecture *(Code is Data) and*
      Guest Lecture *(Jane Street)*
    - *Only simple/shallow questions about Wednesday's lecture*

- Closed book, but…
  - You may use one letter-sized, two-sided, *handwritten* sheet of notes during the exam.

We're almost done

- **Today**: Swing IV: Adapters, Paint revisited, Design Patterns
- **Wednesday**: Advanced Java Topics
- **Friday**: Semester Recap

- **Monday**: OCaml at Jane Street

36: How far along are you on the Game?

0

I haven't started yet

0%

I have the basic design implemented

0%

I'm about halfway done, I think

0%

I'm nearly finished

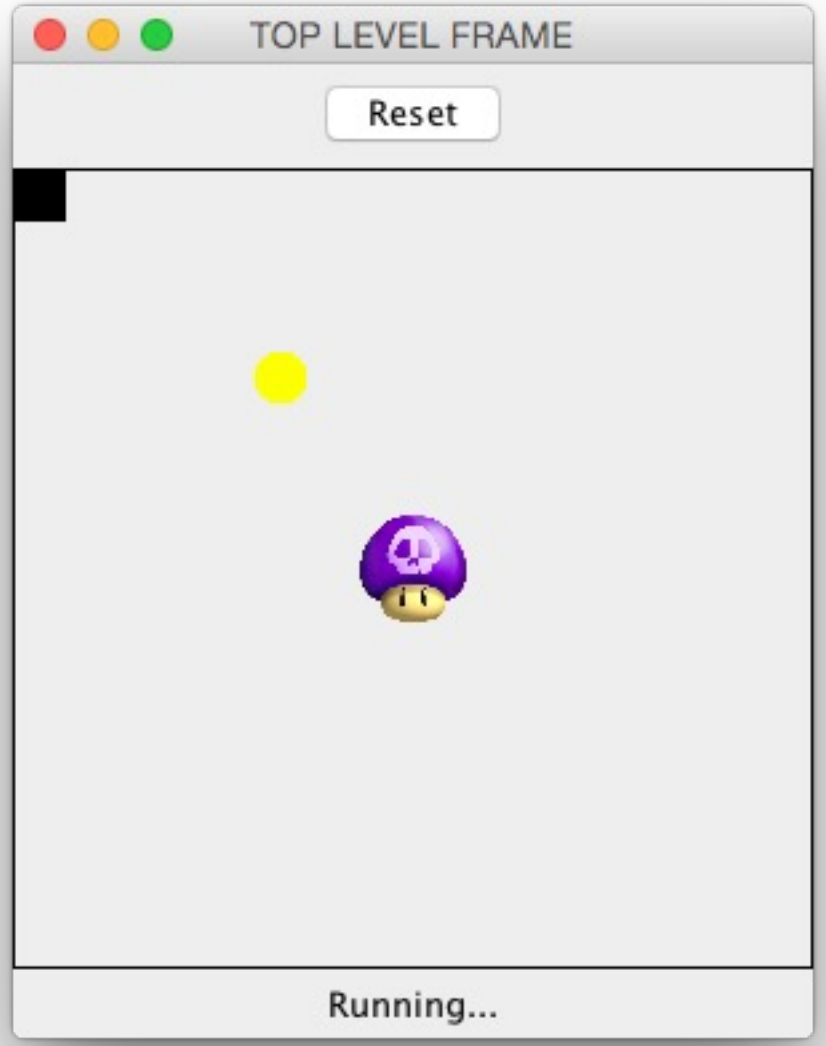
0%

I'm done!

0%

Mushroom of Doom

How do we put Swing components together to make a complete game?


TOP LEVEL FRAME
Reset
Running...

Adapters

MouseAdapter

KeyAdapter

Two interfaces for mouse listeners

```
interface MouseListener extends EventListener {
   public void mouseClicked(MouseEvent e);
   public void mouseEntered(MouseEvent e);
   public void mouseExited(MouseEvent e);
   public void mousePressed(MouseEvent e);
   public void mouseReleased(MouseEvent e);
}
```

```
interface MouseMotionListener extends EventListener {
    public void mouseDragged(MouseEvent e);

    public void mouseMoved(MouseEvent e);
}
```

Lots of boilerplate

- There are seven methods in the two interfaces.
- We only want to do something interesting for three of them.
- Need "trivial" implementations of the other four to implement the interface…

```
public void mouseMoved(MouseEvent e)   { return; }
public void mouseClicked(MouseEvent e) { return; }
public void mouseEntered(MouseEvent e) { return; }
public void mouseExited(MouseEvent e)  { return; }
```

- Solution: MouseAdapter class…

Adapter classes:

- Swing provides a collection of abstract event adapter classes
- These adapter classes implement listener interfaces with empty, do-nothing methods
- To implement a listener class, we extend an adapter class and override just the methods we need

```
private class Mouse extends MouseAdapter {
   public void mousePressed(MouseEvent e) { … }
   public void mouseReleased(MouseEvent e) { … }
   public void mouseDragged(MouseEvent e) { … }
}
```

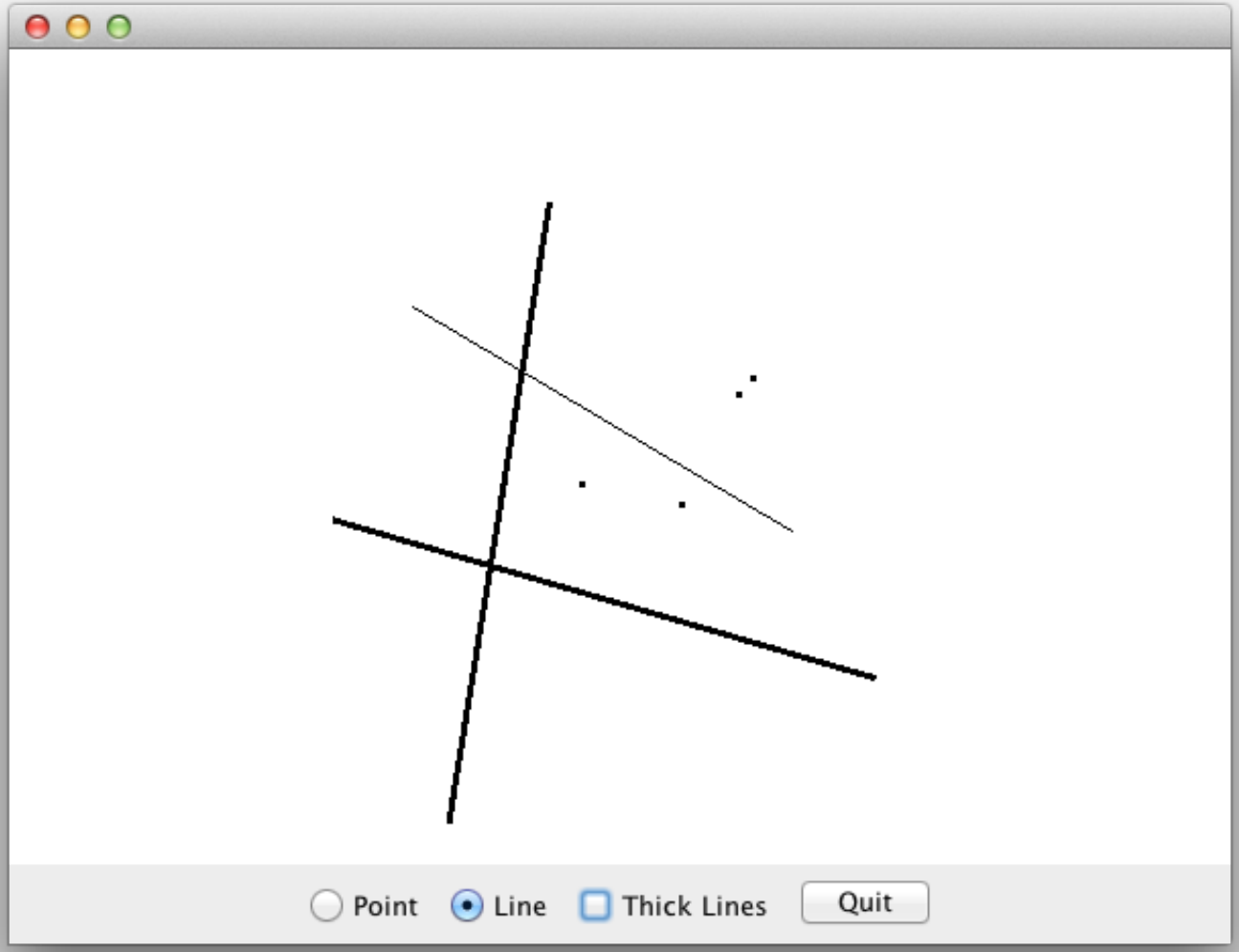
Paint Demo

(PaintA.java … PaintE.java)

Paint Revisited

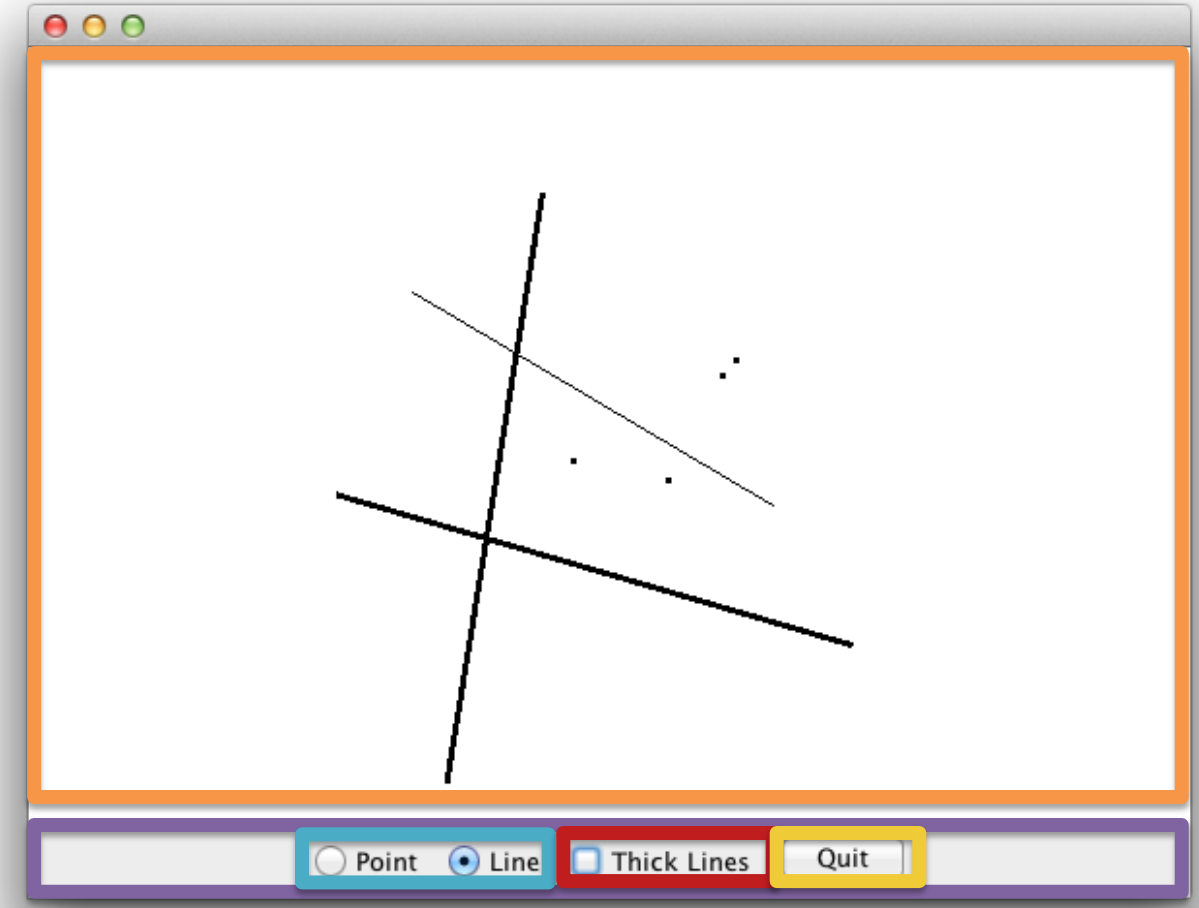
Using Anonymous Inner Classes

Refactoring for OO Design



What layout would you use for this app? What components would you use?

Canvas
subclass of
JPanel
(canvas)



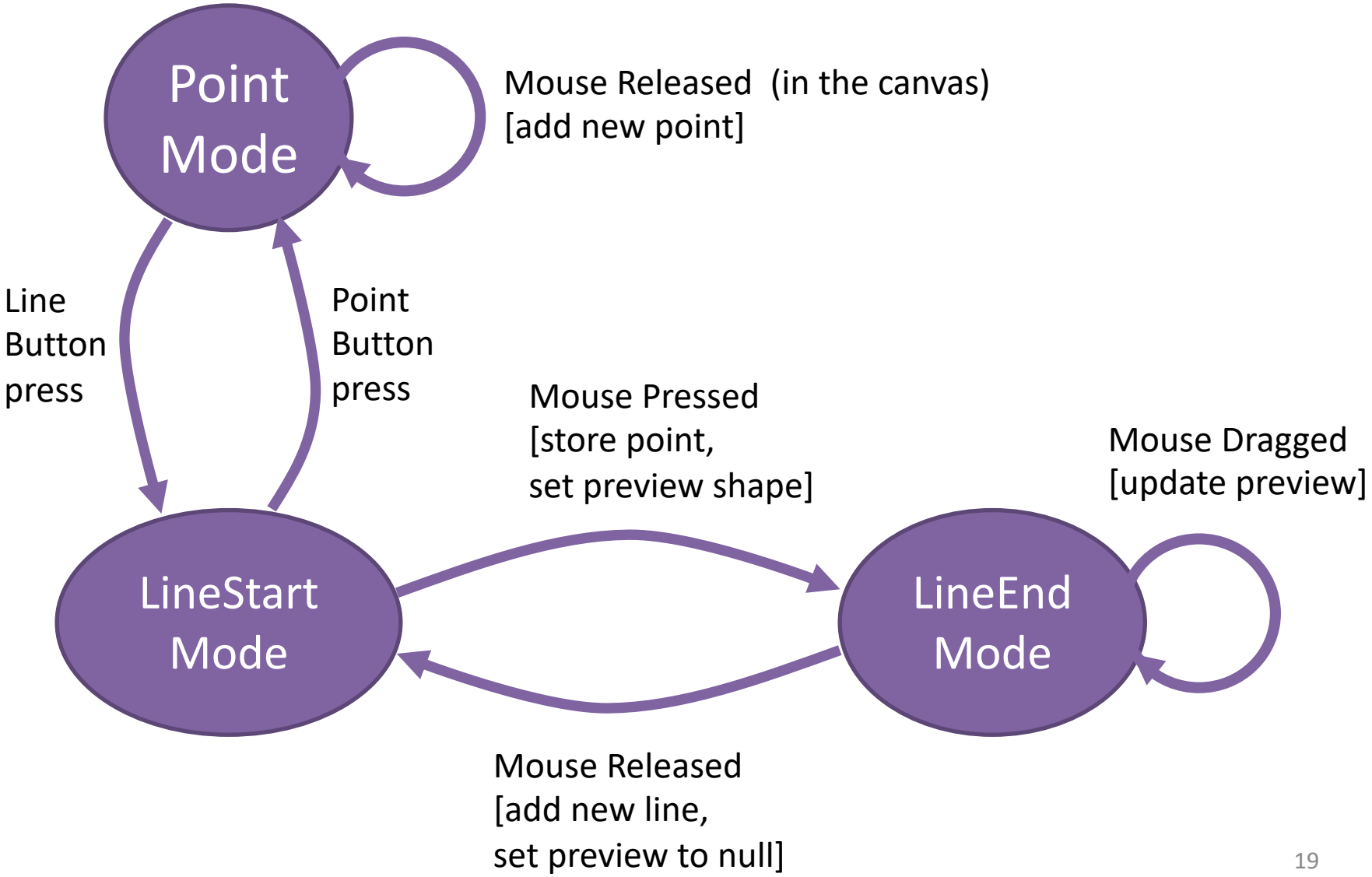
JPanel
(toolbar)

JRadioButton
(point, line)

JCheckbox
(thick)

JButton
(quit)

Mouse Interaction in Paint



Advanced Java Miscellany

Advanced Java

- Design Patterns (MVC)
- Java Streams (and lambdas)
- Threads & Synchronization
- Garbage Collection
- Hashing: HashSets & HashMaps

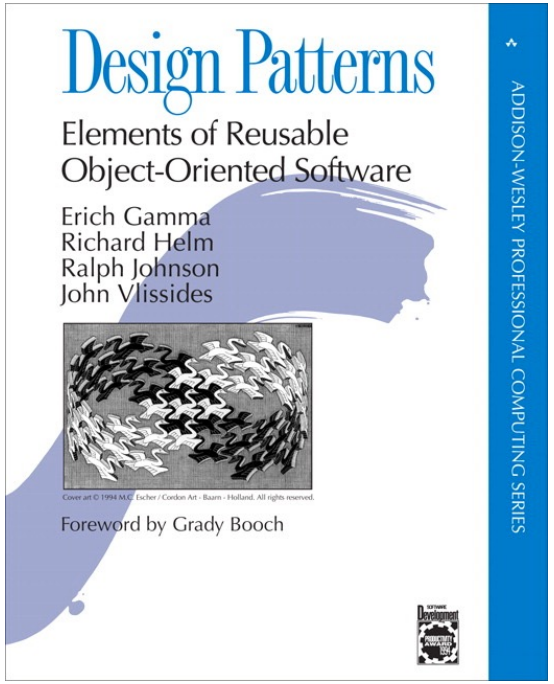
The slides touch on these. Lecture will cover only some parts...

- Packages
- JVM (Java Virtual Machine) and compiler details:
  - class loaders, security managers, just-in-time compilation
- Advanced Generics
  - *Bounded Polymorphism*: type parameters with 'extends' constraints
    `class C<A extends Runnable> { … }`
  - Type Erasure
  - Interaction between generics and arrays
- Reflection
  - The `Class` class

For all the beautiful details:
Java Language Specification
http://docs.oracle.com/javase/specs/

Design Patterns

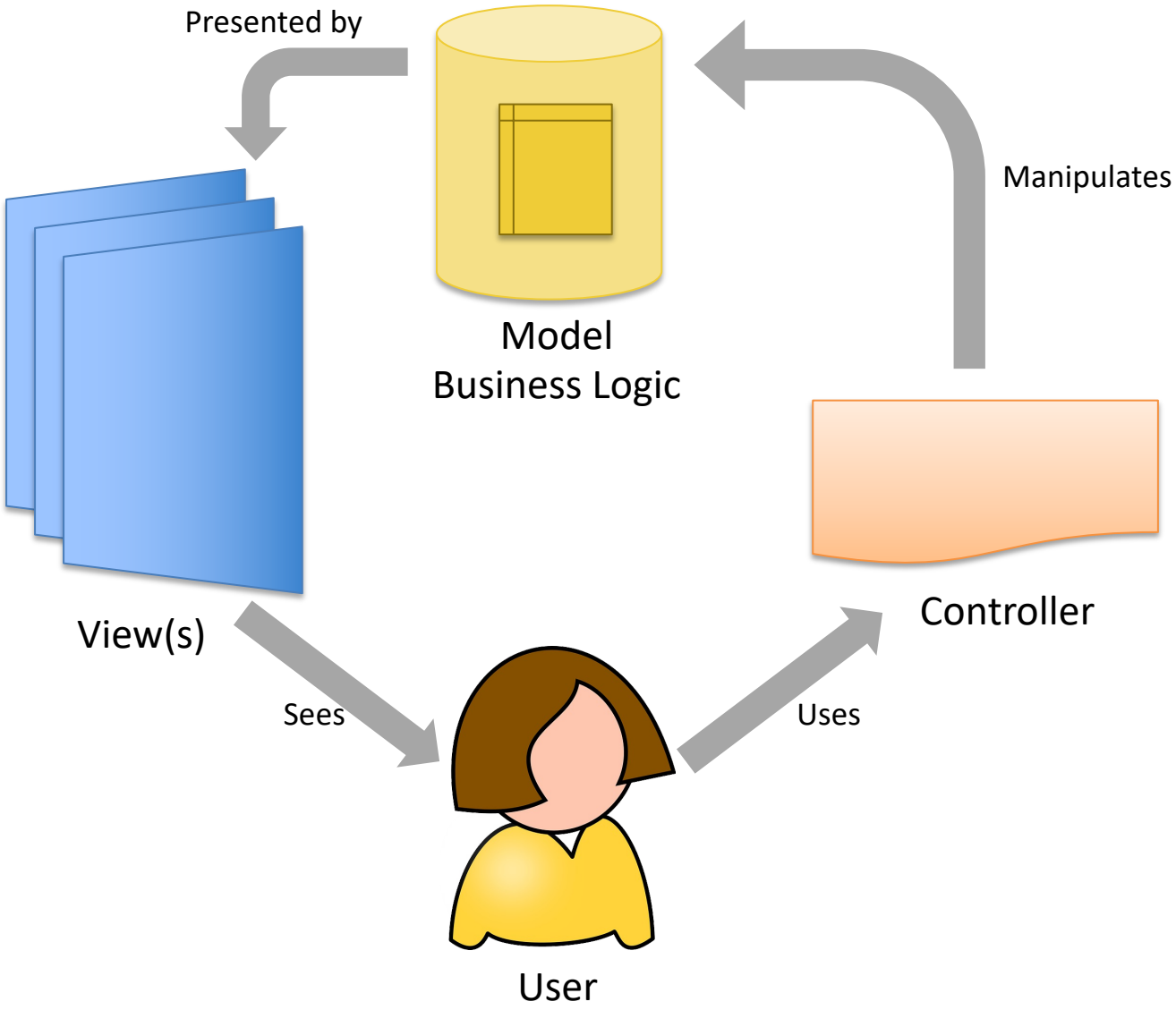
- Design Patterns
  - Influential OO design book published in 1994 (so a bit dated)
  - Identifies many common situations and "patterns" for implementing them in OO languages



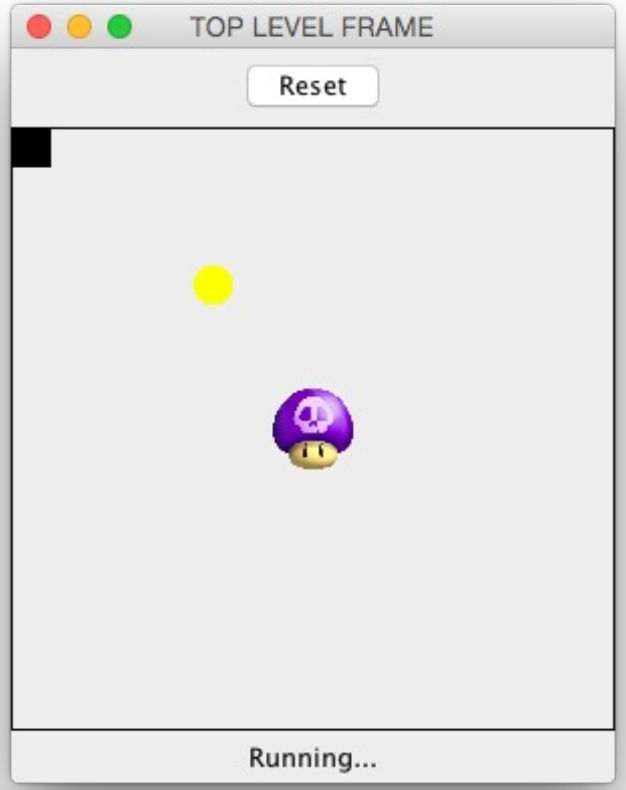
- Some we have seen explicitly:
  - e.g. *Iterator* pattern
- Some we've used but not explicitly described:
  - e.g. The parts of the Chat HW uses the *Factory* pattern
- Some are workarounds for OO's lack of some features:
  - e.g. The *Visitor* pattern is like OCaml's fold + pattern matching

Model View Controller Design Pattern

Model-View-Controller Design Pattern



Example 1: Mushroom of Doom



Example: MOD Program Structure

- GameCourt, GameObj + subclass local state
  - object location & velocity
  - status of the game (playing, win, loss)
  - how the objects interact with eachother (tick)

Model

- Draw methods
  - paintComponent in GameCourt
  - draw methods in GameObj subclasses
  - status label

View

- Game / GameCourt
  - Reset button (updates model)
  - Keyboard control (updates square velocity)

Controller

Example: Paint Program Structure

- Main frame for application (class Paint)
  - List of shapes to draw
  - The current color
  - The current line thickness

Model

- Drawing panel (class Canvas, inner class of Paint)

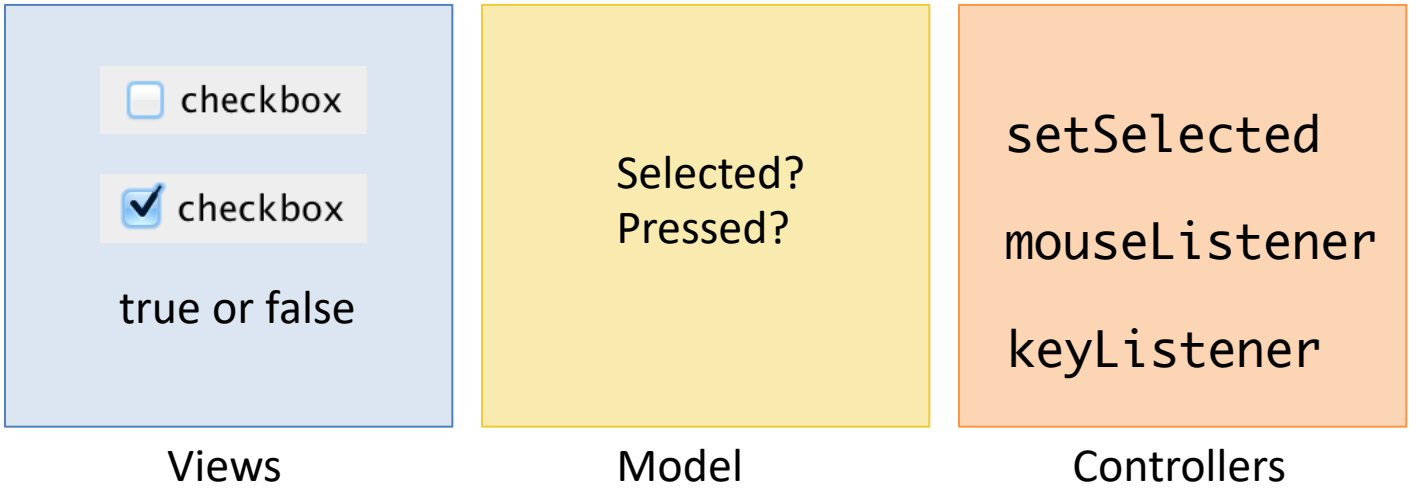
View

- Control panel (class JPanel)
  - Contains radio buttons for selecting shape to draw
  - Line thickness checkbox, undo and quit buttons

Controller

- Connections between Preview shape (if any…)
  - Preview Shape: View <-> Controller
  - MouseAdapter: Controller <-> Model

Example: CheckBox

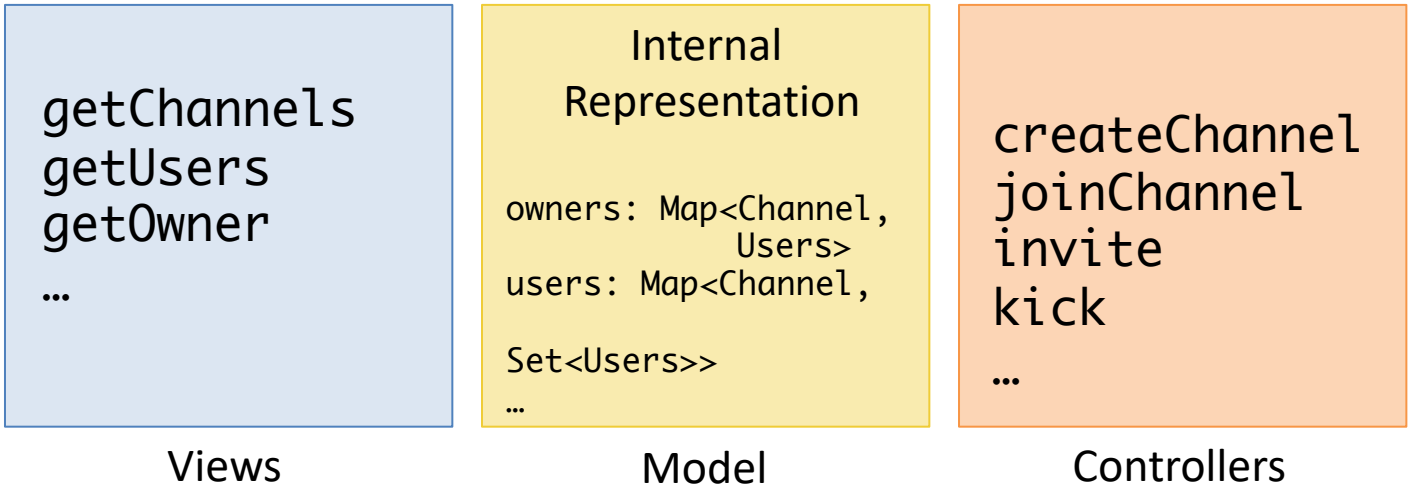



```
Class JToggleButton.ToggleButtonModel

boolean   isSelected()
void   setPressed(boolean b)
void   setSelected(boolean b)
```

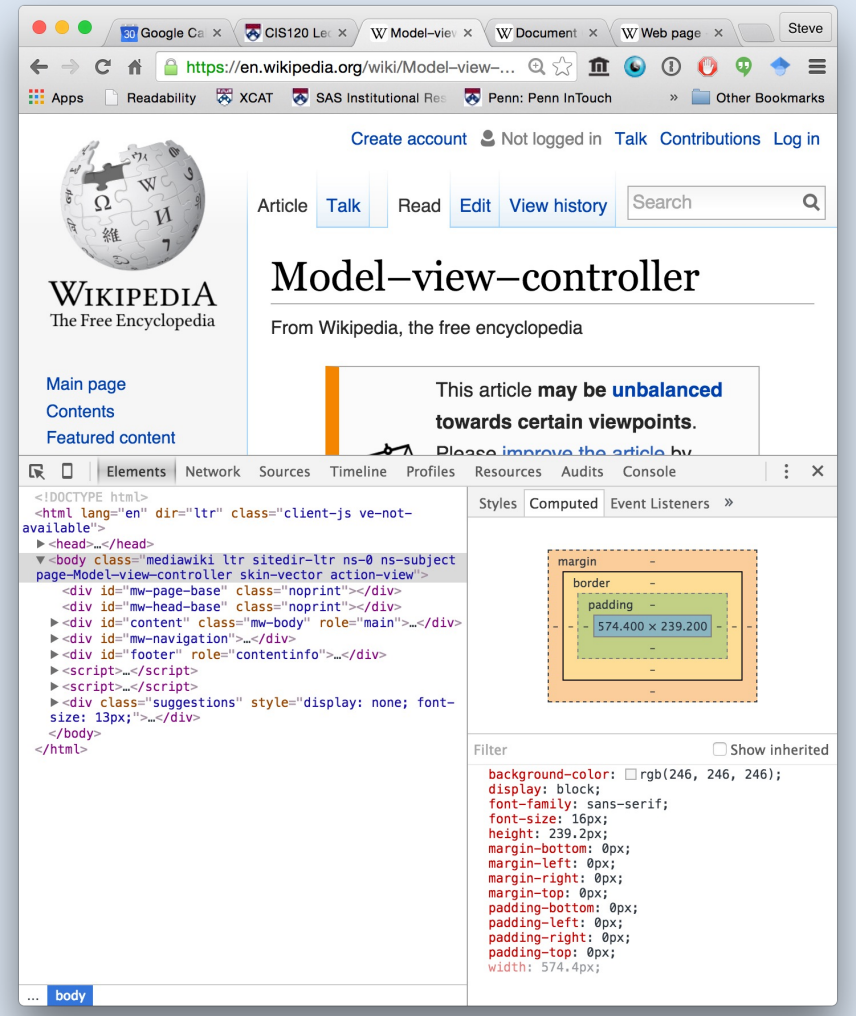
Checks if the button is selected.
Sets the pressed state of the button.
Sets the selected state of the button.

Example: Chat Server



ServerModel

Example: Web Pages



Views

Internal Representation: DOM (Document Object Model)

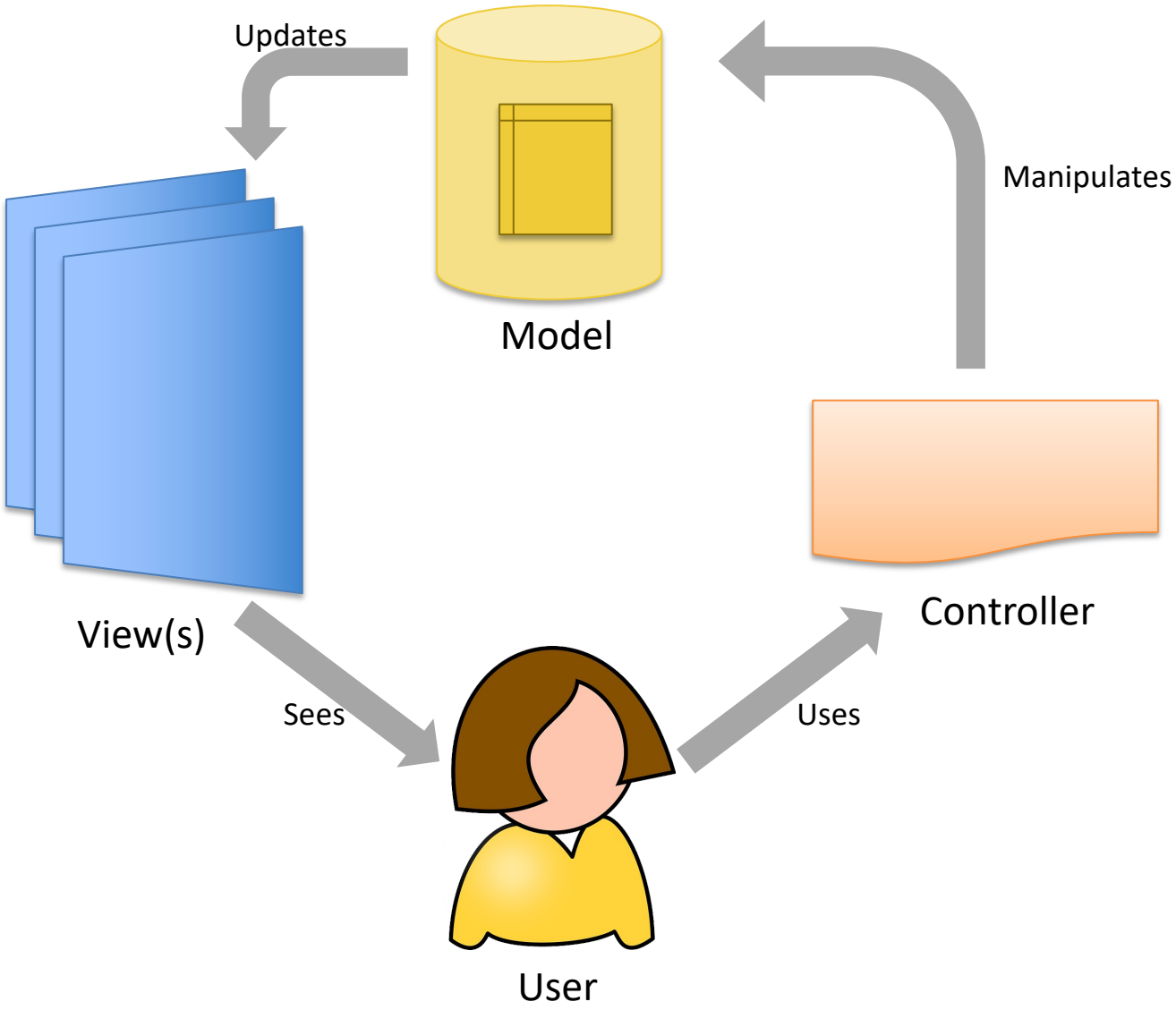
Model

JavaScript API

```
document.
addEventListener()
```

Controllers

MVC Pattern


Updates
Model
Manipulates
View(s)
Controller
Sees
Uses
User

MVC Benefits?

- Decouples "model state" from how that state is presented and manipulated
  - Suggests how to decompose the design to make it more flexible
- Multiple views
  - e.g. from different angles, or for multiple different users
- Multiple controllers
  - e.g. mouse vs. keyboard interaction
- Key benefit: Makes the model **testable** independent of the GUI

MVC Variations

- Many variations on MVC pattern

- Hierarchical / Nested
  - As in the Swing libraries, in which JComponents often have a "model" and a "controller" part

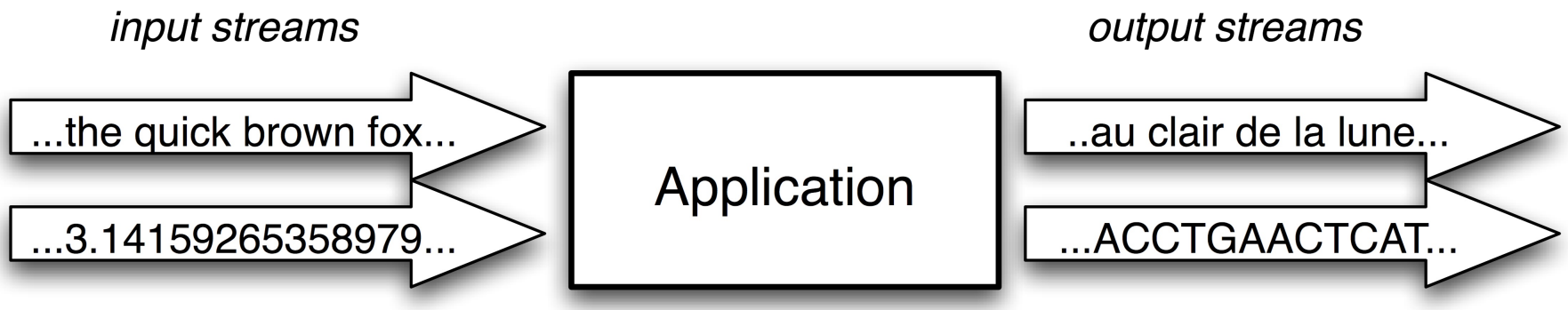
- Coupling between Model / View or View / Controller
  - e.g. in MOD the Model and the View are coupled because the model carries most of the information about the view

Functional Programming + Streams

(See Streams.java)

I/O Streams

- The *stream* abstraction represents a communication channel with the outside world.
  - can be used to read or write a potentially unbounded number of data items (unlike a list)
  - data items are read from or written to a stream one at a time
- The Java I/O library uses subtyping to provide a unified view of disparate data sources and sinks.



Streams redux

- Use *streams* of elements to support functional-style operations on collections
- Key differences between streams and collections:
  - No storage (i.e., not a data structure)
  - Functional in nature (i.e., do not modify the source)
  - Possibly unbounded (i.e., computations on infinite streams can complete in finite time)
  - Consumable (i.e., similar to Iterator)
  - Lazy-seeking
    - "Find the first input String that begins with a vowel" doesn't need to look at *all* Strings from the input

Creating Streams (1)

- From a `Collection` via the `stream()` and `parallelStream()` methods
- From an array via `Arrays.stream()`
- The lines of a file can be obtained from `BufferedReader.lines()`
- Streams of random numbers can be obtained from `Random.ints()`;
- Numerous other stream-bearing methods in the JDK

Creating Streams (2)

- Can create your own Low-Level Stream
- Similar to having a custom class like `WordScanner` that implements `Iterator`

- `Spliterator` – parallel analogue to `Iterator`
  - (Possibly infinite) Collection of elements
  - Support for:
    - Sequentially advancing elements (similar to `next()`)
    - Bulk Traversal (performs the given action for each remaining element, *sequentially* in the current thread)
    - Splitting off some portion of the input into another spliterator, which can be processed in *parallel* (much easier than doing threads manually!)

Stream Pipeline Operations

- Intermediate (Stream-producing) operations
  - E.g., `filter`, `map`, `sorted`
  - Similar to transform in Ocaml
  - Return a new stream
  - Always lazy (produce elements as needed, not ahead of time)
  - Traversal of the source does not begin until the terminal operation of the pipeline is executed
- Terminal (value- or side-effect-producing) operations
  - E.g. `forEach`, `reduce`, `findFirst`, `allMatch`, `max`, `min`
  - Similar to fold in Ocaml
  - Produce a result or side-effect

- Combined to create Stream pipelines

Lambdas, Streams, Pipelines

The Beauty and Joy of functional programming, now in Java!

```
roster.stream()
      .filter(p ->
               p.getHomeSchool().equals("SEAS")
               && p.getAge() >= 18
               && p.getAge() <= 25)
      .map(p -> p.getEmailAddress())
      .forEach(email -> System.out.println(email));
```

```
int sum = widgets.stream()
                 .filter(b -> b.getColor() == RED)
                 .mapToInt(b -> b.getWeight())
                 .sum();
```

Functional Programming + Parallelism

(See Streams.java)

Functional Programming + Parallelism

- Parallelism by design in Java 1.8
  - Streams are functional in nature (i.e., do not modify the source)
  - `Spliterator`
- Much easier than doing it manually
  - No need for `synchronized`
  - No need for locks
  - Don't have to worry about race conditions!
- Use `parallelStream()` (instead of `stream()`)!
  - Java will automatically create the necessary threads and scale based on your computer's hardware

Sample Problem

- Given a list of numbers, find the sum of the squares of the numbers

- **Iterative Approach**

```
int sum = 0;
for (int i = 0; i < list.size(); i++) {
   int x = list.get(i);
   sum += x * x;
}
```

- Works, more likely to have bugs (off-by-one), harder to parallelize

Sample Problem

- Given a list of numbers, find the sum of the squares of the numbers

- **Functional Approach**
- Use `transform` and `fold` (aka `map` and `reduce` in Java)

```
list.parallelStream()
    .map(x -> x * x)
    .reduce(0, Integer::sum);
```

- Less likely to have bugs, much easier to parallelize