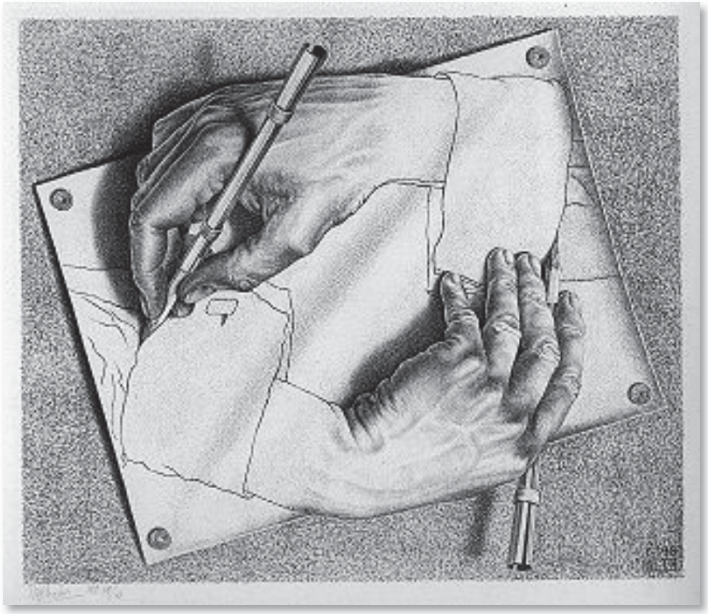
Programming Languages and Techniques (CIS120)

Bonus Lecture

November 22, 2017

Code ***is*** Data

Code *is* Data



M.C. Escher, Drawing Hands, 1948

Code *is* Data

- A Java source file is just a sequence of characters.
- We can represent programs with Strings!

```
String p_3 = "class C { public static void main(String args[])
```

```
String p_13 = "class C { public static void main(String args[])
 {System.out.println(\"Hello, world!\");}}";
```

• • •

```
String p_81201202342312312 30 = /* SpellChecker! */
      "class SpellChecker { public static void main(String args[]) {…}}
```

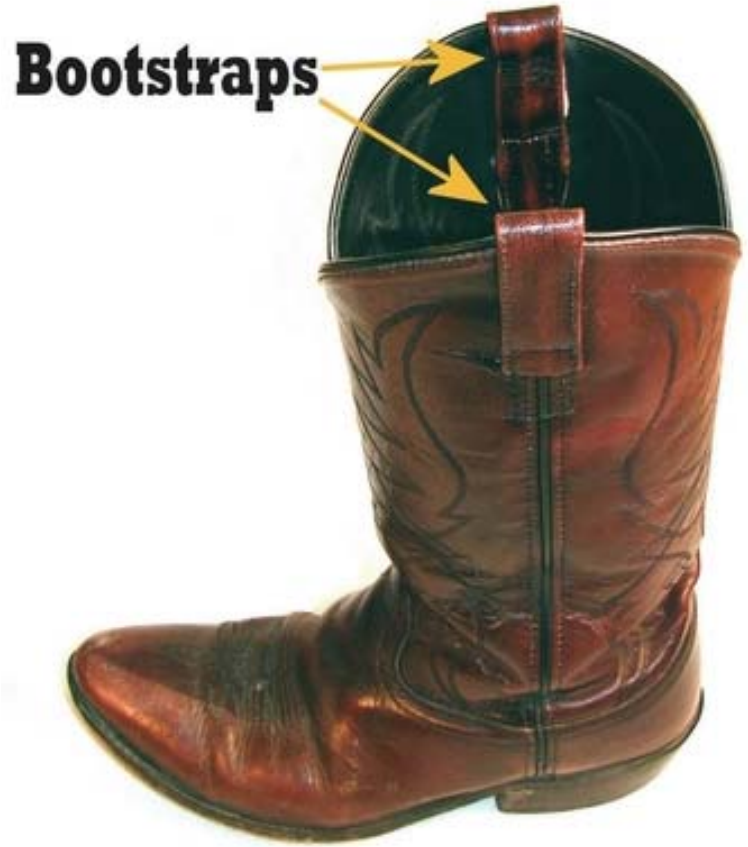
• • •

```
String p_99993249000902300239400823407023 4 = /* Minecraft! */
      "class Minecraft { public static void main(String args[]) {…}}";
```

• • •

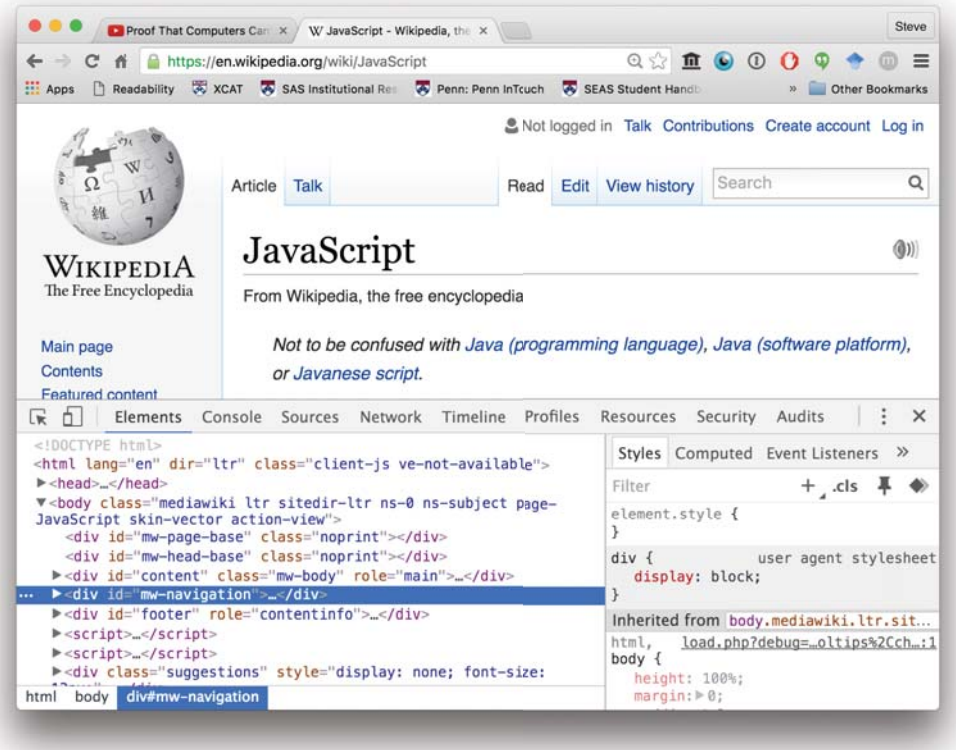
```
String p_992349923423999993249000234282340734502345 34 = /* Eclipse! */
      "class Eclipse { public static void main(String args[]) {…}}";
```

Consequence 1: Programs that manipulate programs



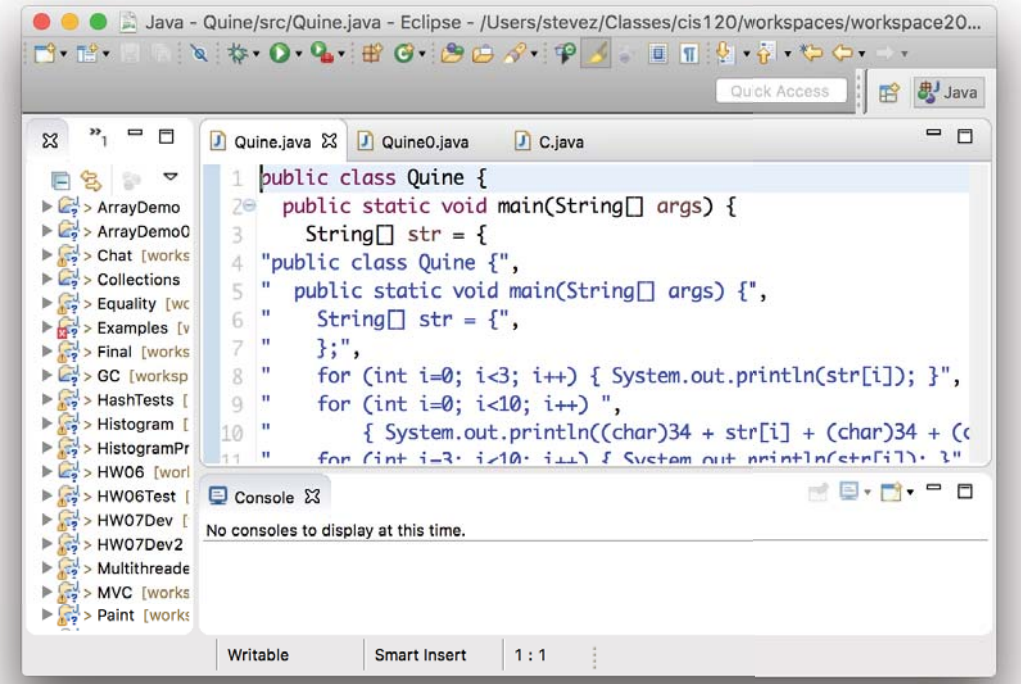
Interpreters

- We can create *programs* that manipulate *programs*
- An *interpreter* is a program that executes other programs
- `interpret ("3 + 4")` ➔ `7`
- Example 1: JavaScript



IDEs and Compilers

- Example 2: Eclipse
  - Note that Eclipse manipulates a *representation* of Java programs
  - Eclipse itself is written in Java
  - So you could use Eclipse to edit the code for Eclipse… ?!



- Example 3: Compiler
  - The Java compiler takes a representation of a Java program
  - It outputs a "low-level" representation of the program as a .class file (i.e. Java byte code)
  - Can also compile to other representations, e.g. x86 "machine code"

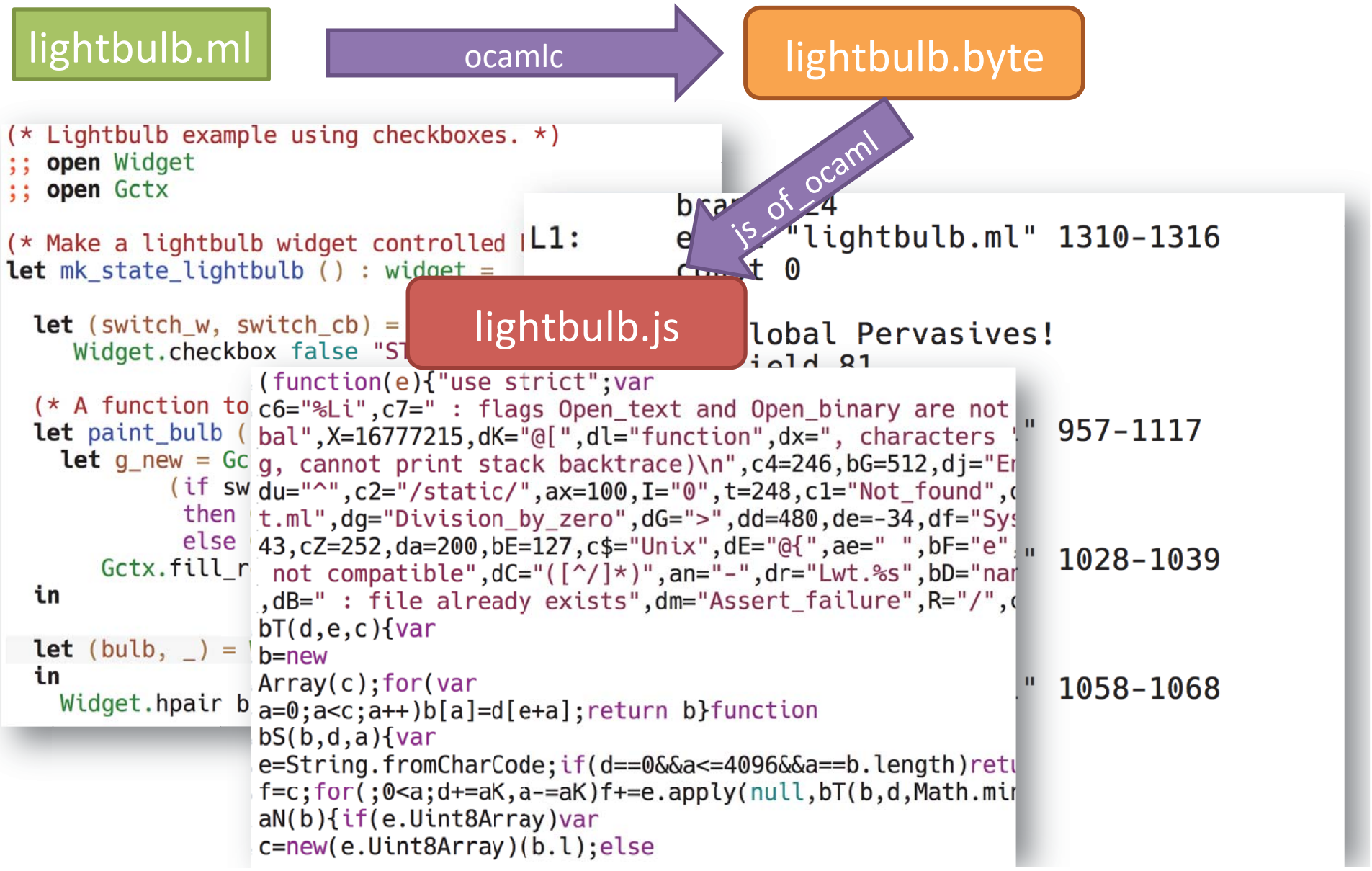
Example: js_of_ocaml in GUI project

lightbulb.ml → ocamlc → lightbulb.native

```
(* Lightbulb example using checkboxes. *)
;; open Widget
;; open Gctx

(* Make a lightbulb widget controlled by
let mk_state_lightbulb () : widget =

  let (switch_w, switch_cb) =
     Widget.checkbox false "STATE LIGHT"

  (* A function to display the bulb  *)
  let paint_bulb (g:gctx) : unit =
    let g_new = Gctx.with_color g
            (if switch_cb.get_value ()
             then Gctx.yellow
             else Gctx.black) in
       Gctx.fill_rect g_new (0, 99) (99,
  in

  let (bulb, _) = Widget.canvas (100,100)
  in
    Widget.hpair bulb switch_w
```

```
_camlLightbulb__mk_state_lightbulb_1253:
00000001000017d0        subq    $0x8, %rsp
00000001000017d4        leaq    _camlLightbulb__1(%rip), %rbx
00000001000017db        movq    $0x1, %rax
00000001000017e2        callq   _camlWidget__checkbox_1349
00000001000017e7        movq    %rax, (%rsp)
00000001000017eb        movq    0x8(%rax), %rdi
00000001000017ef        subq    $0x20, %r15
00000001000017f3        leaq    _caml_young_limit(%rip), %rax
00000001000017fa        cmpq    (%rax), %r15
00000001000017fd        jb      0x100001840
00000001000017ff        leaq    0x8(%r15), %rbx
0000000100001803        movq    $0xcf7, -0x8(%rbx)
000000010000180b        leaq    _camlLightbulb__paint_bulb_125
0000000100001812        movq    %rax, (%rbx)
0000000100001815        movq    $0x3, 0x8(%rbx)
000000010000181d        movq    %rdi, 0x10(%rbx)
0000000100001821        leaq    _camlLightbulb__6(%rip), %rax
0000000100001828        callq   _camlWidget__canvas_1330
000000010000182d        movq    (%rsp), %rbx
0000000100001831        movq    (%rbx), %rbx
0000000100001834        movq    (%rax), %rax
0000000100001837        addq    $0x8, %rsp
000000010000183b        jmp     _camlWidget__hpair_1272
0000000100001840        callq   _caml_call_gc
0000000100001845        jmp     0x1000017ef
0000000100001847        nopw    (%rax,%rax)
```

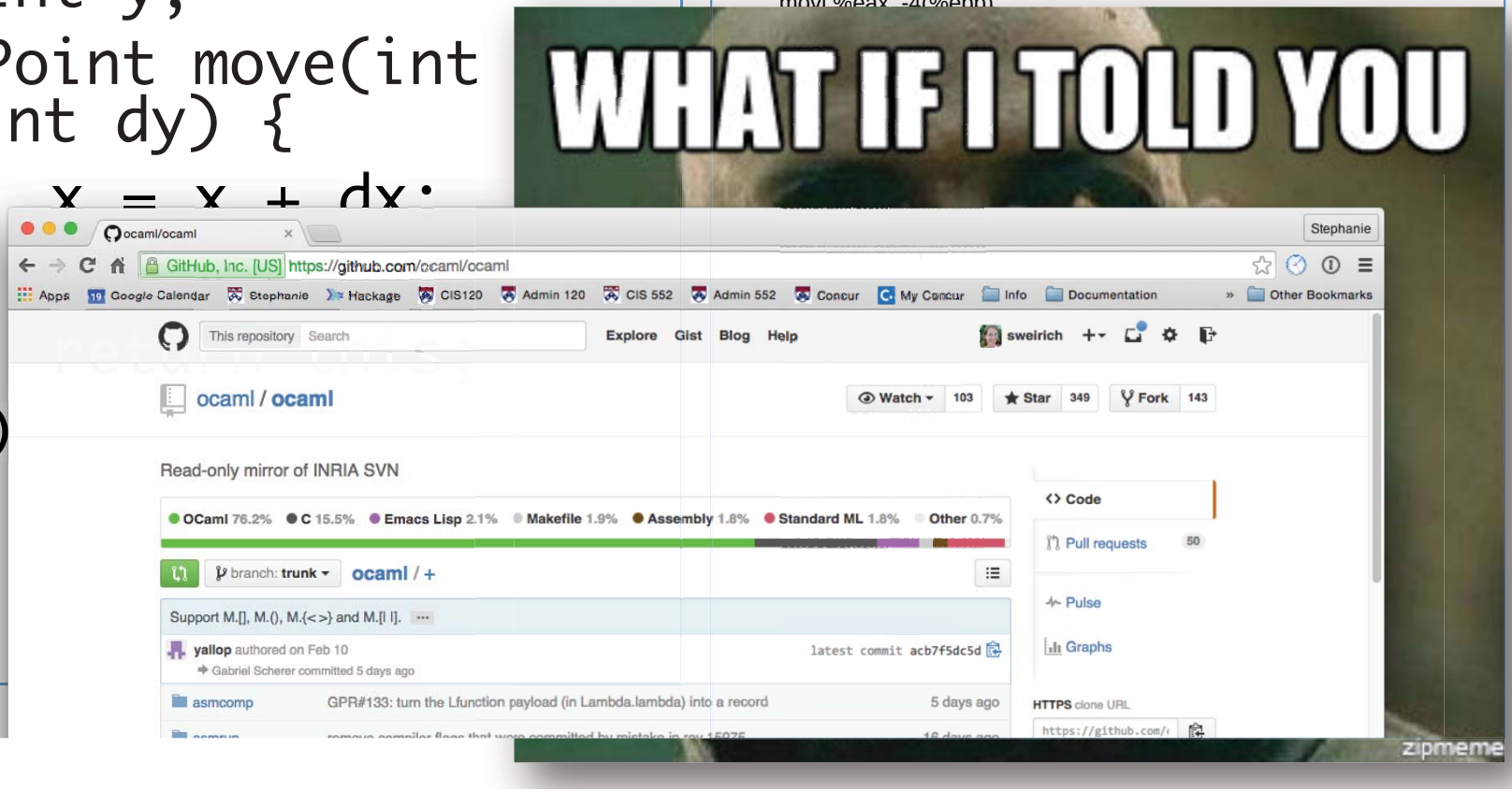
Example: js_of_ocaml in GUI project



Example Compilation: Java to X86

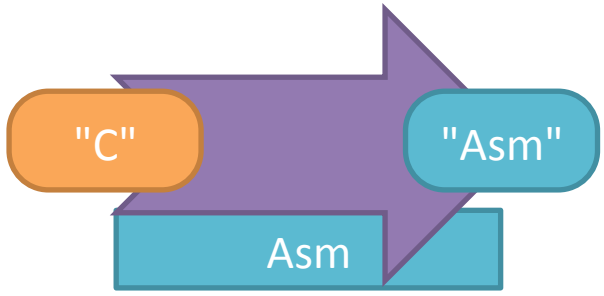
```
class Point {
   int x;
   int y;
   Point move(int
   int dy) {
      x = x + dx;
   }
}
```

```
.globl __fun__Point.move
__fun__Point.move:
      pushl %ebp
      movl %esp, %ebp
      subl $4, %esp
__5:
      movl 8(%ebp), %eax
      movl 4(%eax), %eax
      movl %eax, -4(%ebp)
```



Bootstrap Process

1. Implement, by hand in assembly, a compiler for a lower-level language like C (or create one by earlier bootstrapping work).

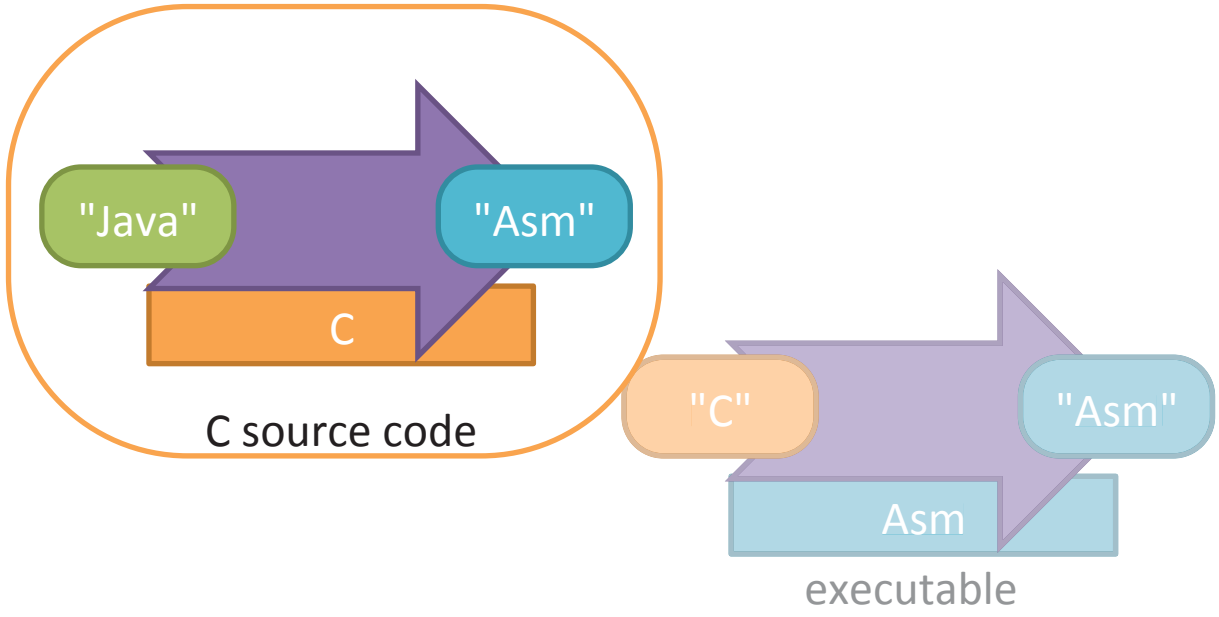


executable*

* Actually you need to write the first assembler in machine code by hand. These slides conflate "assembly" with "machine" code to keep things a bit simpler…

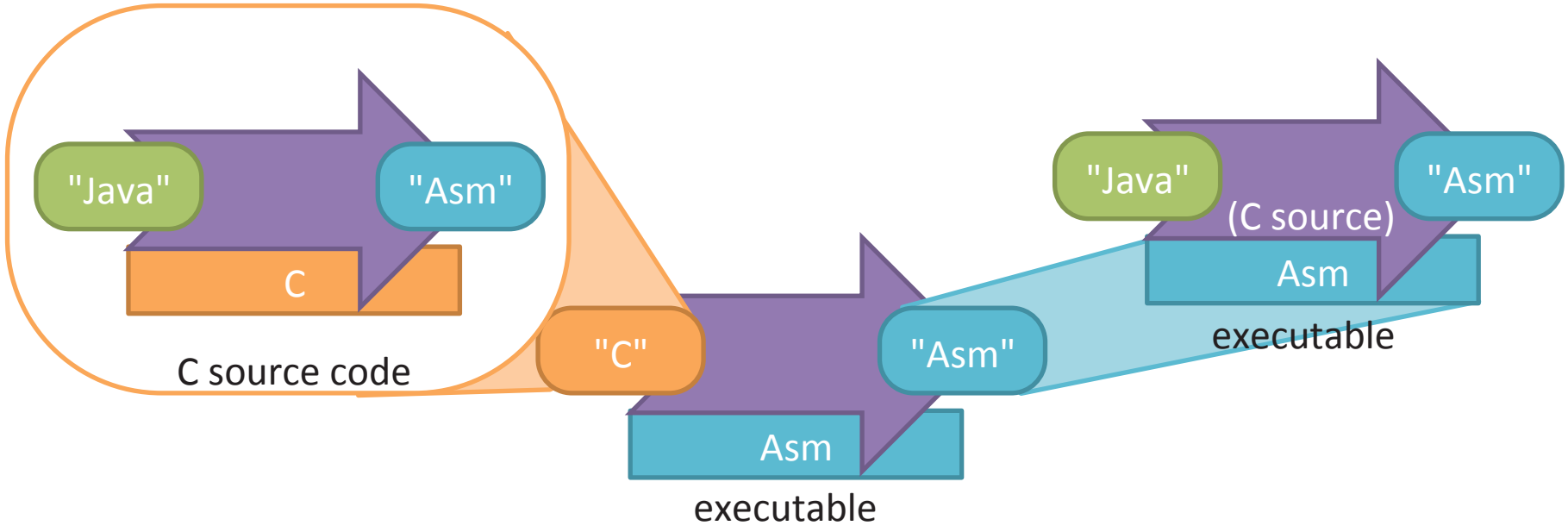
Bootstrap Process

2 Write, in C, a Java to Assembly compiler (or interpreter).



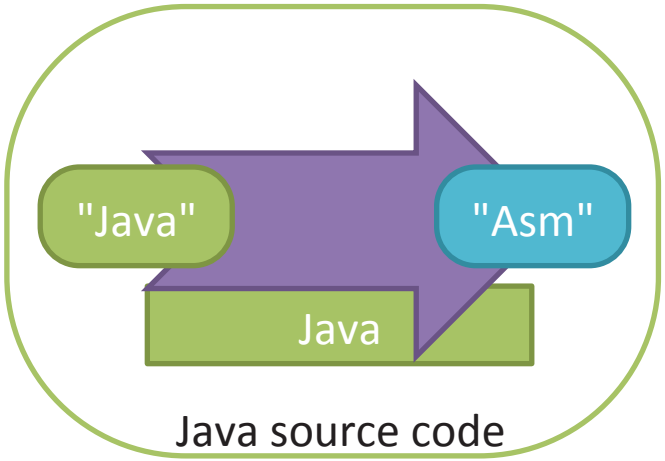
Bootstrap Process

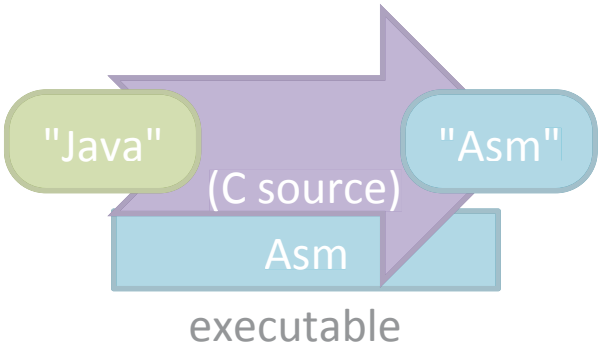
3 Use the C compiler to compile the Java compiler to produce a Java compiler executable.



Bootstrap Process

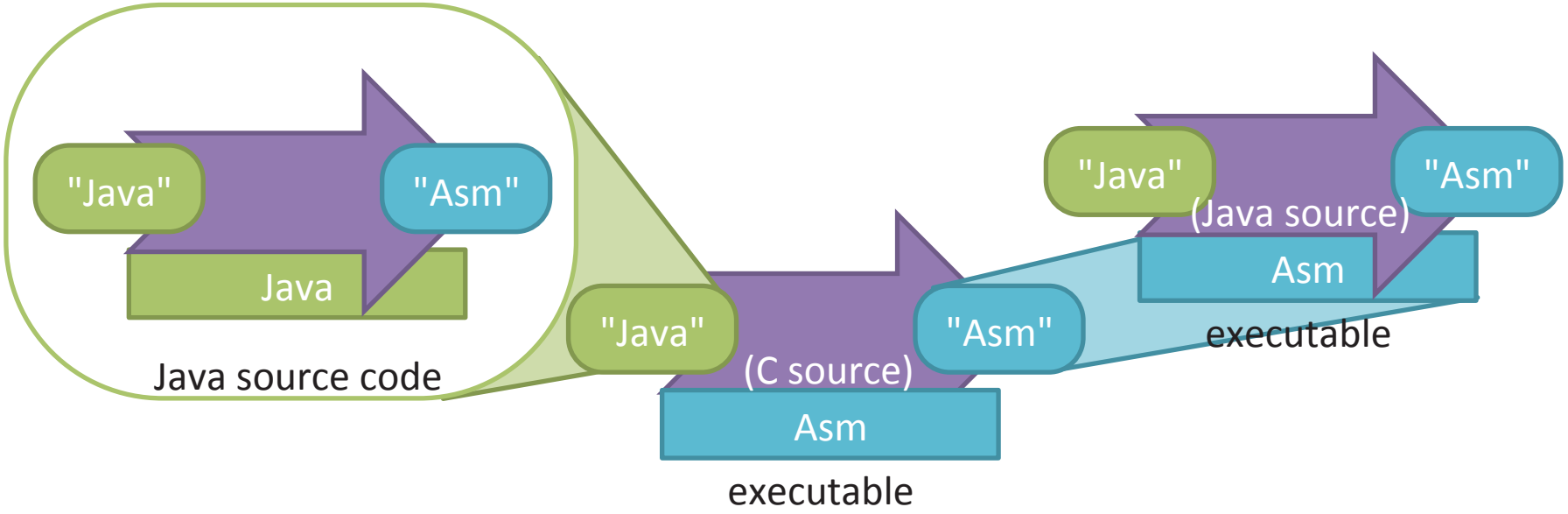
4 Write a Java compiler in Java.





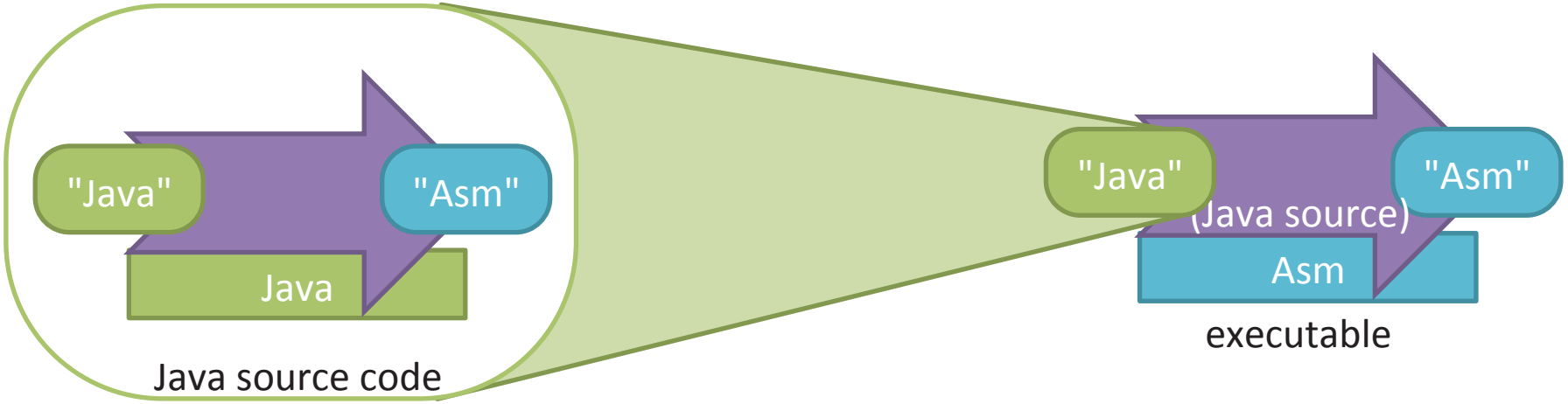
Bootstrap Process

5 Use the (C source) Java compiler to compile the Java compiler (Java source) implementation.



Bootstrap Process

6 Throw all the earlier stuff away and use the Java compiler executable (derived from Java source)..

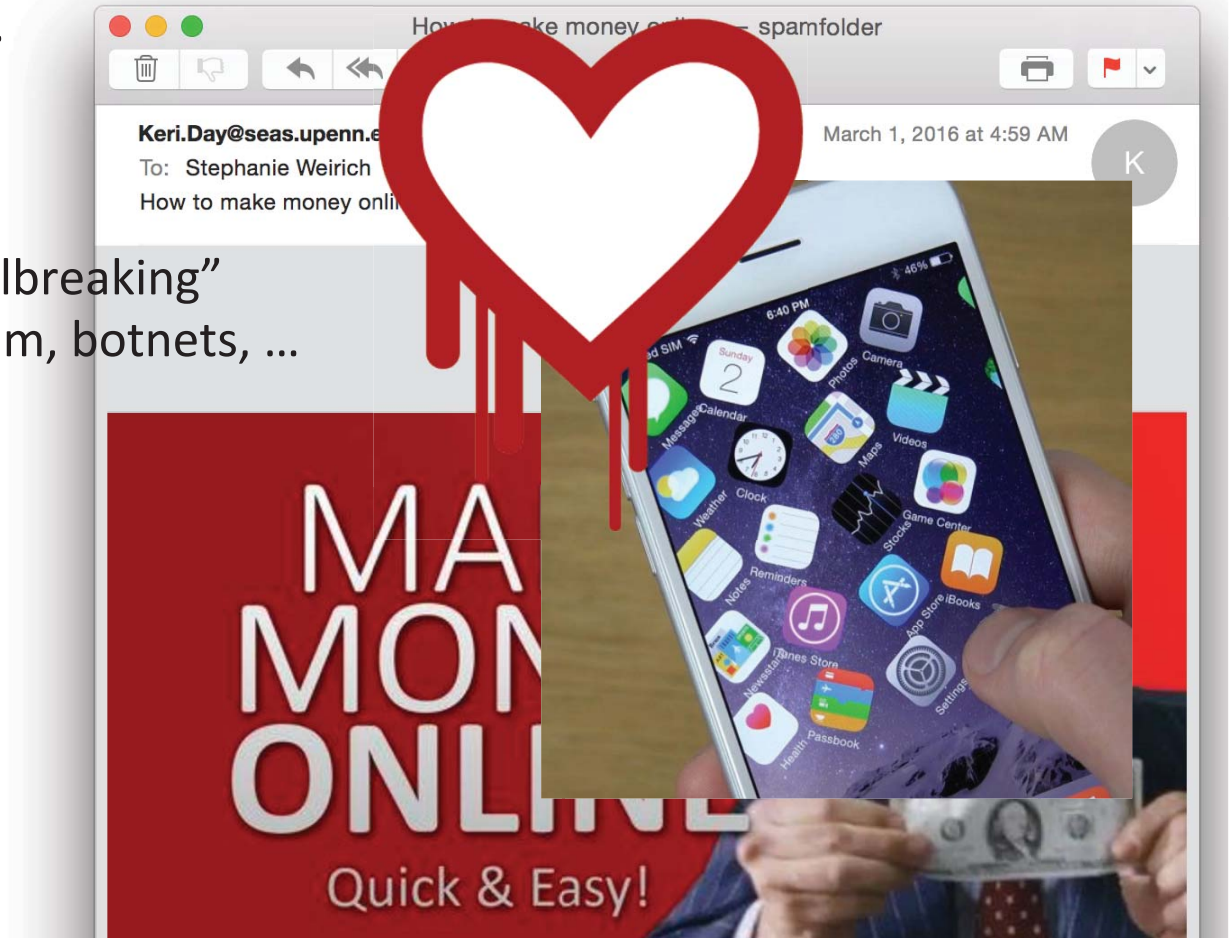


Consequence 2: Malware

Rene Magritte, The Human Condition, 1933

Consequence 2: Malware

- Why does Java do array bounds checking?
- *Unsafe* language like C and C++ don't do that checking;
  - They will happily let you write a program that "writes past" the end of an array.
- Result:
  - viruses, worms, "jailbreaking" mobile phones, Spam, botnets, …
- Fundamental issue:
  - Code is data.
  - Why?



Consider this C Program

```
void m() {
  char[] buffer = new char[2];

  char c = read();
  int i = 0;
  while (c != -1) {
      buffer[i] = c;
      c = read();
      i++;
  }
  process(buffer);
}

void main() {
  m();
  // do some more stuff
}
```

Notes:

- C doesn't check array bounds
- Unlike Java, it stores arrays directly on the stack
- What could possibly go wrong?

Abstract Stack Machine

"Stack Smashing Attack"

Abstract Stack Machine

Workspace

Stack

```
m();
// do some more stuff
```

Call to main() to start the program…

Abstract Stack Machine

Workspace

```
char[2] buffer;

char c = read();
int i = 0;
while (c != -1) {
  buffer[i] = c;
  c = read();
  i++;
}
process(buffer);
```

Stack

```
_;
// do some more stuff
```

Push the saved workspace, run m()

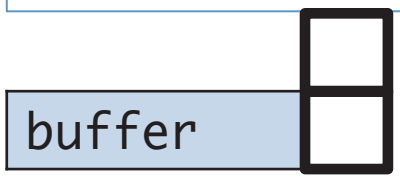
Abstract Stack Machine

Workspace

```
char c = read();
int i = 0;
while (c != -1) {
  buffer[i] = c;
  c = read();
  i++;
}
process(buffer);
```

Stack

```
_;
// do some more stuff
```



Allocate space for buffer on the stack.

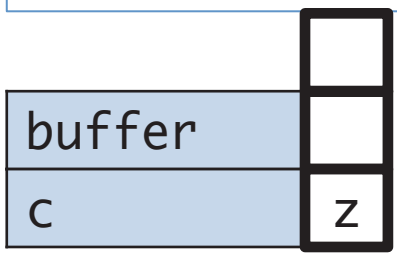
Abstract Stack Machine

Workspace

```
int i = 0;
while (c != -1) {
  buffer[i] = c;
  c = read();
  i++;
}
process(buffer);
```

Stack

```
_;
// do some more stuff
```



Allocate space for c.
Read the first user input… 'z'.

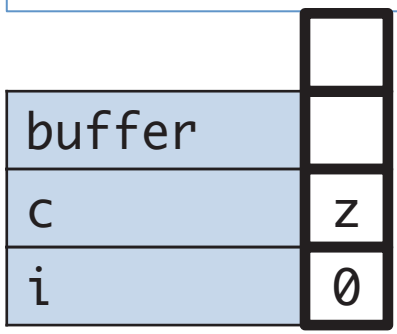
Abstract Stack Machine

Workspace

```
while (c != -1) {
  buffer[i] = c;
  c = read();
  i++;
}
process(buffer);
```

Stack

```
_;
// do some more stuff
```



Allocate space for i.

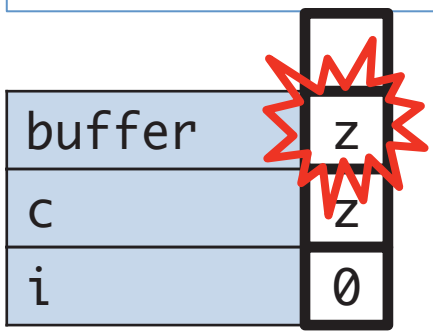
Abstract Stack Machine

Workspace

```
while (c != -1) {
  buffer[i] = c;
  c = read();
  i++;
}
process(buffer);
```

Stack

```
_;
// do some more stuff
```



Copy (contents of) c to buffer[0]

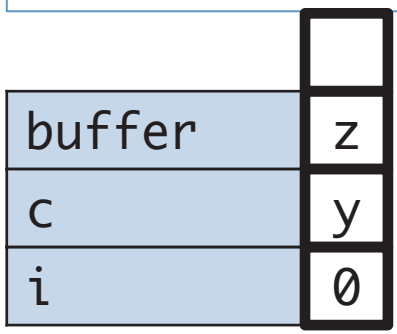
Abstract Stack Machine

Workspace

```
while (c != -1) {
  buffer[i] = c;
  c = read();
  i++;
}
process(buffer);
```

Stack

```
_;
// do some more stuff
```



Read next character … 'y'

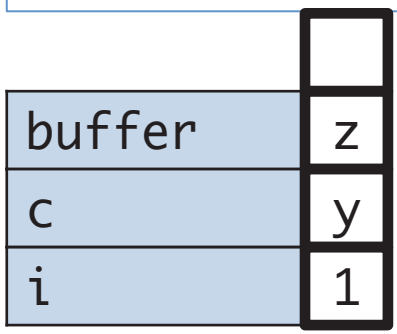
Abstract Stack Machine

Workspace

```
while (c != -1) {
  buffer[i] = c;
  c = read();
  i++;
}
process(buffer);
```

Stack

```
_;
// do some more stuff
```



Increment i

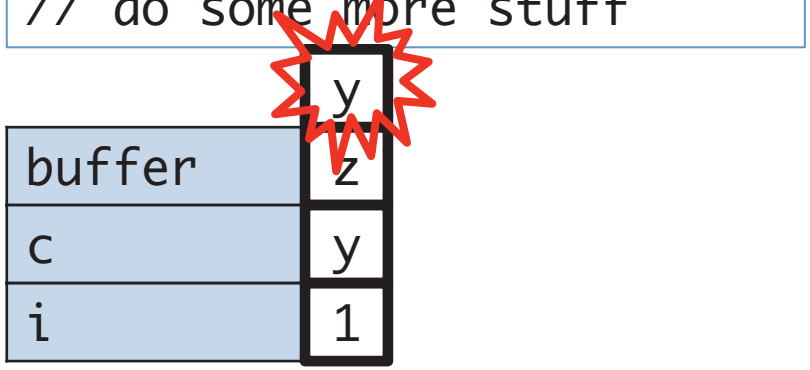
Abstract Stack Machine

Workspace

```
while (c != -1) {
  buffer[i] = c;
  c = read();
  i++;
}
process(buffer);
```

Stack

```
_;
// do some more stuff
```



Copy (contents of) c to buffer[1]

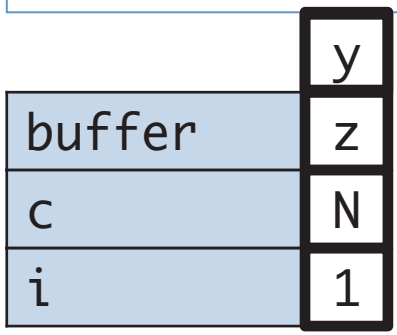
Abstract Stack Machine

Workspace

```
while (c != -1) {
  buffer[i] = c;
  c = read();
  i++;
}
process(buffer);
```

Stack

```
_;
// do some more stuff
```



Read next character … 'N'

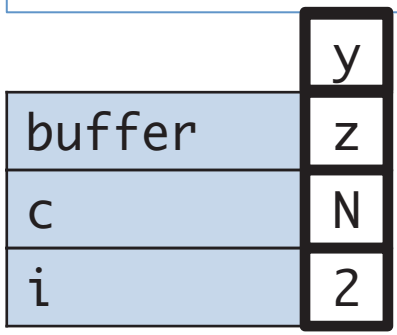
Abstract Stack Machine

Workspace

```
while (c != -1) {
  buffer[i] = c;
  c = read();
  i++;
}
process(buffer);
```

Stack

```
_;
// do some more stuff
```



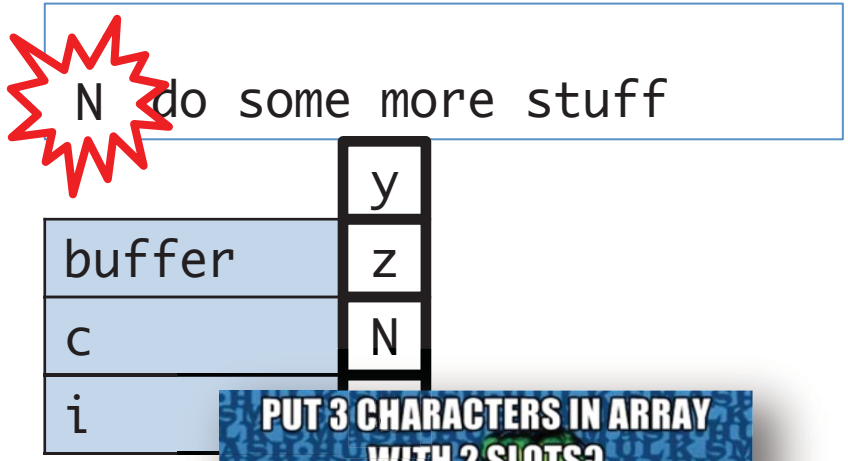
Increment i

Abstract Stack Machine

Workspace

```
while (c != -1) {
  buffer[i] = c;
  c = read();
  i++;
}
process(buffer);
```

Stack



Copy (contents of) c to buffer[2] ?!?

Overwrites the saved workspace!?

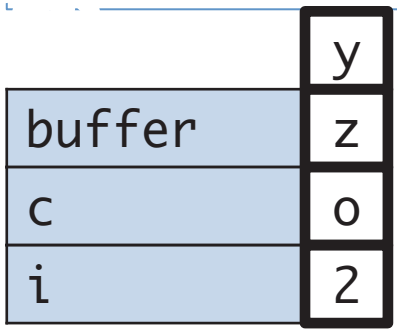
Abstract Stack Machine

Workspace

```
while (c != -1) {
  buffer[i] = c;
  c = read();
  i++;
}
process(buffer);
```

Stack

```
N  do some more stuff
```



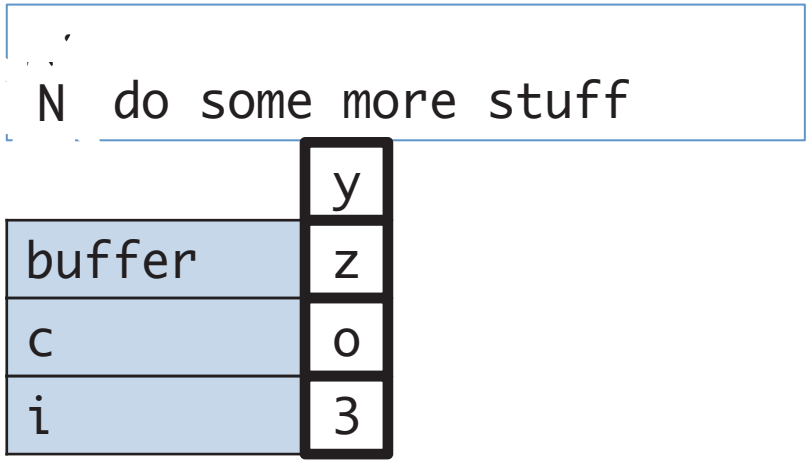
Keep going… read 'o'…

Abstract Stack Machine

Workspace

```
while (c != -1) {
  buffer[i] = c;
  c = read();
  i++;
}
process(buffer);
```

Stack



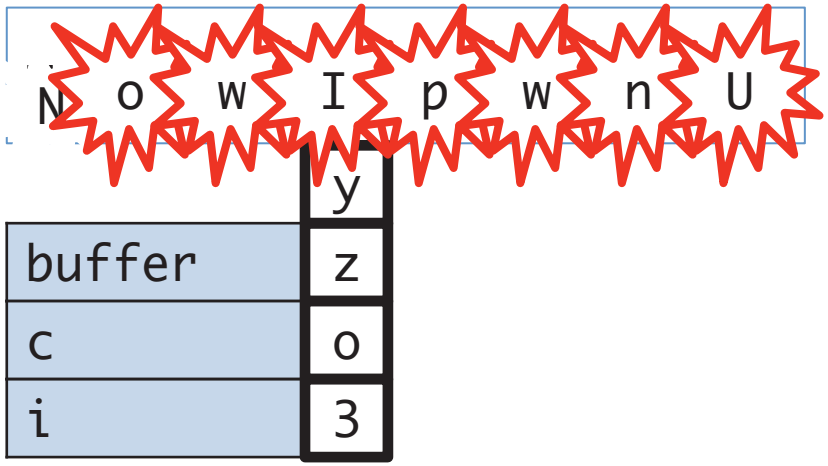
Keep going… read 'o'…increment i…

Abstract Stack Machine

Workspace

```
while (c != -1) {
  buffer[i] = c;
  c = read();
  i++;
}
process(buffer);
```

Stack



Keep going… read 'o'…increment i…write 'o' into saved workspace…

Abstract Stack Machine

Workspace

Stack

Now I pwn U!!!!

buffer	z
c	o
i	3

Later…

Abstract Stack Machine

Workspace

Stack

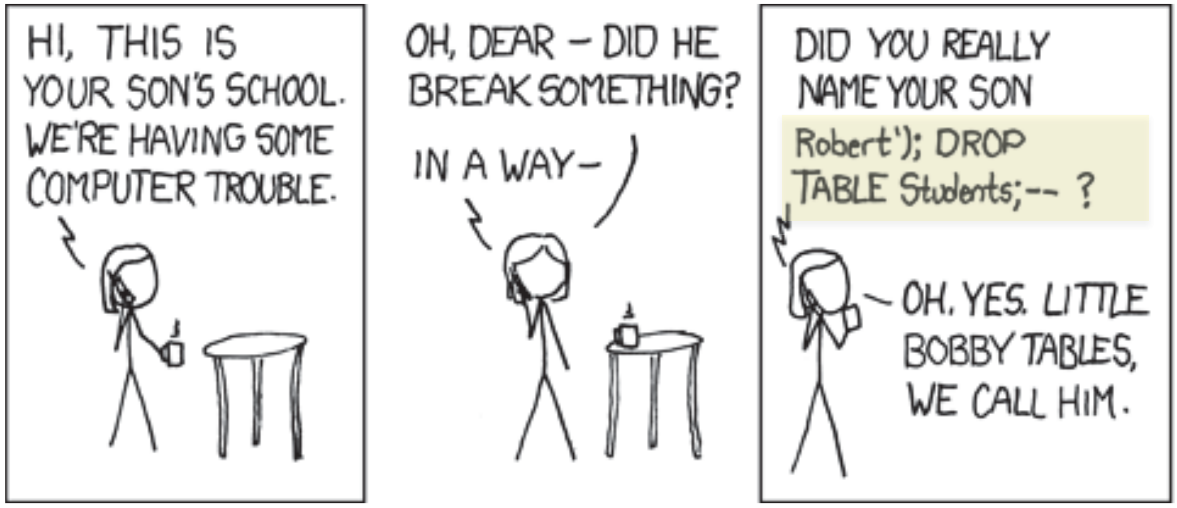
```
Now I pwn U!!!!
```

The stack smashing attack successfully wrote *arbitrary* code into the program's workspace…

Other Code Injection Attacks

```
void registerStudent() {
  print("Welcome to student registration.");
  print("Please enter your name:");
  String name = readLine();
  evalSQL("INSERT INTO Students('" + name + "')"  );
}
```

```
"INSERT INTO Students('Robert'); DROP TABLE Students; --')"
        + "Robert'); DROP TABLE Students; --" + "')"
```



http://xkcd.com/327/

Consequence 3: Undecidability

Undecidability Theorem

Theorem: *It is impossible to write a method*

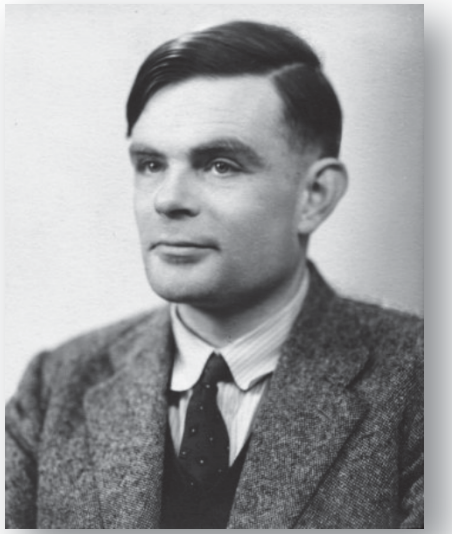
```
boolean halts(String prog)
```

such that, for any valid Java program P represented as a string `p_P`,

```
halts(p_P)
```

returns true exactly when the program P halts, and false otherwise.

Alonzo Church, April 1936



Alan Turing, May 1936

Halt Detector

- Suppose we could write such a program:

```
class HaltDetector {
  public static boolean halts(String javaProgram) {
    // ...do some super-clever analysis...
    // return true if javaProgram halts
    // return false if javaProgram does not
  }
}
```

- A correct implementation of HaltDetector.halts(p) always returns either true or false
  - i.e., it never raises an exception or loops
- HaltDetector.halts(p) ⇒ true means "p halts"
- HaltDetector.halts(p) ⇒ false means "p loops forever"

Do these methods halt?

```
boolean m(){ return false; }
```

⇒ YES

```
boolean m(){ return m(); }
```

⇒ NO (assuming infinite stack space)

```
boolean m() {
    if ("abc".length() == 3 ) return true;
    else return m();
}
```

⇒ YES

Does this method halt for *all* n?

```
boolean m(int n) {
  if (n<=1) return true;
  else if ((n%2) == 0) return m(n/2);
  else return m(3*n + 1);
}
```

Assuming infinite amount of stack space and arbitrarily large integers, it is *unknown* whether this program halts for all (strictly) positive integers!

Collatz Conjecture proposed in 1937!

Consider this Program called Q:

```
class HaltDetector {
  public static boolean halts(String javaProgram) {
    // ...do some super-clever analysis...
    // return true if javaProgram halts
    // return false if javaProgram does not
  }
}

class Main {
  public static void Q() {
     String p_Q = ???;   // string representing Q
     if (HaltDetector.halts(p_Q)) {
       while (true) {}  // infinite loop!
     }
  }
}
```

What happens when we run Q?

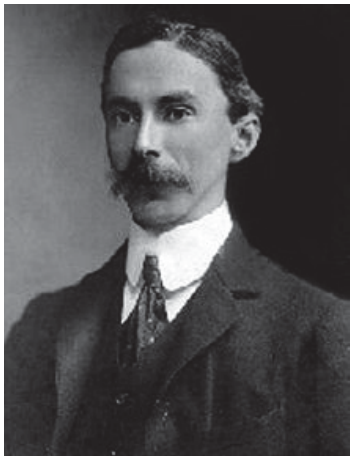
```
public static void Q() {
      String p_Q = ???;    // string representing Q
      if (HaltDetector.halts(p_Q)) {
        while (true) {}  // infinite loop!
      }
  }
```

if HaltDetector.halts(p_Q) ⇒ true then Q ⇒ infinite loop

if HaltDetector.halts(p_Q) ⇒ false then Q ⇒ halts

Contradiction!

- Russell's Paradox (1901)
- Gödel's Incompleteness Theorem (1931)
- Both rely on *self reference*

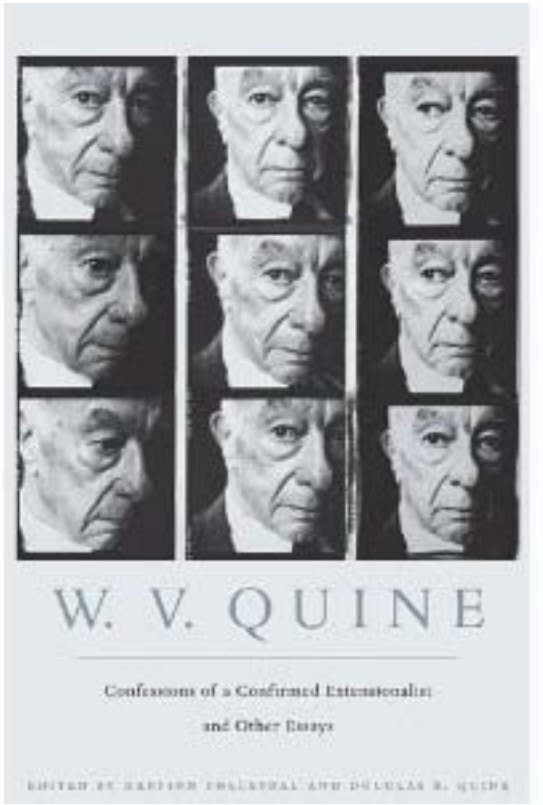


Bertrand Russell, 1901

Kurt Gödel, 1931

Potential Hole in the Proof

- What about the ??? in the program Q?
- It is supposed to be a String representing the program Q itself.
- How can that be possible?
- Answer: code is data!
  - And there's more than one representation for the same data.
- See Quine.java



Profound Consequences

- The "halting problem" is *undecidable*
  - *There are problems that cannot be solved by a computer program!*

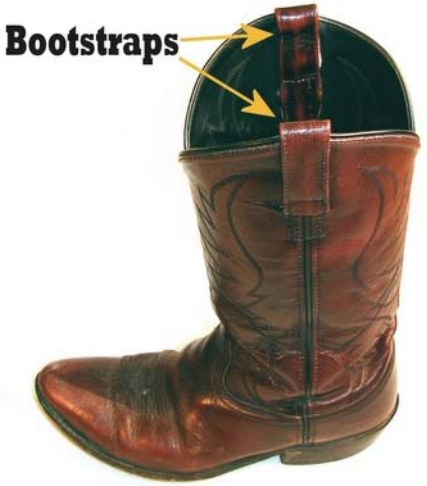
- Rice's Theorem:
  - Every "interesting" property about computer programs is undecidable!
  - You can't test two functions for "equality"

- You can't write a perfect virus detector!
  (whether a program is a virus is certainly interesting)
  1. virus detector might go into an infinite loop
  2. it gives you false positives (i.e. says something is a virus when it isn't)
  3. it gives you false negatives (i.e. it says a program is not a virus when it is)

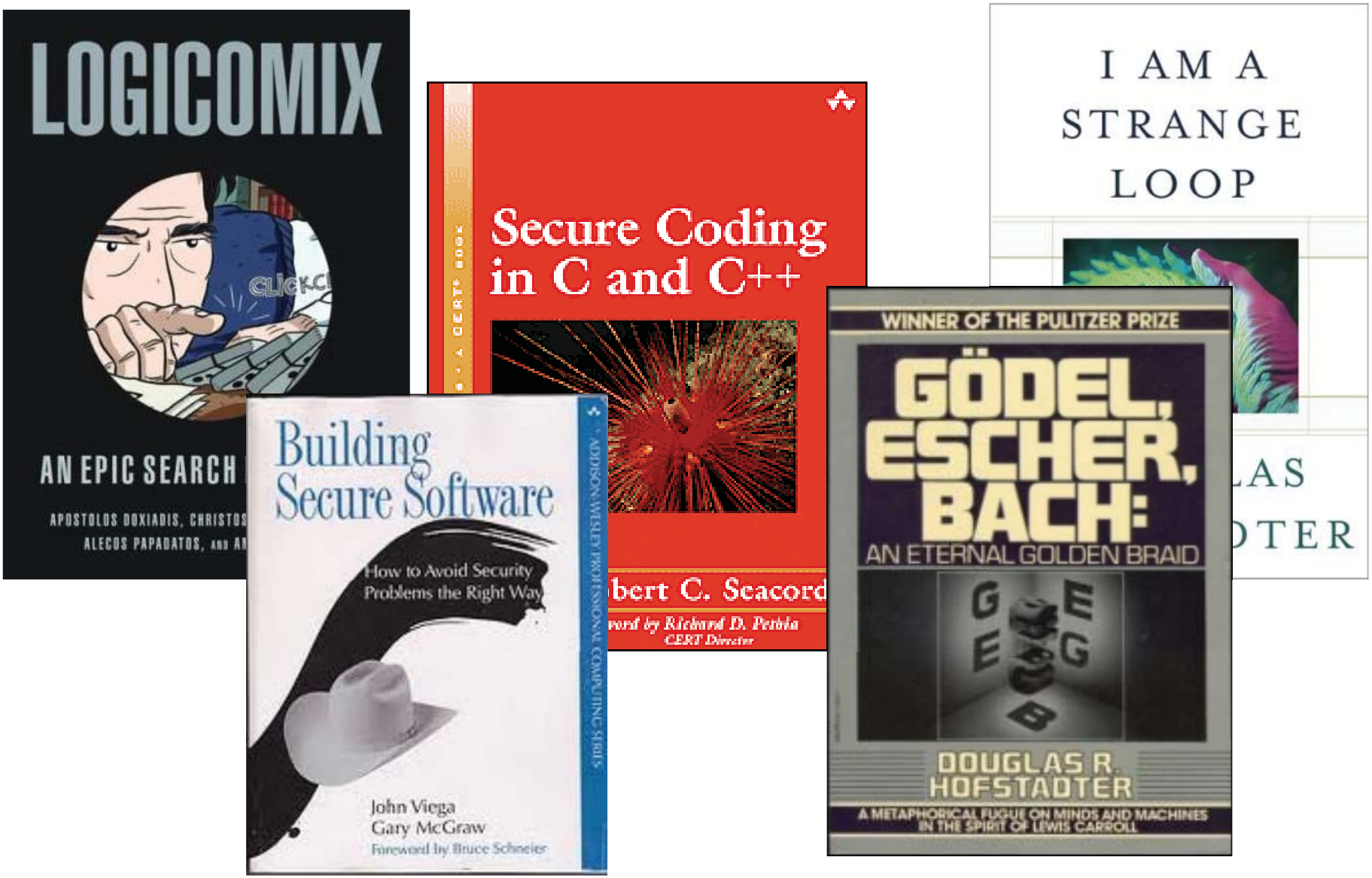
- Also: You can't write a perfect autograder!
  (whether a program is correct is certainly interesting)

Recommended Courses

- Programs that manipulate Programs
  - CIS 341: Compilers and interpreters
- Malware
  - CIS 331: Intro to Networks and Security
- Undecidability
  - CIS 262: Automata, Computability and Complexity



Recommended Reading



Programming Languages and Techniques (CIS120)

Lecture 34

November 27, 2017

Swing I: Drawing and Event Handling

Chapter 29

Announcements

- HW09: Game project
- Checkpoint: Describe your game, what features you will use.
  - DUE: Tomorrow night at 11:59pm
  - Submission via Gradescope (now enabled!)

- Game Project
  - Due: MONDAY, December 11th at midnight
  - No late days!

Swing

Java's GUI library

Why study GUIs (yet again)?

- Most common example of *event-based programming*
- Heavy and effective use of OO inheritance
- Case study in library organization
  - and some advanced Java features
- Ideas applicable everywhere:
  - Web apps
  - Mobile apps
  - Desktop apps
- Fun!

Terminology overview

	GUI (OCaml)	Swing
Graphics Context	`Gctx.gctx`	`Graphics`
Widget type	`Widget.widget`	`JComponent`

Basic Widgets	`button`
	`label`
	`checkbox`
		`JButton`
		`JLabel`
		`JCheckBox`

Container Widgets	`hpair, vpair`	`JPanel, Layouts`

Events	`event`	`ActionEvent`
		`MouseEvent`
		`KeyEvent`

Event Listener	`mouse_listener`
	`mouseclick_listener`
	(any function of type `event -> unit`)
		`ActionListener`
		`MouseListener`
		`KeyListener`

Swing practicalities

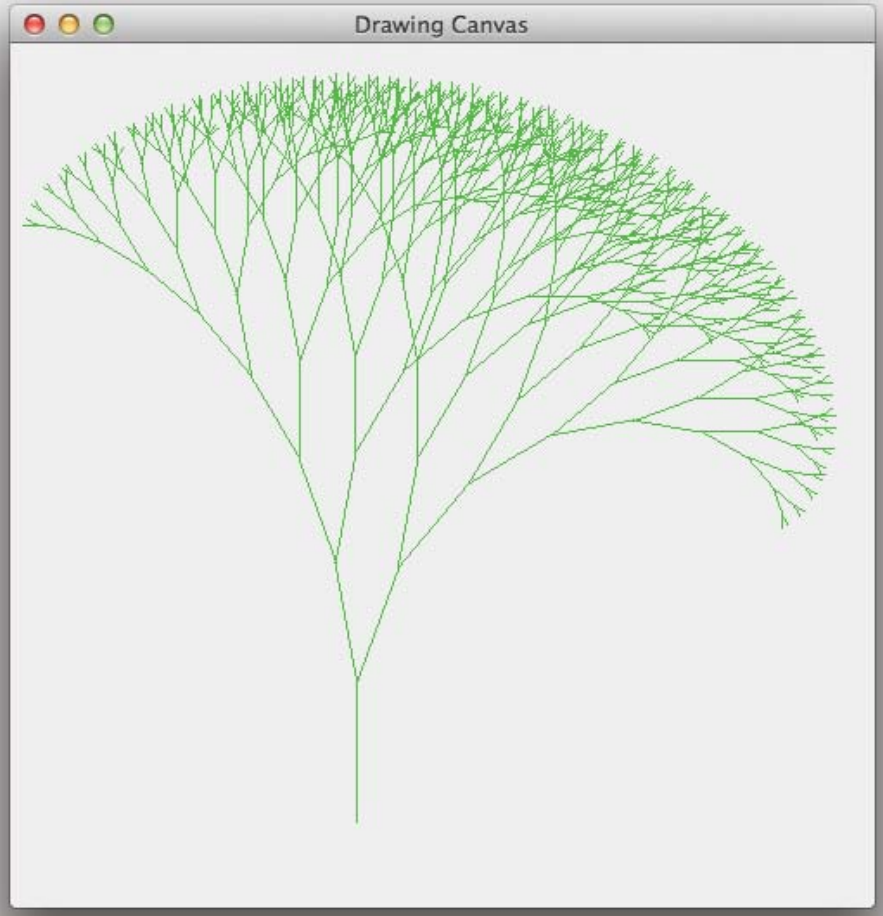
- Java library for GUI development
  - javax.swing.*
- Built on existing library: AWT
  - java.awt.*
  - When there are two versions of something, use Swing's. (e.g., java.awt.Button vs. javax.swing.JButton)
    - The "Jxxx" version is usually the one you want, rather than "xxx".
- Portable
  - Communicates with underlying OS's native window system
  - Same Java program looks appropriately different when run on PC, Linux, and Mac

Simple Drawing

DrawingCanvas.java

DrawingCanvasMain.java

Fractal Drawing Demo



Recursive function for drawing

```
private static void fractal(Graphics gc, int x, int y,
          double angle, double len) {

   if (len > 1) {
     double af = (angle * Math.PI) / 180.0;
     int nx = x + (int)(len * Math.cos(af));
     int ny = y + (int)(len * Math.sin(af));

     gc.drawLine(x, y, nx, ny);

     fractal(gc, nx, ny, angle + 20, len - 8);
     fractal(gc, nx, ny, angle - 10, len - 8);
   }
}
```

How do we draw a picture?

- In the OCaml GUI HW, we created widgets whose `repaint` function used the graphics context to draw an image

```
let w_draw : widget =
{
   repaint = (fun (gc:gctx) ->
                fractal (with_color gc green)
                        200 450 270 80) ;

   size    = (fun () -> (200,200));

   handle  = (fun () -> ())
}
```

- In Swing, we *extend* from class JComponent …

Fundamental class: JComponent

- Analogue to widget type from GUI project
  - (*Terminology*: widget == JComponent)
- Subclasses *override* methods
  - paintComponent (like repaint, displays the component)
  - getPreferredSize (like size, calculates the size of the component)
  - Events handled by listeners (don't need to use overriding…)
- Much more functionality available
  - minimum/maximum size
  - font
  - foreground/background color
  - borders
  - what is visible
  - many more…

Simple Drawing Component

```
public class DrawingCanvas extends JComponent {

    public void paintComponent(Graphics gc) {
        super.paintComponent(gc);

        // set the pen color to green
        gc.setColor(Color.GREEN);

        // draw a fractal tree
        fractal (gc, 200, 450, 270, 80);
    }

    // get the size of the drawing panel
    public Dimension getPreferredSize() {
        return new Dimension(200,200);
    }
```

How to display this component?

JFrame

- Represents a top-level window
  - Displayed directly by OS (looks different on Mac, PC, etc.)
- Contains JComponents
- Can be moved, resized, iconified, closed

```
public void run() {
    JFrame frame = new JFrame("Tree");

    // set the content of the window to be the drawing
    frame.getContentPane().add(new DrawingCanvas());

    // make sure the application exits when the frame closes
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // resize the frame based on the size of the panel
    frame.pack();

    // show the frame
    frame.setVisible(true);
}
```

User Interaction

Start Simple: Lightswitch Revisited

Task: Program an application that displays a button. When the button is pressed, it toggles a “lightbulb” on and off.

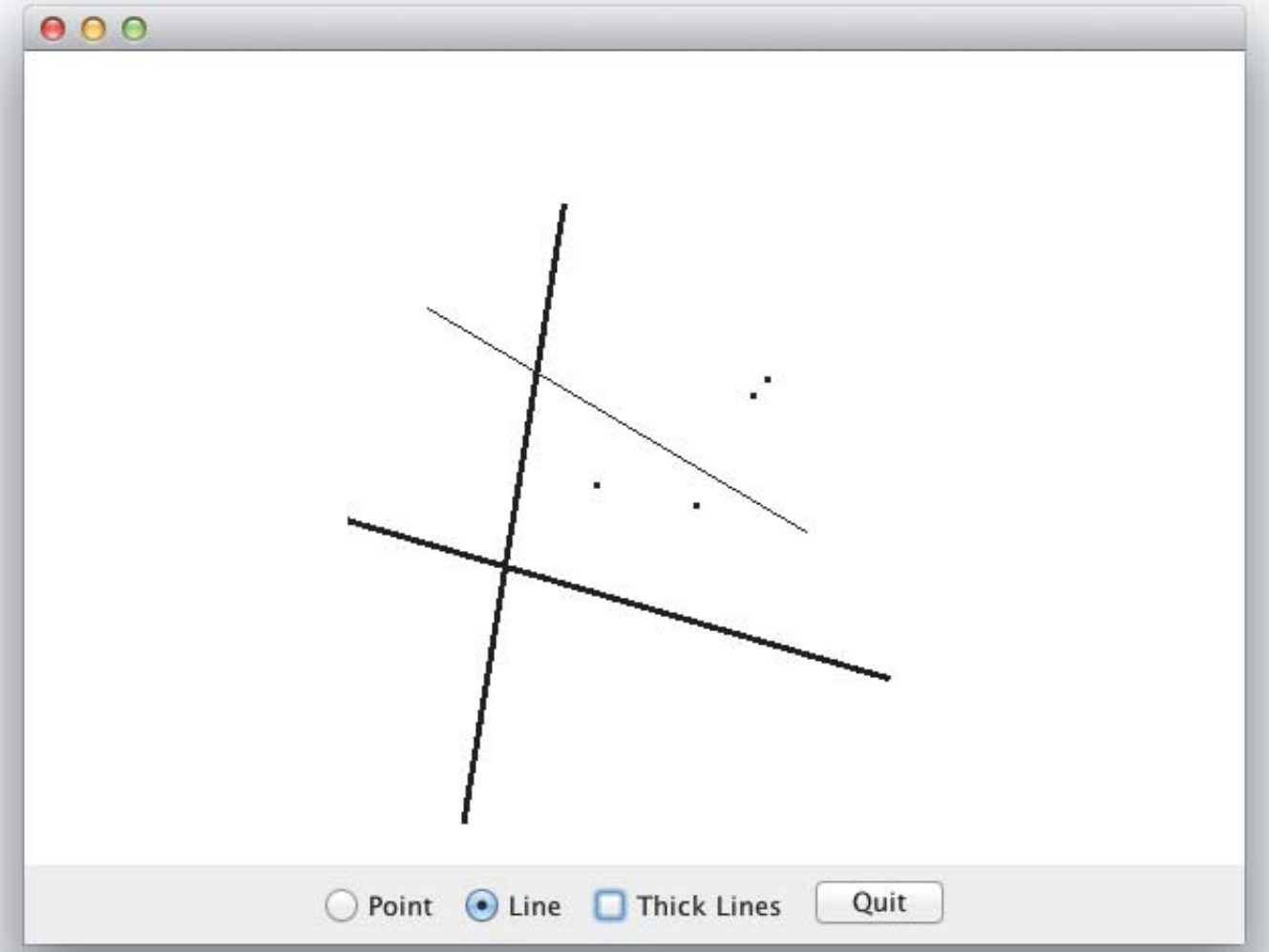
Key idea: use a ButtonListener to toggle the state of the "lightbulb"

OnOffDemo

The Lightswitch GUI program in Swing.

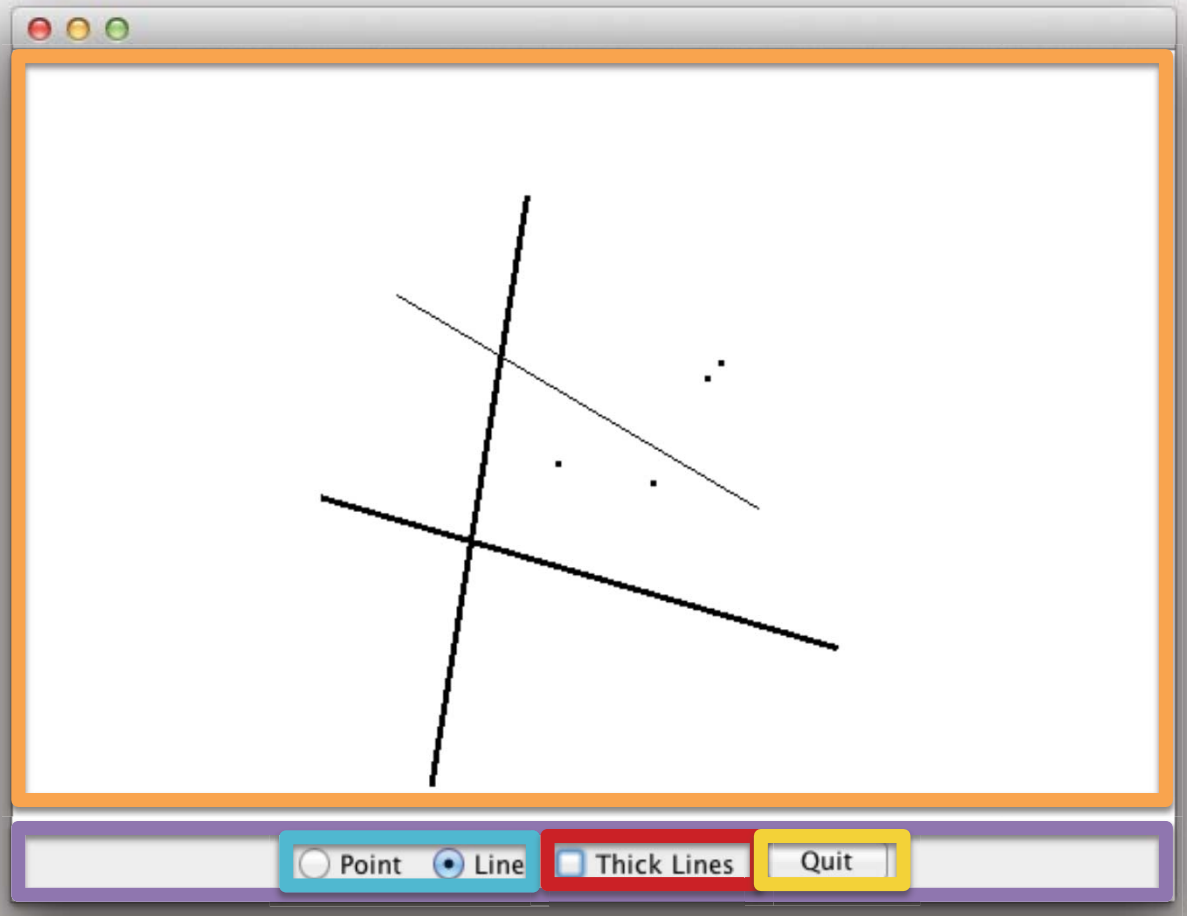
Layout Demo

Layout.java



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)

Inner Classes

Inner Classes

- Useful in situations where two objects require “deep access” to each other’s internals
- Replaces tangled workarounds like “owner object”
  - Solution with inner classes is easier to read
  - No need to allow public access to instance variables of outer class
- Also called “dynamic nested classes”

Basic Example

Key idea: Classes can be *members* of other classes…

```
class Outer {
  private int outerVar;
  public Outer () {
    outerVar = 6;
  }
  public class Inner {
    private int innerVar;
    public Inner(int z) {
      innerVar = outerVar + z;
    }
    public int getInnerVar() {
      return innerVar;
    }
  }
}
```

Name of this class is Outer.Inner (which is also the static type of objects that this class creates)

Inner class can refer to fields bound in outer class

Constructing Inner Class Objects

Based on your understanding of the Java object model, which of the following make sense as ways to construct an object of an inner class type?

1. Outer.Inner obj = new Outer.Inner(3)
2. Outer.Inner obj = (new Outer()).new Inner(3);
3. Outer.Inner obj = new Inner(3);
4. Outer.Inner obj = Outer.Inner.new (3)
5. All of the above
6. None of the above
7. Just (1) and (2)

Answer: 2

Programming Languages and Techniques (CIS120)

Lecture 35

November 29, 2017

Swing II: Building GUIs

Inner Classes

Chapter 29

Announcements

- Game Project Complete Code
  - Due: December 11th
  - *NO LATE SUBMISSIONS*
  - *Grading will be by demo. Stay tuned for details!*

Swing

Java's GUI library

OnOff.java revisited

```
public void run() {
    // ... other code omitted

    LightBulb bulb = new LightBulb();
    panel.add(bulb);

    JButton button = new JButton("On/Off");
    panel.add(button);

    button.addActionListener(new ButtonListener(bulb));

    // ... other code omitted
}
```

Too much boilerplate code…

- ButtonListener really only needs to do bulb.flip()
- But we need all this boilerplate code to build the class
- Often we will only instantiate *one* instance of a given Listener class in a GUI

```
class ButtonListener implements ActionListener {
    private LightBulb bulb;
    public ButtonListener (LightBulb b) {
    bulb = b;
    }
    @Override
    public void actionPerformed(ActionEvent e) {
        bulb.flip();
        bulb.repaint();
    }
}
```

Inner Classes

Inner Classes

- Useful in situations where two objects require “deep access” to each other’s internals
- Replaces tangled workarounds like “owner object”
  - Solution with inner classes is easier to read
  - No need to allow public access to instance variables of outer class

- Also called “dynamic nested classes”

Basic Example

Key idea: Classes can be *members* of other classes…

```
class Outer {
  private int outerVar;
  public Outer () {
    outerVar = 6;
  }
  public class Inner {
    private int innerVar;
    public Inner(int z) {
      innerVar = outerVar + z;
    }
    public int getInnerVar() {
      return innerVar;
    }
  }
}
```

Name of this class is Outer.Inner (which is also the static type of objects that this class creates)

Reference from inner class to instance variable bound in outer class

Constructing Inner Class Objects

Based on your understanding of the Java object model, which of the following make sense as ways to construct an object of an inner class type?

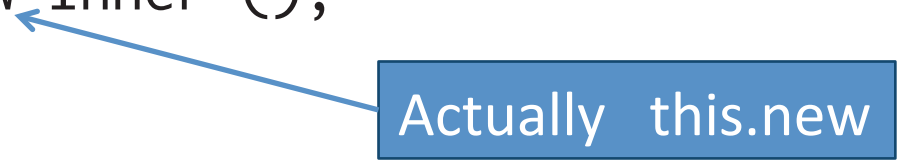
1. `Outer.Inner obj =  new Outer.Inner()`
2. `Outer.Inner obj = (new Outer()).new Inner();`
3. `Outer.Inner obj = new Inner();`
4. `Outer.Inner obj = Outer.Inner.new ()`

Answer: 2 – the inner class instances can refer to non-static fields of the outer class (even in the constructor), so the invocation of "new" must be relative to an existing instance of the Outer class.

Object Creation

- Inner classes can refer to the instance variables and methods of the outer class
- Inner class instances usually created by the methods/constructors of the outer class

```
public Outer () {
    Inner b = new Inner ();
}
```



- Inner class instances *cannot* be created independently of a containing class instance.

```
Outer.Inner b = new Outer.Inner()
```

```
Outer a = new Outer();
Outer.Inner b = a.new Inner();
```

```
Outer.Inner b = (new Outer()).new Inner();
```

Anonymous Inner Classes

- Define a class and create an object from it all at once, inside a method

```
quit.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }
});
```

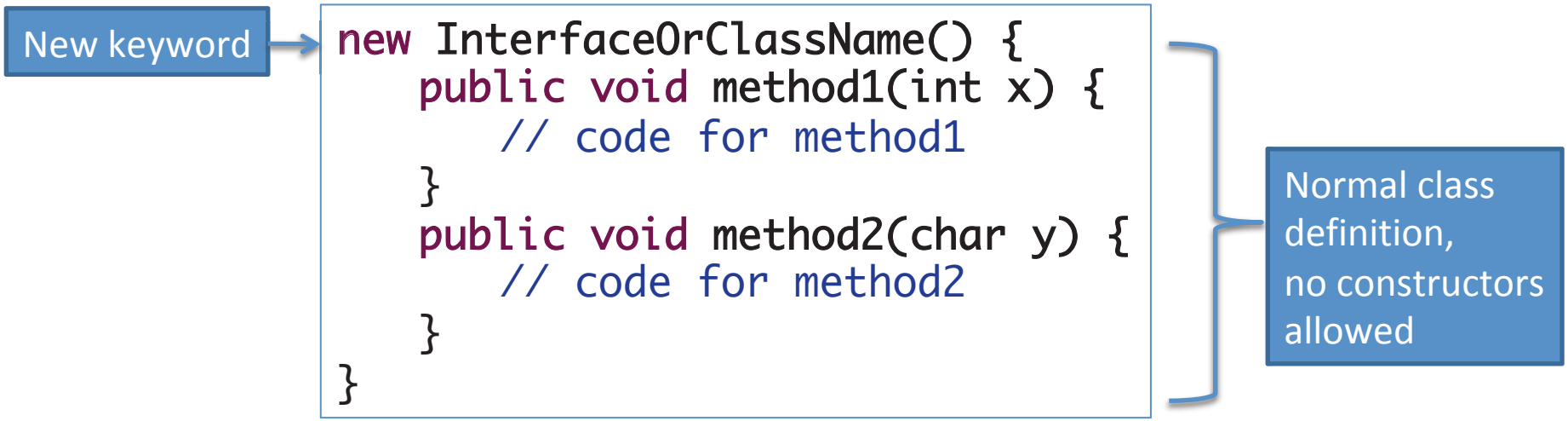
Puts button action right with button definition

```
line.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        shapes.add(new Line(…));
        canvas.repaint();
    }
});
```

Can access fields and methods of outer class, as well as final local variables

Anonymous Inner class

- New *expression* form: define a class and create an object from it all at once



Static type of the expression is the Interface/superclass used to create it

Dynamic class of the created object is anonymous! Can't refer to it.

Like first-class functions

- Anonymous inner classes are roughly* analogous to Ocaml's first-class functions
- Both create "delayed computation" that can be stored in a data structure and run later
  - Code stored by the event / action listener
  - Code only runs when the button is pressed
  - Could run once, many times, or not at all
- Both sorts of computation can refer to variables in the current scope
  - OCaml: Any available variable
  - Java: only instance variables (fields) of enclosing object and local variables marked `final`

*see next slide

“Anonymous Lambdas”

- Java 8 introduced "lambda expressions" which are simplified syntax for anonymous classes with "functional interfaces" – interfaces with just one method

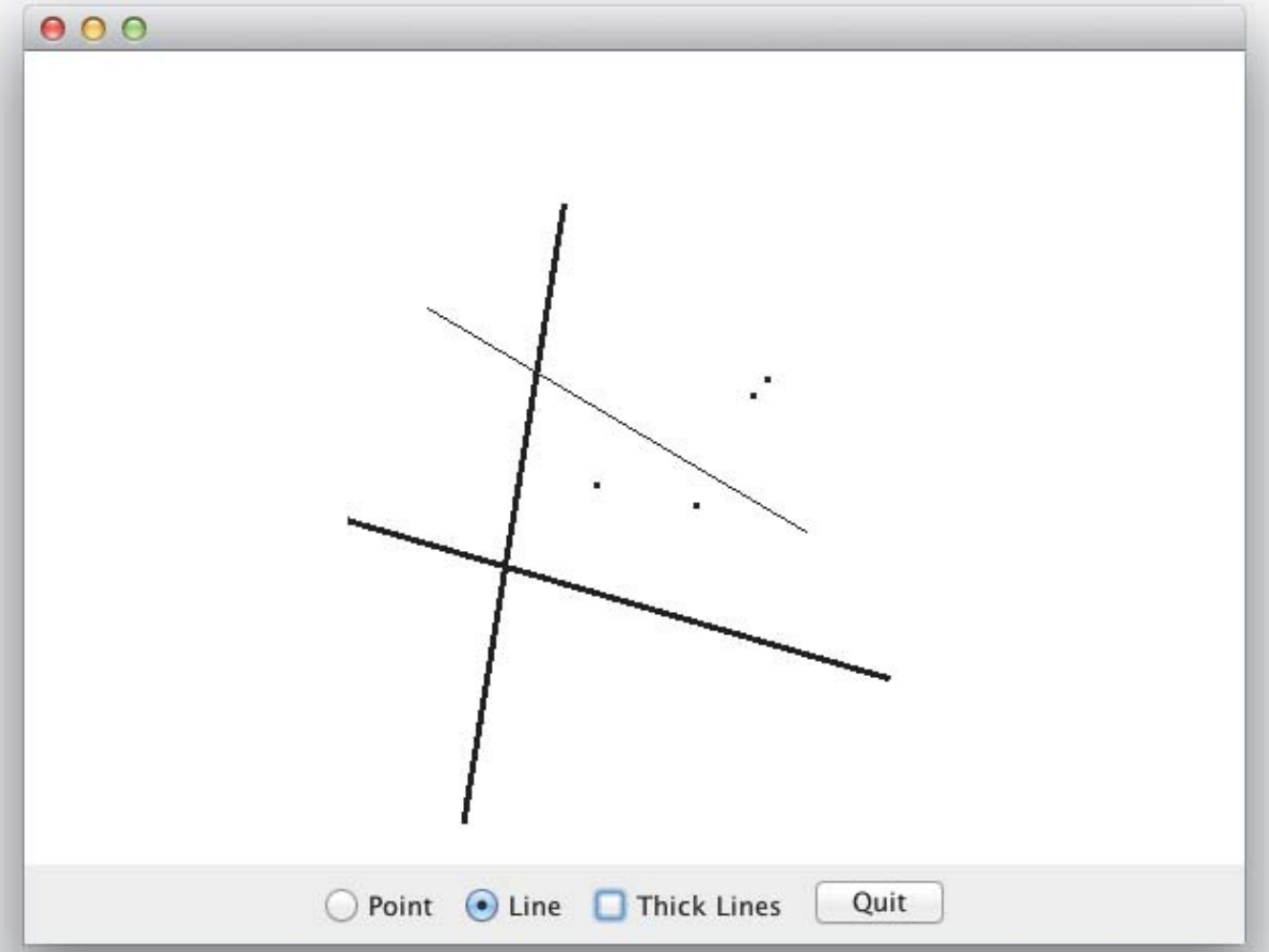
```
persons.forEach(p -> p.setLastName("Doe"))
```

- Wait until next week for more details…

Paint Revisited

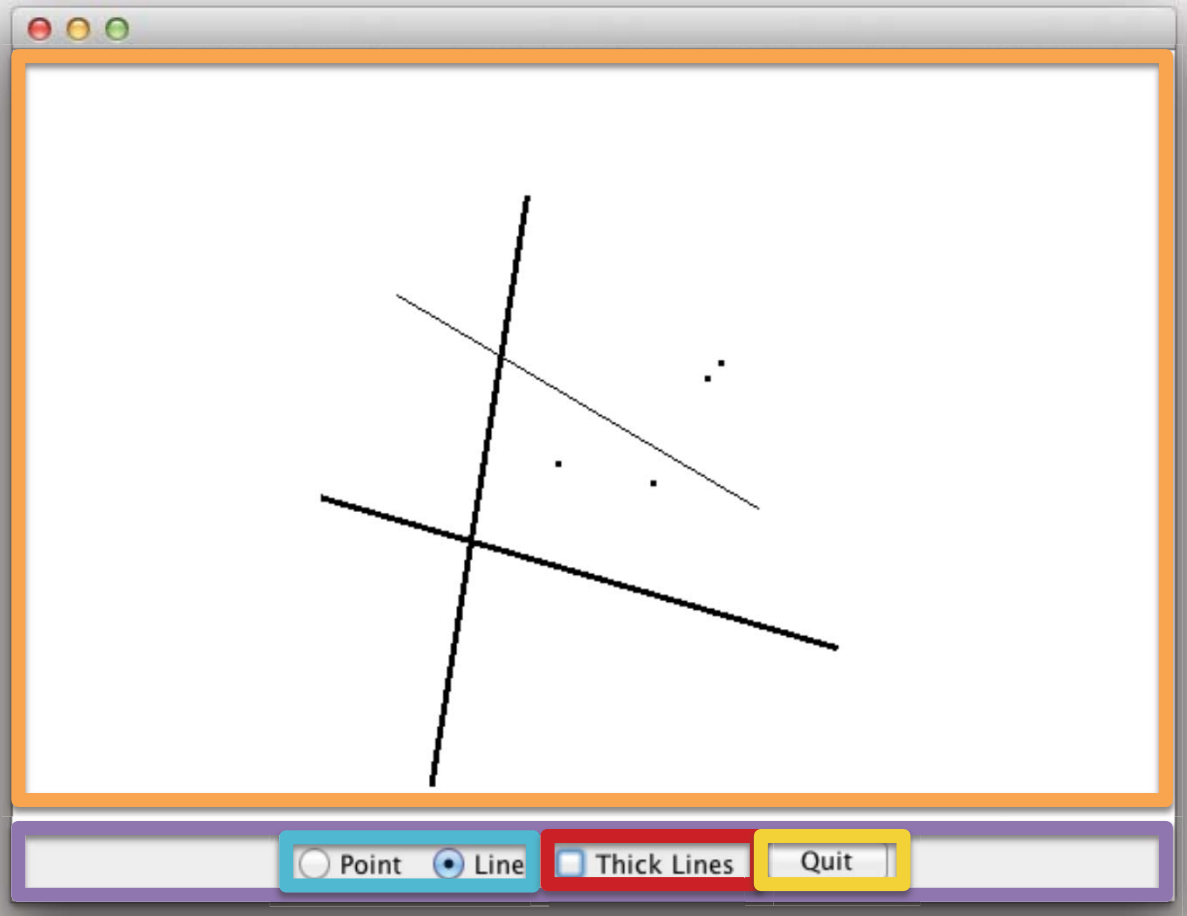
Using Anonymous Inner Classes

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)

Paint demo

(See PaintA.java … PaintE.java)

Programming Languages and Techniques (CIS120)

Lecture 36

December 1, 2017

Mushroom of Doom

Model / View / Controller

Chapter 31

Announcements

- Game Project Complete Code
  - Due: Monday, December 11th
  - *NO LATE SUBMISSIONS PERMITTED*

- *Final Exam*
  - *Friday, December 15th, 6:00-8:00PM*
  - *Locations: TBA*
  - makeup exam
    form available on the course web site

Paint Revisited

Using Anonymous Inner Classes

Refactoring for OO Design

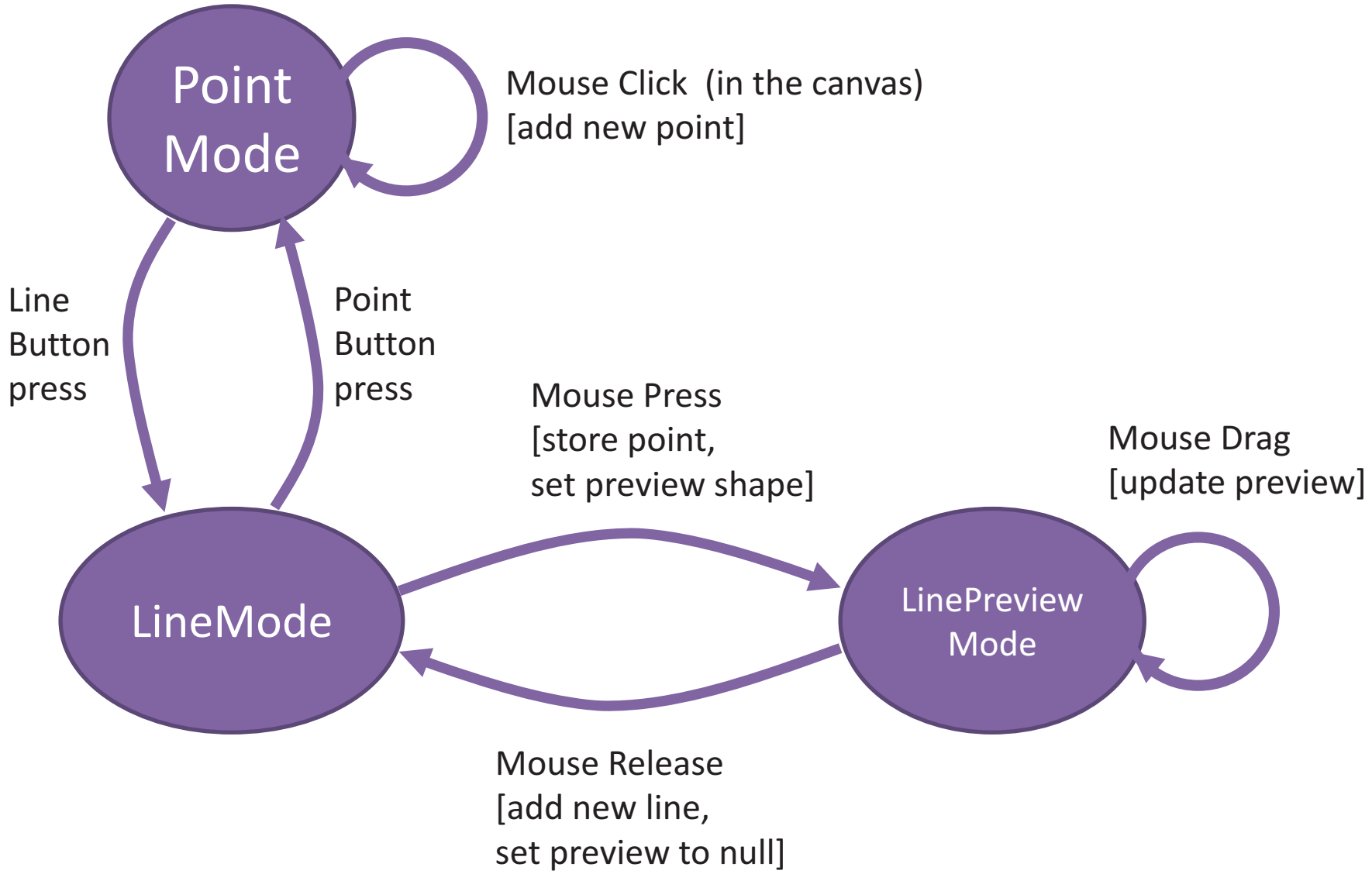
(See PaintA.java … PaintE.java)

Adapters

MouseAdapter

KeyAdapter

Mouse Interaction in Paint



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:

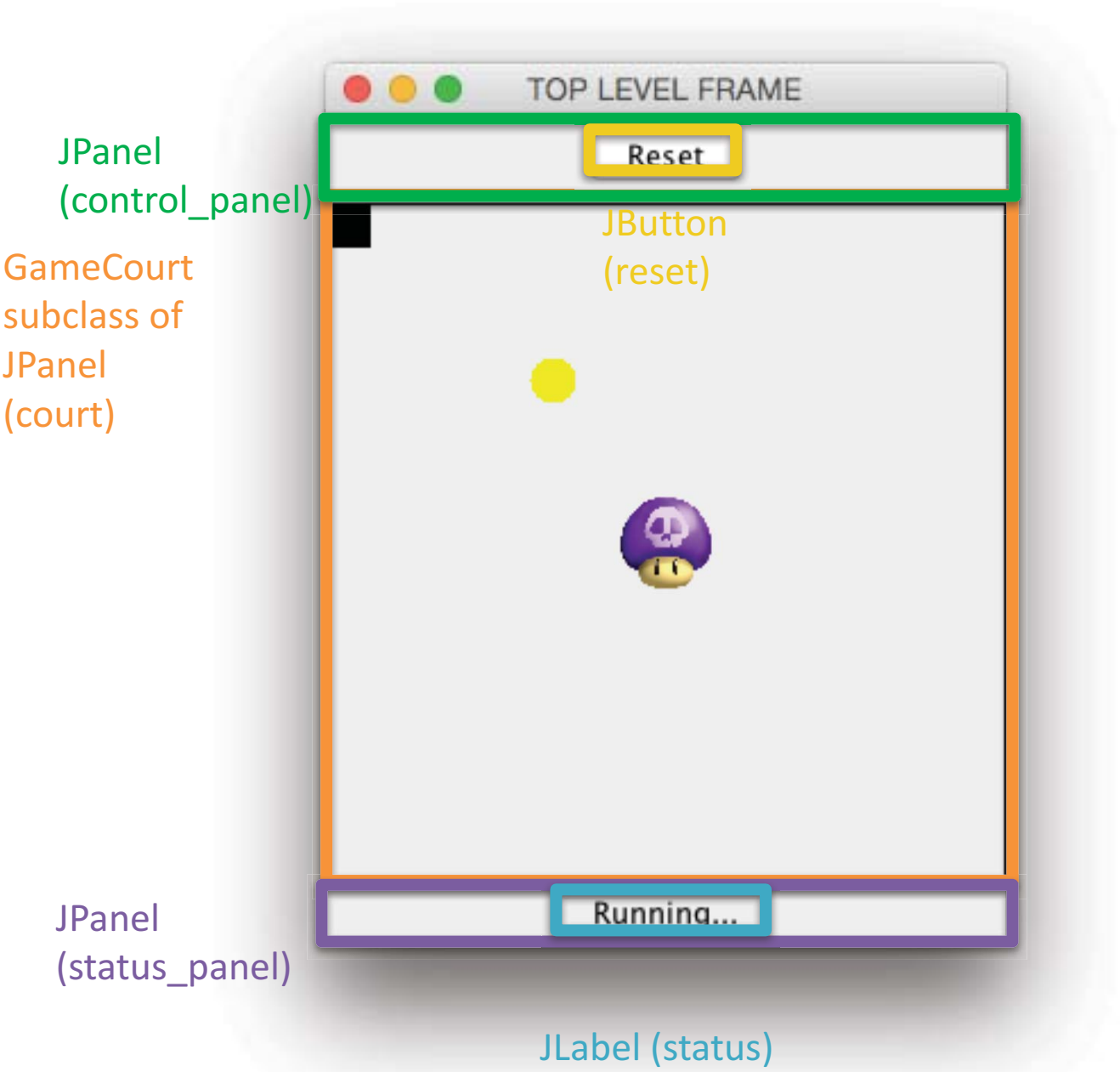
- Swing provides a collection of abstract event adapter classes
- These adapter classes implement listener interfaces with empty, do-nothing methods
- To implement a listener class, we extend an adapter class and override just the methods we need

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

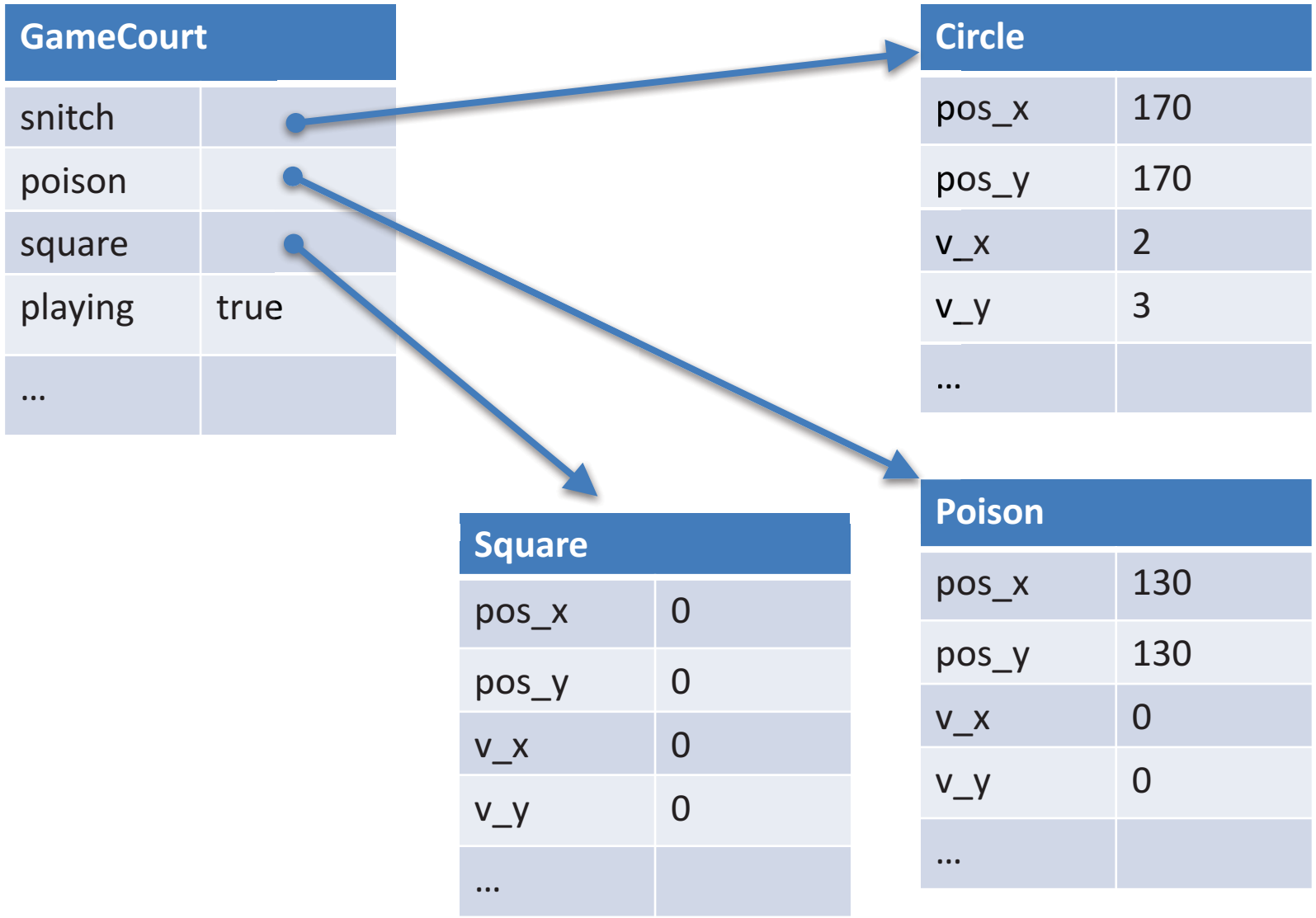
Mushroom of Doom

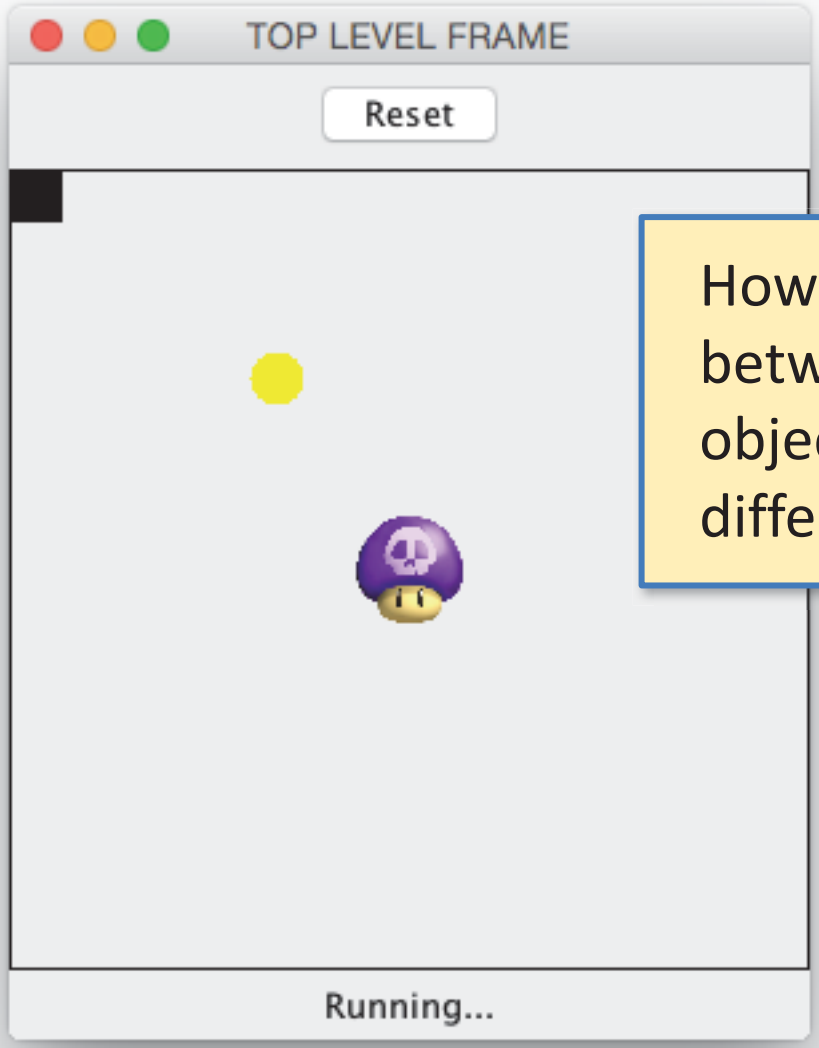
How do we put Swing components together to make a complete game?


TOP LEVEL FRAME
Reset
Running...


TOP LEVEL FRAME
Reset
JPanel
(control_panel)
JButton
(reset)
GameCourt
subclass of
JPanel
(court)
Running...
JPanel
(status_panel)
JLabel (status)

Game State


GameCourt
snitch
poison
square
playing
true
…
Circle
pos_x
170
pos_y
170
v_x
2
v_y
3
…
Square
pos_x
0
pos_y
0
v_x
0
v_y
0
…
Poison
pos_x
130
pos_y
130
v_x
0
v_y
0
…



How can we share code between the game objects, but show them differently?

Updating the Game State: timer

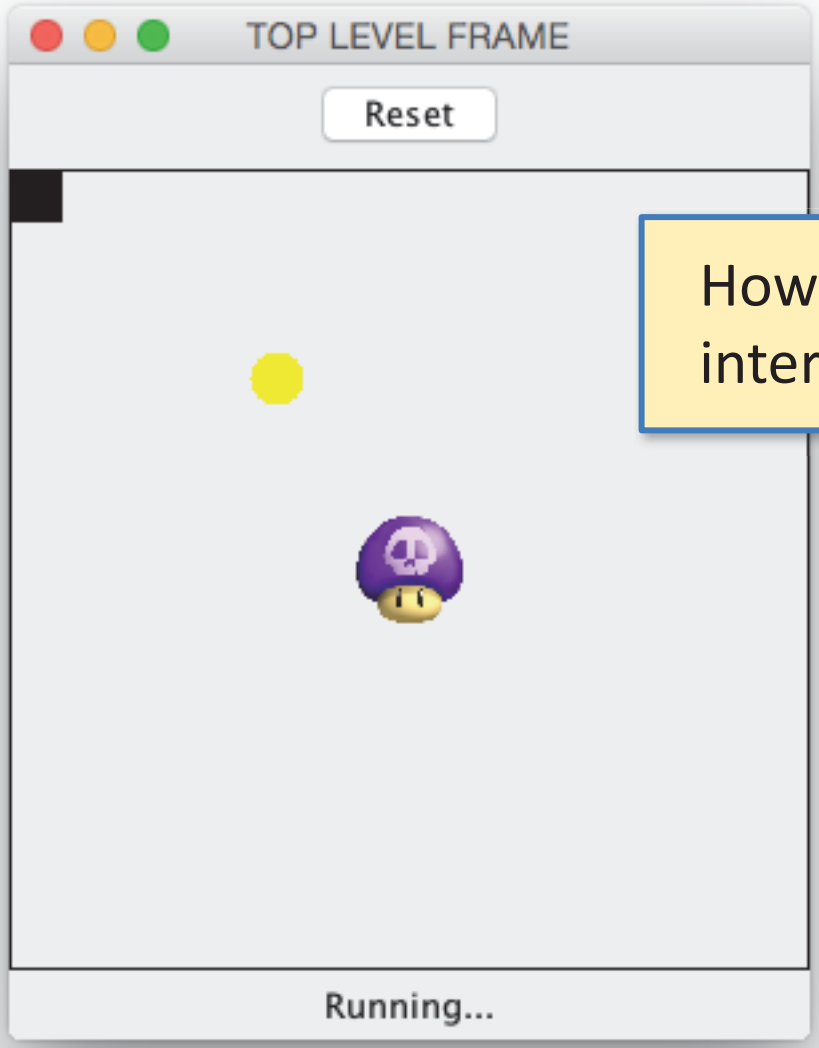
```
void tick() {
  if (playing) {
    square.move();
    snitch.move();
    snitch.bounce(snitch.hitWall());      // bounce off walls...
    snitch.bounce(snitch.hitObj(poison)); // ...and the mushroom

    if (square.intersects(poison)) {
       playing = false;
       status.setText("You lose!");
    } else if (square.intersects(snitch)) {
       playing = false;
       status.setText("You win!");
    }
    repaint();
  }
}
```

Updating the Game State: keyboard

```
setFocusable(true);
addKeyListener(new KeyAdapter() {
  public void keyPressed(KeyEvent e) {
    if (e.getKeyCode() == KeyEvent.VK_LEFT)
        square.v_x = -SQUARE_VELOCITY;
    else if (e.getKeyCode() == KeyEvent.VK_RIGHT)
        square.v_x = SQUARE_VELOCITY;
    else if (e.getKeyCode() == KeyEvent.VK_DOWN)
        square.v_y = SQUARE_VELOCITY;
    else if (e.getKeyCode() == KeyEvent.VK_UP)
        square.v_y = -SQUARE_VELOCITY;
    }

    public void keyReleased(KeyEvent e) {
        square.v_x = 0;
        square.v_y = 0;
    }
});
```

Make square's velocity nonzero when a key is pressed

Make square's velocity zero when a key is released

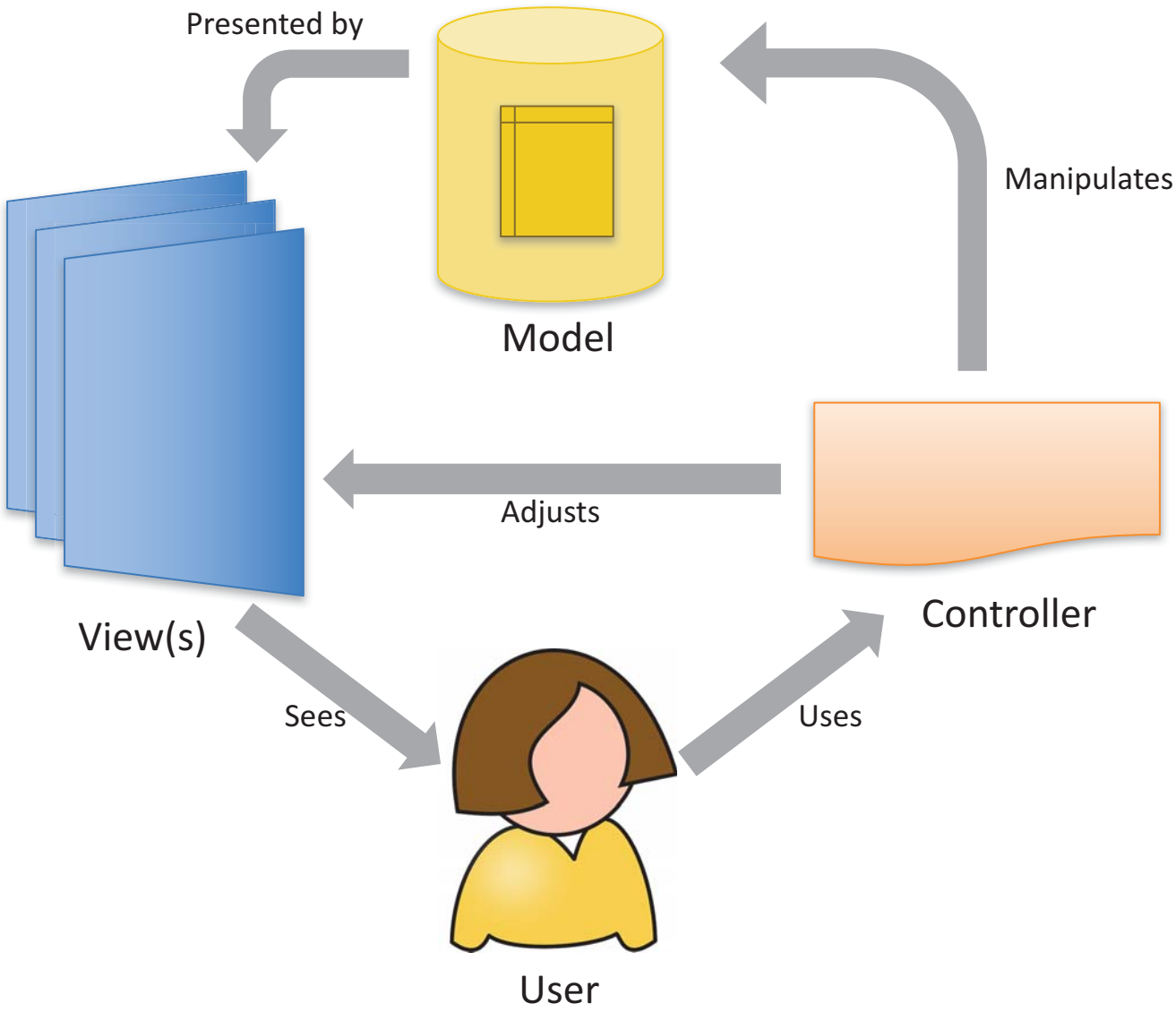


How does the user interact with the game?

1. Clicking Reset button restarts the game
2. Holding arrow key makes square move
3. Releasing key makes square stop

Model View Controller Design Pattern

MVC Pattern


Presented by
Model
Manipulates
Adjusts
View(s)
Controller
Sees
Uses
User

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)

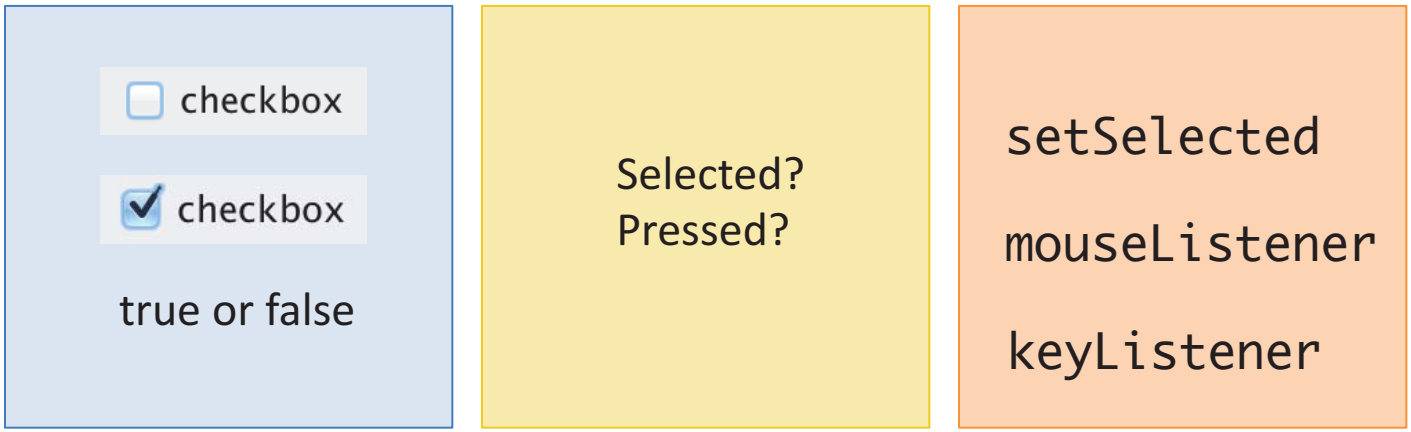
View

- 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

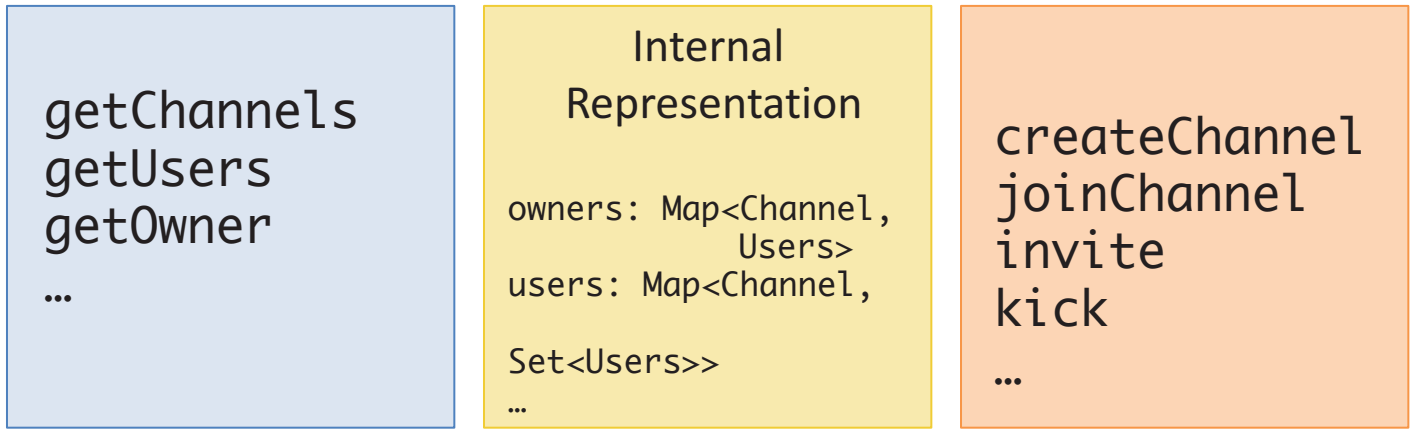
Example: CheckBox




```
Class JToggleButton.ToggleButtonModel

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

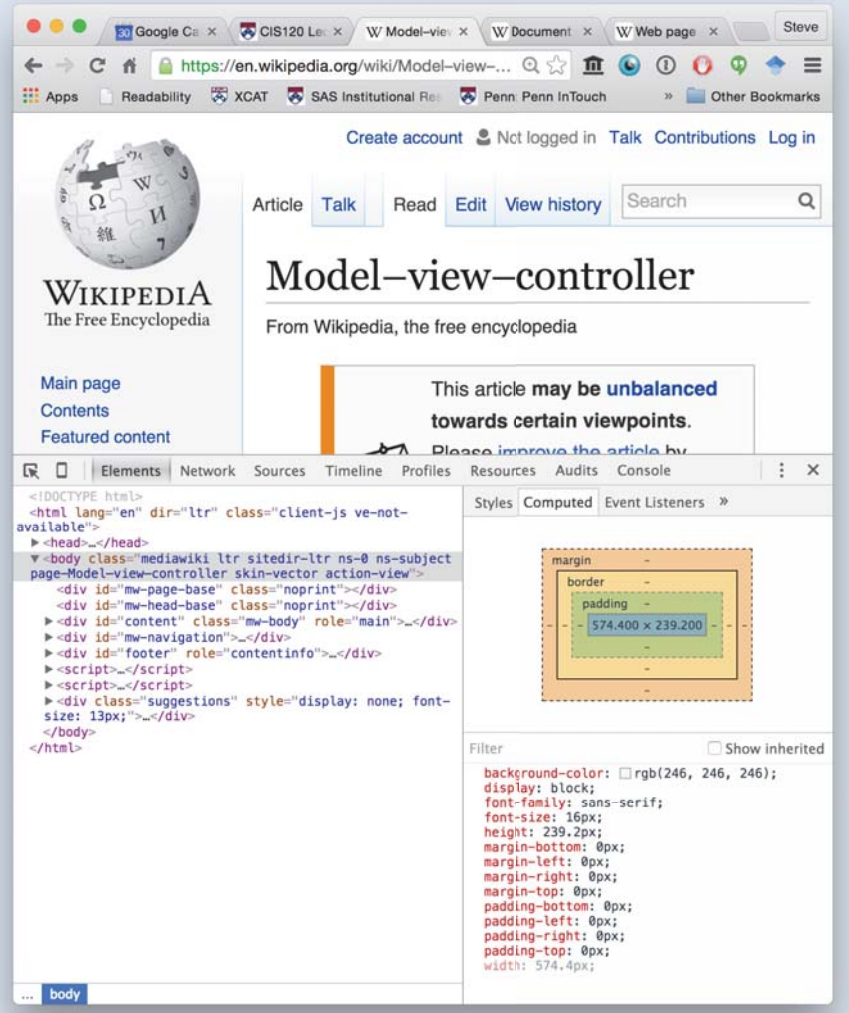
Checks if the button is selected.
Sets the pressed state of the button.
Sets the selected state of the button.

Example: Chat Server



ServerModel

Example: Web Pages



Views

Internal Representation: DOM (Document Object Model)

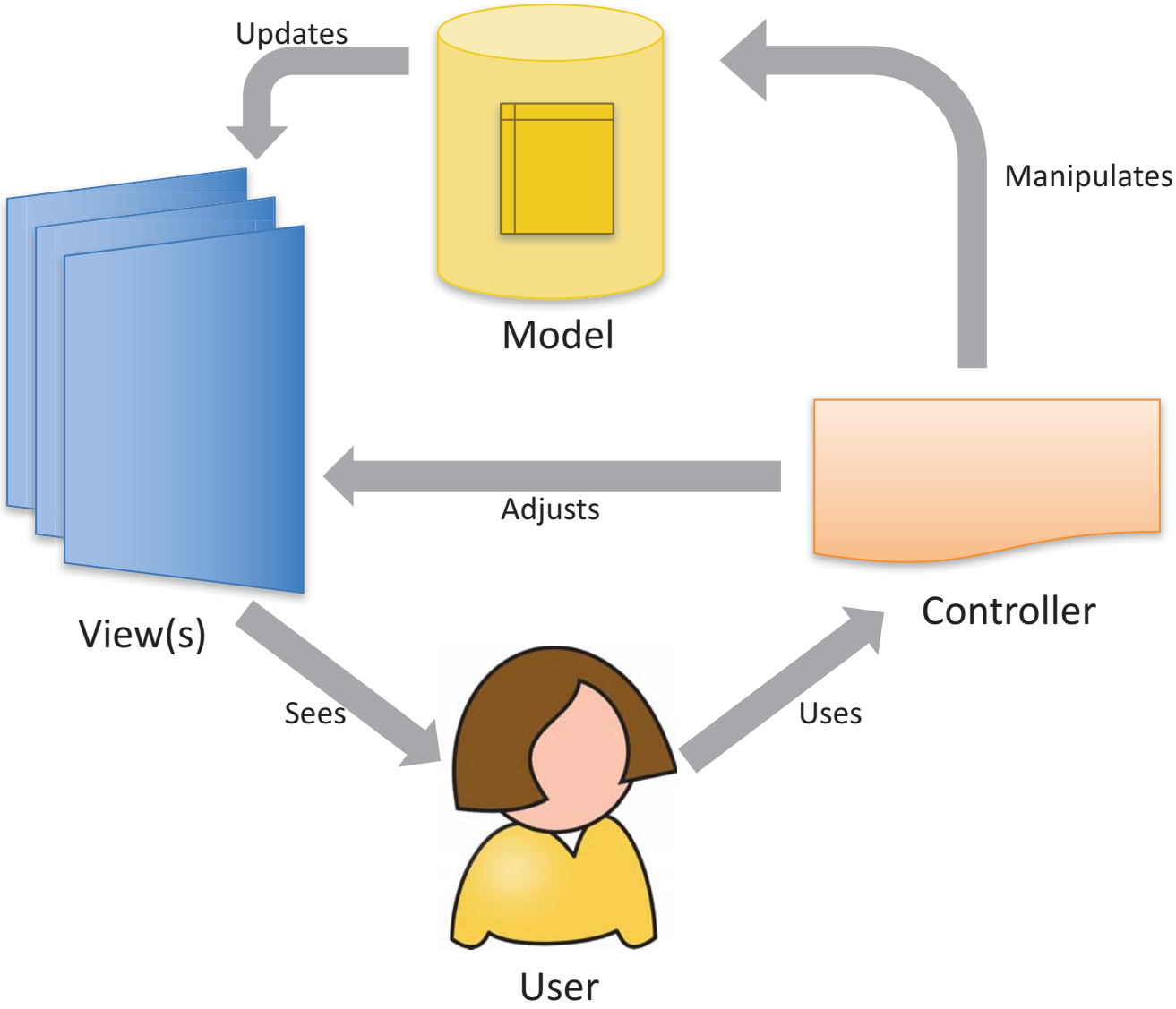
Model

JavaScript API

```
document.
addEventListener()
```

Controllers

MVC Pattern


Updates
Model
Manipulates
Adjusts
View(s)
Controller
Sees
Uses
User

MVC Benefits?

- Decouples important "model state" from how that state is presented and manipulated
  - Suggests where to insert interfaces in the design
  - Makes the model testable independent of the GUI
- Multiple views
  - e.g. from two different angles, or for multiple different users
- Multiple controllers
  - e.g. mouse vs. keyboard interaction

MVC Variations

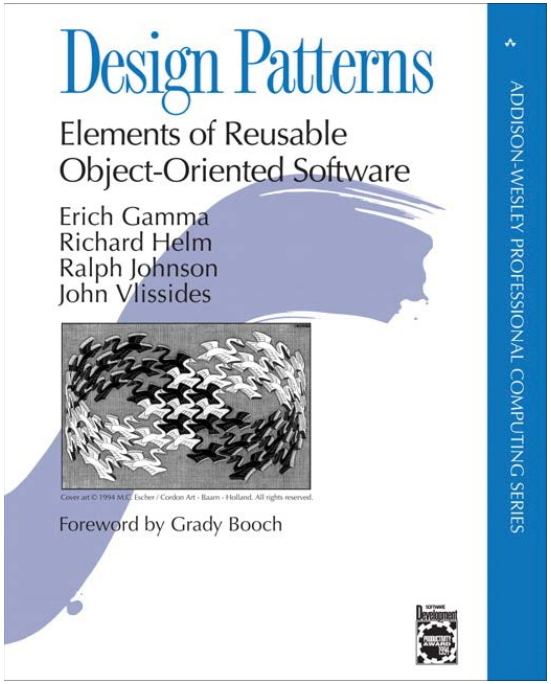
- Many variations on MVC pattern

- 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

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 Broadcast class from 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

Programming Languages and Techniques (CIS120)

Lecture 37

December 4, 2017

Advanced Java Miscellany (Hashing)

Announcements

- HW09: Game of your own
  - Due Monday, December 11th at 11:59pm
  - No late submissions!

Final Exam:

- Friday, December 15th 6:00-8:00 PM
- 3 Locations:
  - Towne 100 Last Names A – G
  - Stiteler B6 Last Names H – Pa
  - Cohen G17 Last Names Pe -- Z

- Mock Exam:

  Wednesday, December 13th 6:00-8:00PM

  Towne 100

Advanced Java Miscellany

- Hashing: HashSets & HashMaps
- Threads & Synchronization
- Garbage Collection
- Java 1.8 Lambdas

We'll touch on these.

- 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 nitty-gritty details:
Java Language Specification
http://docs.oracle.com/javase/specs/

Hash Sets & Hash Maps

array-based implementation of sets and maps

Hash Sets and Maps: The Big Idea

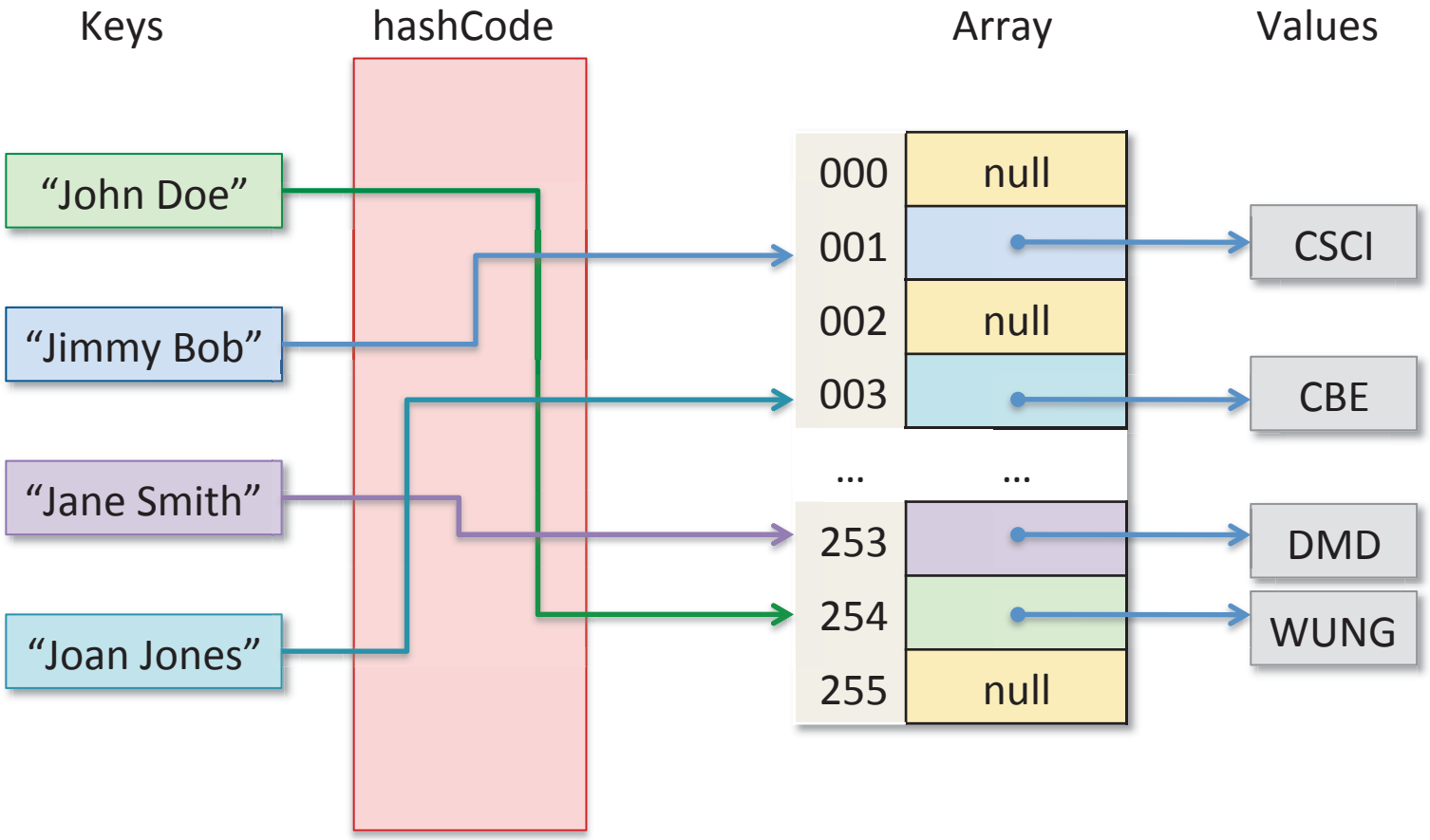
Combine:

- the advantage of arrays:
  - *efficient* random access to its elements
- with the advantage of a map datastructure
  - arbitrary keys (not just integer indices)

How?

- Create an index into an array by *hashing* the data in the key to turn it into an int
  - Java's hashCode method maps key data to ints
  - Generally, the space of keys is much larger than the space of hashes, so, unlike array indices, hashCodes might not be unique

Hash Maps, Pictorially



A schematic HashMap taking Strings (student names) to Undergraduate Majors. The hashCode takes each string name to an integer code, which we then take "mod 256" to get an array index between 0 and 255.

For example, `"John Doe".hashCode()` mod 256 is 254.

Hash Collisions

- Uh Oh: Indices derived via hashing may not be unique!

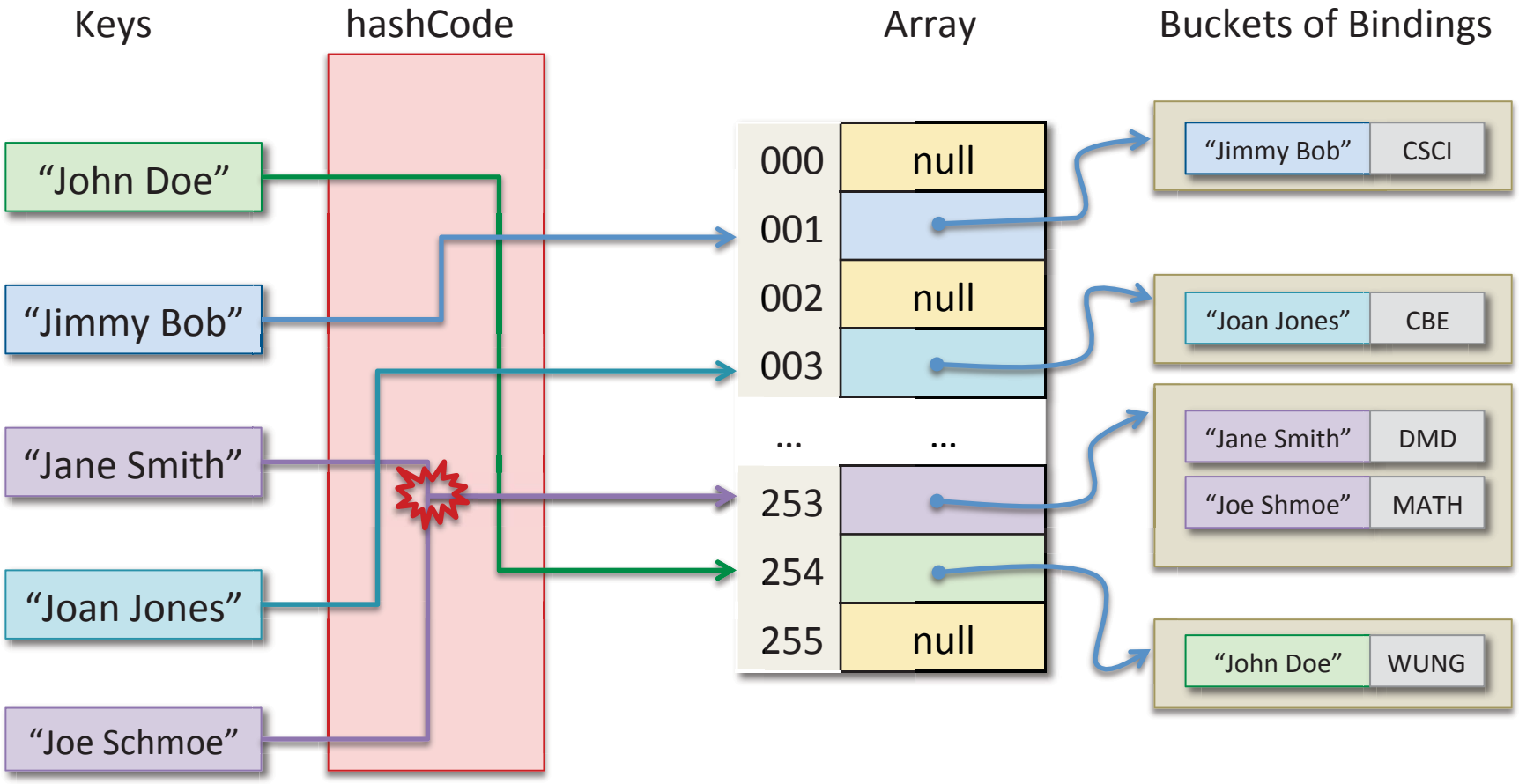
  `"Jane Smith".hashCode() % 256` ➔ 253

  `"Joe Schmoe".hashCode() % 256` ➔ 253

- Good hashCode functions make it *unlikely* that two keys will produce the same hash

- But, it can still sometimes happen that two keys produce the same index – that is, their hashes *collide*

Bucketing and Collisions



Here, "Jane Smith".hashCode() and "Joe Schmoe".hashCode() happen to collide. The *bucket* at the corresponding index of the Hash Map array stores the map data.

Bucketing and Collisions

- Using an array of *buckets*
  - Each bucket stores the mappings for keys that have the same hash.
  - Each bucket is itself a map from keys to values (implemented by a linked list or binary search tree).
  - The buckets can't use hashing to index the values – instead they use key equality (via the key's equals method)

- To look up a key in the Hash Map:
  1. Find the right bucket by indexing the array through the key's hash
  2. Search linearly through the bucket contents to find the value associated with the key

- Not the only solution to the collision problem

Hashing and User-defined Classes

```
public class Point {
    private final int x;
    private final int y;
    public Point(int x, int y) { this.x = x; this.y = y; }
    public int getX() { return x; }
    public int getY() { return y; }
}

// somewhere else…
Map<Point,String> m = new HashMap<Point,String>();
m.put(new Point(1,2), "House");
System.out.println(m.containsKey(new Point(1,2)));
```

What gets printed to the console?

1. true
2. false
3. I have no idea

ANSWER: 2 – hashCode not implemented

HashCode Requirements

Whenever you override `equals` you must also override `hashCode` in a consistent way:

- whenever `o1.equals(o2) == true` you must ensure that `o1.hashCode() == o2.hashCode()`

Why? Because comparing hashes is supposed to be a quick approximation for equality.

- Note: the converse does not have to hold:
  - `o1.hashcode() == o2.hashCode()` does *not* necessarily mean that o1.equals(o2)

Example for Point

```
public class Point {
   @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + x;
        result = prime * result + y;
        return result;
    }
}
```

- Examples:
  - (new Point(1,2)).hashCode() yields 994
  - (new Point(2,1)).hashCode() yields 1024
- Note that equal points (in the sense of `equals`) have the same hashCode
- Why 31? Prime chosen to create more uniform distribution
- Note: Tools (e.g. eclipse) can *generate* this code

Recipe: Computing Hashes

- What is a good recipe for computing hash values for your own classes?
  - intuition: "smear" the data throughout all the bits of the resulting

1. Start with some constant, arbitrary, non-zero int in `result`.
2. For each significant field f of the class (i.e. each field taken into account when computing equals), compute a "sub" hash code `c` for the field:
   - For boolean fields: `(f ? 1 : 0)`
   - For byte, char, int, short: `(int) f`
   - For long: `(int) (f ^ (f >>> 32))`
   - For references: 0 if the reference is null, otherwise use the `hashCode()` of the field.
3. Accumulate those subhashes into the result by doing (for each field's `c`):
   `result = prime * result + c;`
4. return `result`

Hash Map Performance

- Hash Maps can be used to efficiently implement Maps and Sets
  - There are many different strategies for dealing with hash collisions with various time/space tradeoffs
  - Real implementations also dynamically rescale the size of the array (which might require re-computing the bucket contents)
  - See CIS 121 for more info!

- If the hashCode function gives a good (close to uniform) distribution of hashes, the buckets are expected to be small (only one or two elements)
- If the hashCode function gives a bad distribution (e.g. return 0;), the buckets will be large (and performance will be bad)

- Performance depends on workload

NOTE: Terminological Clash

- The word "hash" is also used in *cryptography*
  - SHA-1, SHA-2, SHA-3, MD5, etc.
- All hash functions reduce large objects to short summaries
- Cryptographic hashes have some extra requirements:
  - Are "one way" (i.e. very hard to *invert)*
  - Should only very rarely have collisions
  - Are considerably more expensive to compute than hashCode (so not suitable for hash tables)

- Never use hashCode when you need a cryptographic hash!
  - See CIS 331 for more details

Hashing: take away lessons

equals

hashCode

compareTo

Collections Requirements

- All collections invoke `equals` method on elements
  - Defaults to == (reference equality)
  - Override `equals` to create structural equality
  - Should always be an equivalence relation: reflexive, symmetric, transitive

- HashSets/HashMaps also invoke `hashCode` method on elements
  - Override when equals is overridden
  - Should be "compatible" with `equals`
  - Should try to distribute hash codes uniformly
  - Iterators are not guaranteed to follow order of hashCodes

- Ordered collections (`TreeSet`, `TreeMap`) require element type to implement `Comparable` interface
  - Provide `compareTo` method
  - Should implement a *total order*
  - Should be compatible with equals
    - (i.e. `o1.equals(o2)` exactly when `o1.compareTo(o2) == 0`)

Threads & Synchronization

Avoid Race Conditions!

(see Multithreaded.java)

Threads

- Java programs can be *multithreaded*
  - more than one "thread" of control operating simultaneously
- A `Thread` object can be created from any class that implements the `Runnable` interface
  - `start`: launch the thread
  - `join`: wait for the thread to finish
- Abstract Stack Machine:
  - Each thread has its own workspace and stack
  - All threads *share* a common heap
  - Threads can communicate via shared references

Uses + Perils

- Threads are useful when one program needs to do multiple things simultaneously:
  - game animation + user input
  - chat server interacting with multiple chat clients
  - hide latency: do work in one thread while another thread waits (e.g. for disk or network I/O)

- Problem: Race Conditions
  - What happens when one thread tries to read a memory location at the same time another thread is writing it?
  - What if more than one thread tries to write different values at the same time?

(Unsynchronized) Implementation

```
interface Counter {
  public void inc();
  public int get();
}

class UCounter implements Counter {
  private int cnt = 0;

  public void inc() {
    cnt = cnt + 1;
  }

  public int get() {
    return cnt;
  }
}
```

Setting up a Computation Thread

```
// The computation thread simply increments
// the provided counter 1000 times
class CounterUser implements Runnable {
  private Counter c;
  private int id;

  CounterUser(int id, Counter c) {
    this.id = id;
    this.c = c;
  }

  @Override
  public void run() {
    for (int i = 0; i < 1000; i++) {
      // System.out.println("Thread: " + id);
      c.inc();
    }
  }
}
```

First Try: Two Threads & One Counter

```
public class MultiThreaded {

  public static void main(String[] args) {
    Counter c = new UCounter();

    // set up a race on the shared counter c
    Thread t1 = new Thread(new CounterUser(1, c));
    Thread t2 = new Thread(new CounterUser(2, c));
    t1.start();
    t2.start();
    try {
      t1.join();
      t2.join();
    } catch (InterruptedException e) {
    }
    System.out.println("Counter value = " + c.get());
  }

}
```

Create thread 1

Create thread 2

Start thread 1

Start thread 2

Wait for thread 1 to finish

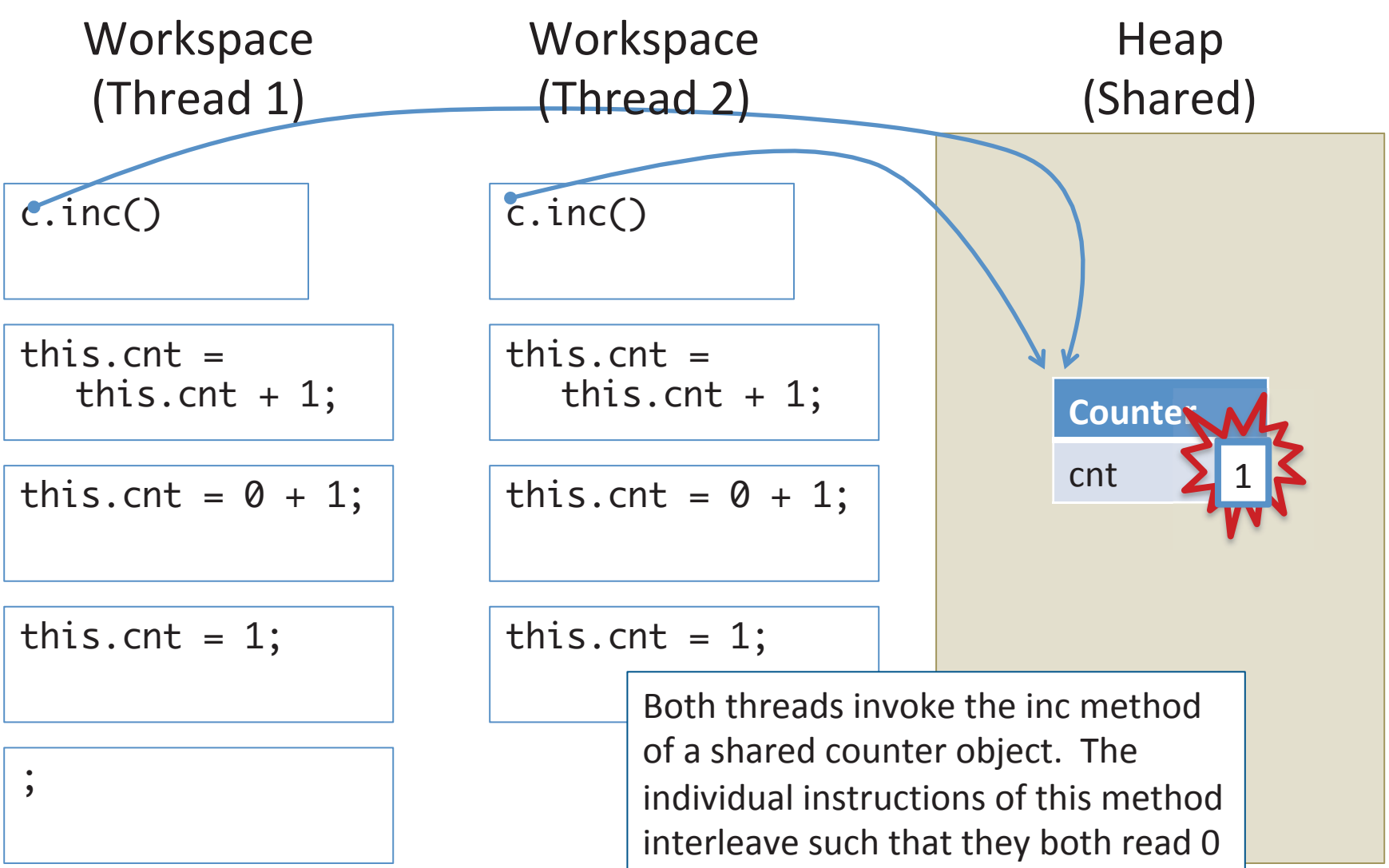
Wait for thread 2 to finish

What behavior do you expect from Multithreaded.java?

1. The program will print "Counter value = 1000"
2. The program will print "Counter value = 2000"
3. The program will print "Counter value = ????" for some other number ????
4. The program will throw an exception.

Answer: The program with print "Counter value = *val*"
for 1000 >= val >= 2000.
The answer will likely be *different* each time the program is run!!!!

Data Races


Workspace (Thread 1)
Workspace (Thread 2)
Heap (Shared)
c.inc()
this.cnt = this.cnt + 1;
this.cnt = 0 + 1;
this.cnt = 1;
;
c.inc()
this.cnt = this.cnt + 1;
this.cnt = 0 + 1;
this.cnt = 1;
cnt
1
Both threads invoke the inc method of a shared counter object. The individual instructions of this method interleave such that they both read 0 and write 1.

The `synchronized` keyword

- Synchronized methods are *atomic*
  - They run without any other threads running
- Careful use will eliminate races
- Tradeoff:
  - less concurrency means worse performance

Second Try: use Synchronization

```
//This class uses synchronization
class SynchronizedCounter implements Counter {
  private int cnt = 0;

  public synchronized void inc() {
   cnt = cnt + 1;
  }

  public synchronized int get() {
   return cnt;
  }
}
```

Using The New Counters

```
public class MultiThreaded {

  public static void main(String[] args) {

    Counter c = new SynchronizedCounter();

    // set up a race on the shared counter c
    Thread t1 = new Thread(new CounterUser(1, c));
    Thread t2 = new Thread(new CounterUser(2, c));
    t1.start();
    t2.start();
    try {
      t1.join();
      t2.join();
    } catch (InterruptedException e) {
    }

    System.out.println("Counter value = " + c.get());
  }

}
```

New!!

Now what behavior do you expect from Multithreaded.java?

1. The program will print "Counter value = 1000"
2. The program will print "Counter value = 2000"
3. The program will print "Counter value = ????" for some other number ????
4. The program will throw an exception.

Answer: The program with print “Counter value = 2000” every time.

Other Synchronization in Java

Need *thread safe* libraries:

- `java.util.concurrent` has `BlockingQueue` and `ConcurrentMap`
- help rule out synchronization errors
- Note: Swing is *not* thread safe!

- Java also provides *locks*
  - objects that act as synchronizers for blocks of code

- *Deadlock*: cyclic dependency in synchronization of locks
  - Thread A waiting for lock held by B,
    Thread B waiting for lock held by A

Immutability!

- Note that read-only datastructures are immune to race conditions
  - It's OK for multiple threads to read a heap location simultaneously
  - Less need for locking, synchronization

- As always: immutable data structures simplify your code

Real-world example:

FaceBook's Haxl Library

- Library written in Haskell
- Concurrency / Distributed Database
- https://github.com/facebook/Haxl

Programming Languages and Techniques (CIS120)

Lecture 38

December 6th, 2017

Java Miscellany: Concurrency, GC & Memory Management

Announcements

- HW09: Game of your own
  - Due Monday, December 11th at 11:59pm
  - No late submissions!

Final Exam:

- Friday, December 15th 6:00-8:00 PM
- 3 Locations:
  - Towne 100 Last Names A – G
  - Stiteler B6 Last Names H – Pa
  - Cohen G17 Last Names Pe -- Z

- Mock Exam:

  Wednesday, December 13th 6:00-8:00PM

  Towne 100

Advanced Java Miscellany

- Hashing: HashSets & HashMaps
- Threads & Synchronization
- Garbage Collection
- Java 1.8 Lambdas

We'll touch on these.

- 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 nitty-gritty details:
Java Language Specification
http://docs.oracle.com/javase/specs/

Threads & Synchronization

Avoid Race Conditions!

(see Multithreaded.java)

Threads

- Java programs can be *multithreaded*
  - more than one "thread" of control operating simultaneously
- A `Thread` object can be created from any class that implements the `Runnable` interface
  - `start`: launch the thread
  - `join`: wait for the thread to finish
- Abstract Stack Machine:
  - Each thread has its own workspace and stack
  - All threads *share* a common heap
  - Threads can communicate via shared references

Uses + Perils

- Threads are useful when one program needs to do multiple things simultaneously:
  - game animation + user input
  - chat server interacting with multiple chat clients
  - hide latency: do work in one thread while another thread waits (e.g. for disk or network I/O)

- Problem: Race Conditions
  - What happens when one thread tries to read a memory location at the same time another thread is writing it?
  - What if more than one thread tries to write different values at the same time?

(Unsynchronized) Implementation

```
interface Counter {
  public void inc();
  public int get();
}

class UCounter implements Counter {
  private int cnt = 0;

  public void inc() {
    cnt = cnt + 1;
  }

  public int get() {
    return cnt;
  }
}
```

Setting up a Computation Thread

```
// The computation thread simply increments
// the provided counter 1000 times
class CounterUser implements Runnable {
  private Counter c;
  private int id;

  CounterUser(int id, Counter c) {
    this.id = id;
    this.c = c;
  }

  @Override
  public void run() {
    for (int i = 0; i < 1000; i++) {
      // System.out.println("Thread: " + id);
      c.inc();
    }
  }
}
```

First Try: Two Threads & One Counter

```
public class MultiThreaded {

  public static void main(String[] args) {
    Counter c = new UCounter();

    // set up a race on the shared counter c
    Thread t1 = new Thread(new CounterUser(1, c));
    Thread t2 = new Thread(new CounterUser(2, c));
    t1.start();
    t2.start();
    try {
      t1.join();
      t2.join();
    } catch (InterruptedException e) {
    }
    System.out.println("Counter value = " + c.get());
  }

}
```

Create thread 1

Create thread 2

Start thread 1

Start thread 2

Wait for thread 1 to finish

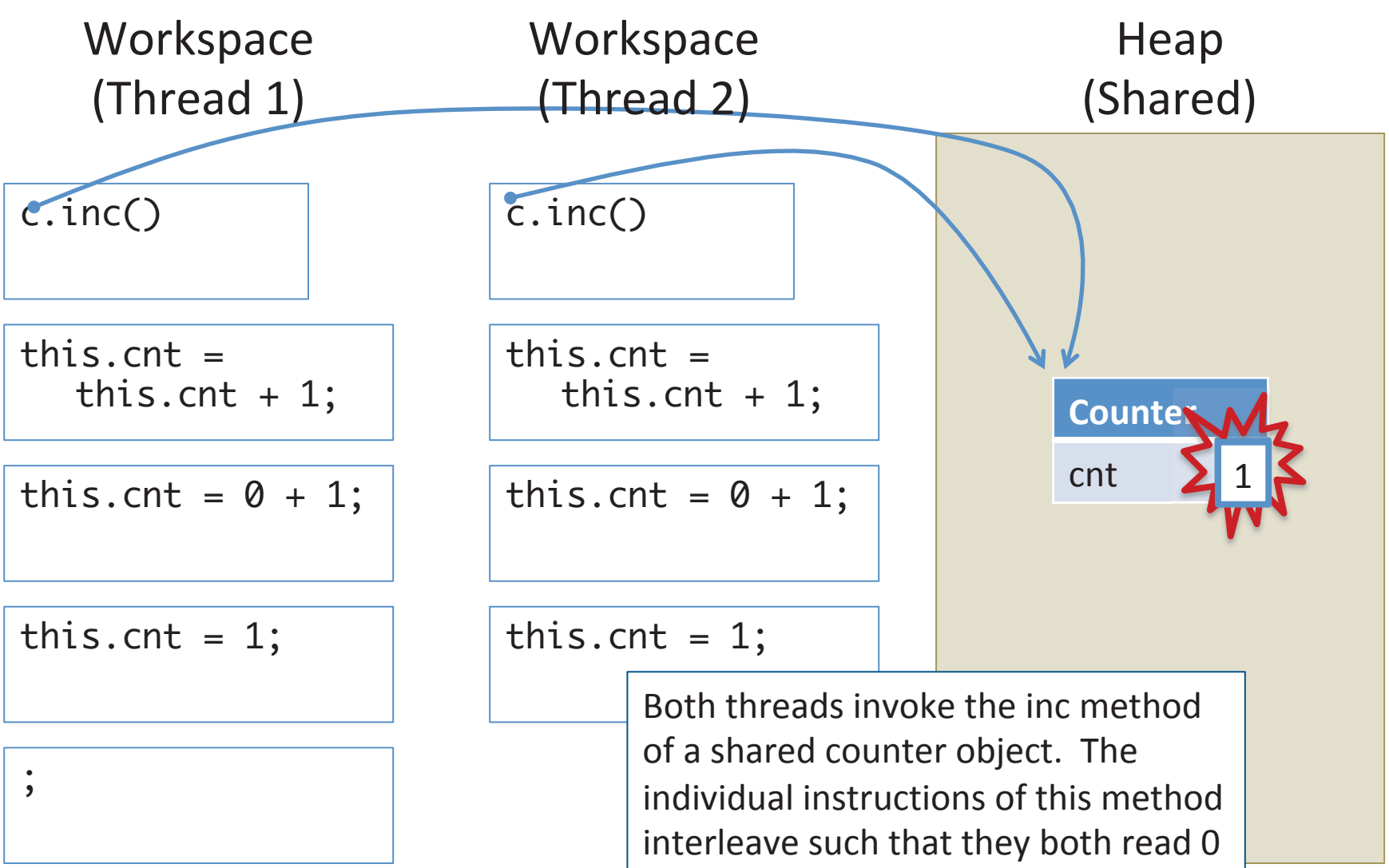
Wait for thread 2 to finish

What behavior do you expect from Multithreaded.java?

1. The program will print "Counter value = 1000"
2. The program will print "Counter value = 2000"
3. The program will print "Counter value = ????" for some other number ????
4. The program will throw an exception.

Answer: The program with print "Counter value = *val*"
for 1000 >= val >= 2000.
The answer will likely be *different* each time the program is run!!!!

Data Races


Workspace (Thread 1)
Workspace (Thread 2)
Heap (Shared)
c.inc()
this.cnt = this.cnt + 1;
this.cnt = 0 + 1;
this.cnt = 1;
;
c.inc()
this.cnt = this.cnt + 1;
this.cnt = 0 + 1;
this.cnt = 1;
cnt
1
Both threads invoke the inc method of a shared counter object. The individual instructions of this method interleave such that they both read 0 and write 1.

The `synchronized` keyword

- Synchronized methods are *atomic*
  - They run without any other threads running
- Careful use will eliminate races
- Tradeoff:
  - less concurrency means worse performance

Second Try: use Synchronization

```
//This class uses synchronization
class SynchronizedCounter implements Counter {
  private int cnt = 0;

  public synchronized void inc() {
   cnt = cnt + 1;
  }

  public synchronized int get() {
   return cnt;
  }
}
```

Using The New Counters

```
public class MultiThreaded {

  public static void main(String[] args) {

    Counter c = new SynchronizedCounter();

    // set up a race on the shared counter c
    Thread t1 = new Thread(new CounterUser(1, c));
    Thread t2 = new Thread(new CounterUser(2, c));
    t1.start();
    t2.start();
    try {
      t1.join();
      t2.join();
    } catch (InterruptedException e) {
    }

    System.out.println("Counter value = " + c.get());
  }

}
```

New!!

Now what behavior do you expect from Multithreaded.java?

1. The program will print "Counter value = 1000"
2. The program will print "Counter value = 2000"
3. The program will print "Counter value = ????" for some other number ????
4. The program will throw an exception.

Answer: The program with print “Counter value = 2000” every time.

Other Synchronization in Java

Need *thread safe* libraries:

- `java.util.concurrent` has `BlockingQueue` and `ConcurrentMap`
- help rule out synchronization errors
- Note: Swing is *not* thread safe!

- Java also provides *locks*
  - objects that act as synchronizers for blocks of code

- *Deadlock*: cyclic dependency in synchronization of locks
  - Thread A waiting for lock held by B,
    Thread B waiting for lock held by A

Immutability!

- Note that read-only datastructures are immune to race conditions
  - It's OK for multiple threads to read a heap location simultaneously
  - Less need for locking, synchronization

- As always: immutable data structures simplify your code

Real-world example:

FaceBook's Haxl Library

- Library written in Haskell
- Concurrency / Distributed Database
- https://github.com/facebook/Haxl

Garbage Collection & Memory Management

Cleaning up the Heap

Memory Management

- The Java Abstract Machine stores all objects in the heap.
  - We imagine that the heap has limitless space…
    … but: real machines have limited amounts of memory
- *Manual memory management*
  - C and C++
  - The programmer explicitly allocates heap objects (`malloc` / `new`)
  - The programmer explicitly de-allocates the objects (`free` / `delete`)
- *Automatic memory management (garbage collection)*
  - Reference Counting: Objective C, Swift, Python, many scripting languages
  - Mark & sweep/Copying GC: Java, OCaml, C#, Haskell (and most other 'managed' languages)

Manual Memory Management

See manmem.c

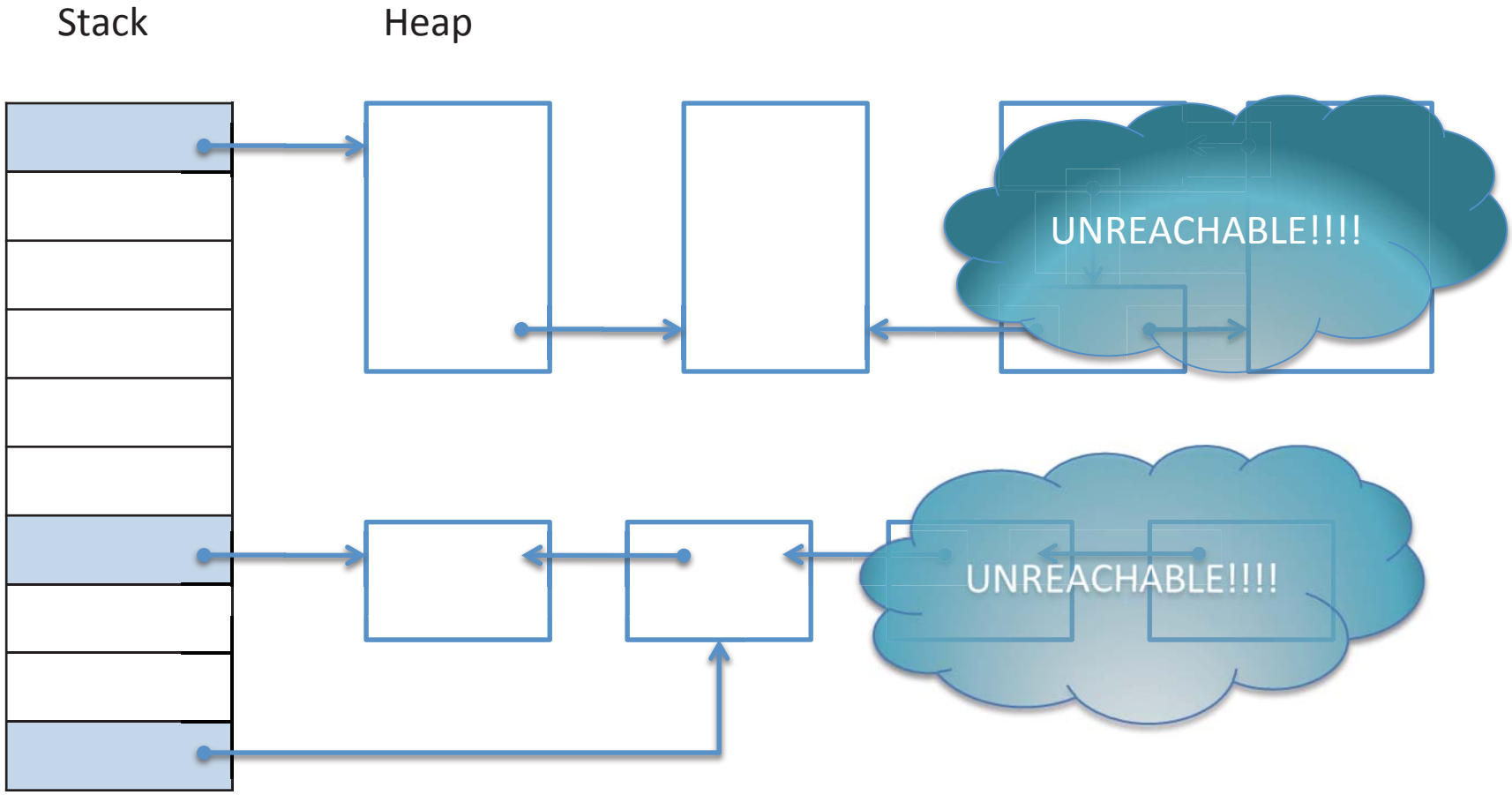
Why Garbage Collection?

- Manual memory management is cumbersome & error prone:
  - Freeing the same reference twice is ill defined (crashes or other bugs)
  - Explicit `free` isn't modular: To properly free all allocated memory, the programmer has to know what code "owns" each object. Owner code must ensure `free` is called just once.
  - Not calling `free` leads to *space leaks*: memory never reclaimed
    - Many examples of space leaks in long-running programs

- Garbage collection:
  - Have the language runtime system determine when an allocated chunk of memory will no longer be used and free it automatically.
  - Extremely convenient and safe
  - Garbage collection does impose costs (performance, predictability)

Graph of Objects in the Heap

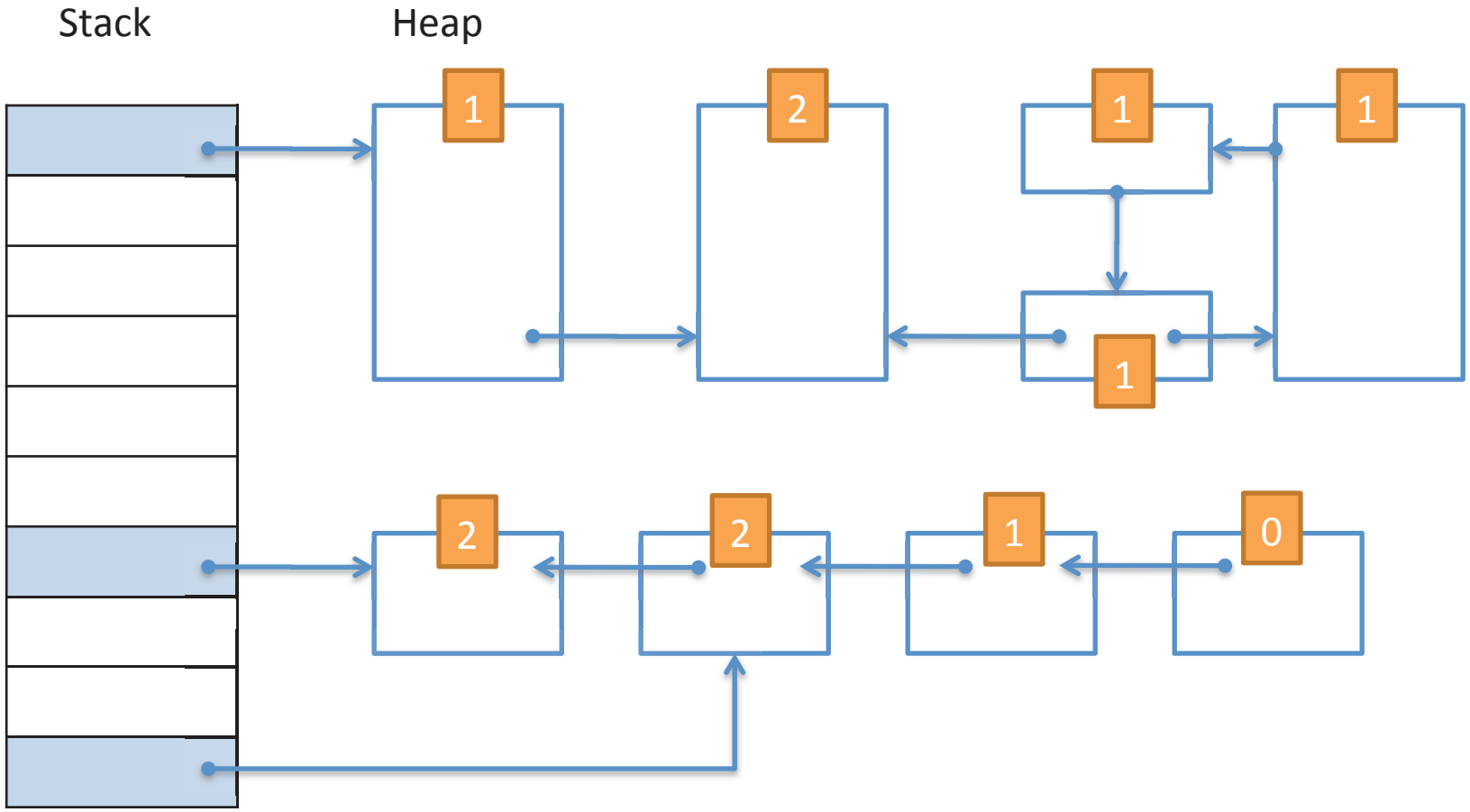
- References in the stack and global static fields are *roots*



Reference Counting

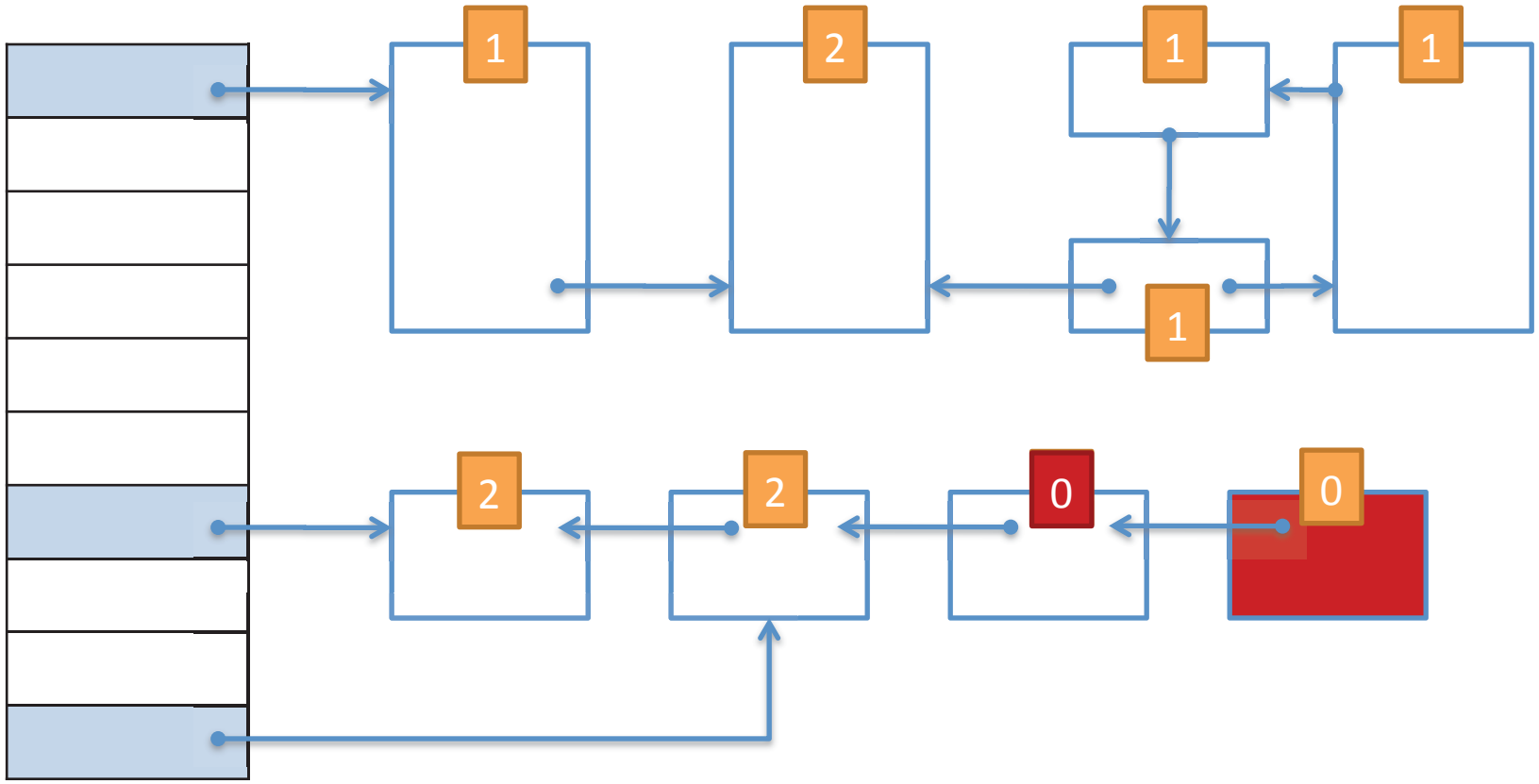
Reference Counting

- Each heap object tracks how many references point to it:



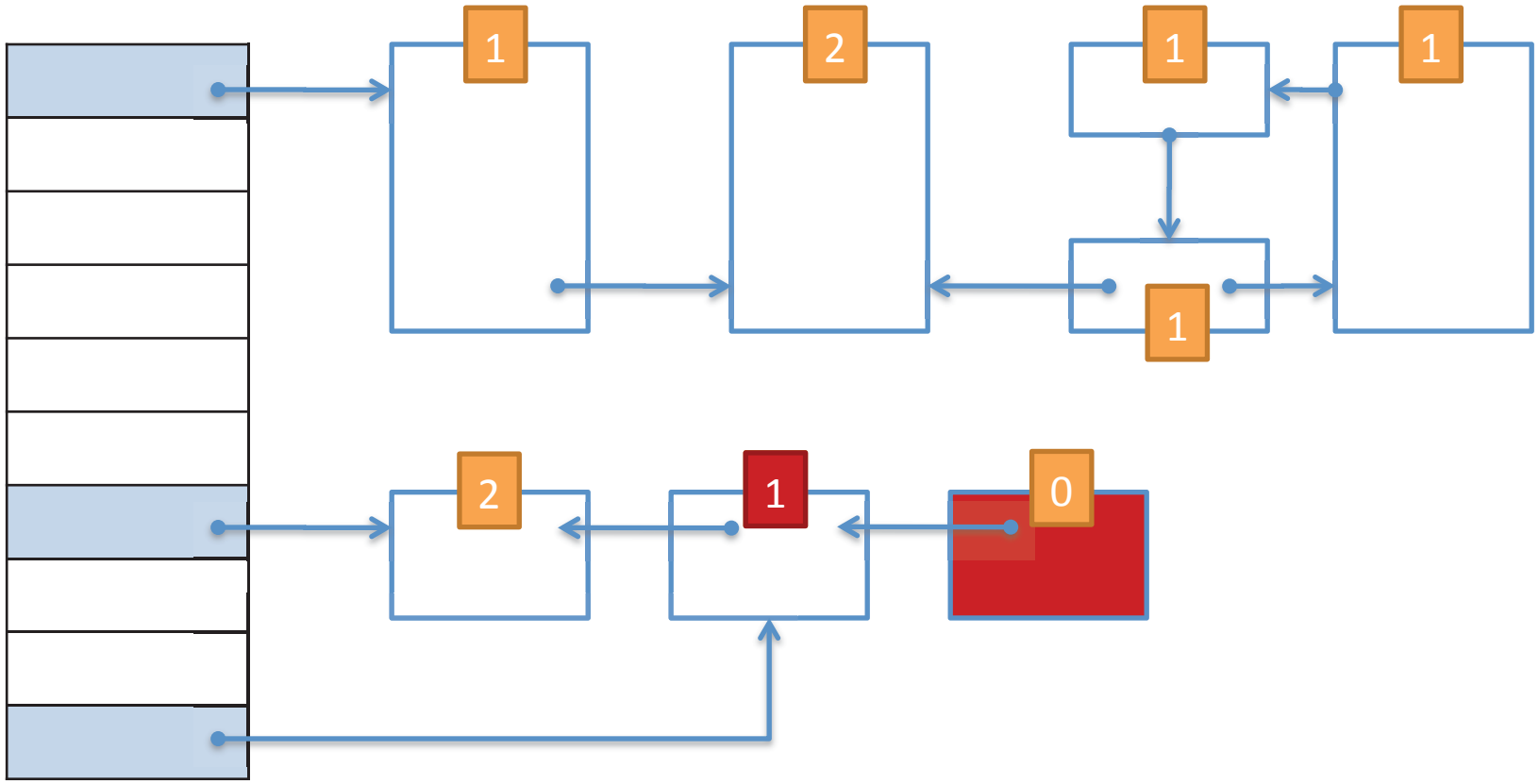
Reference Couting

- When reference count goes to 0, reclaim that space
  - and decrement counts for objects pointed to by that object



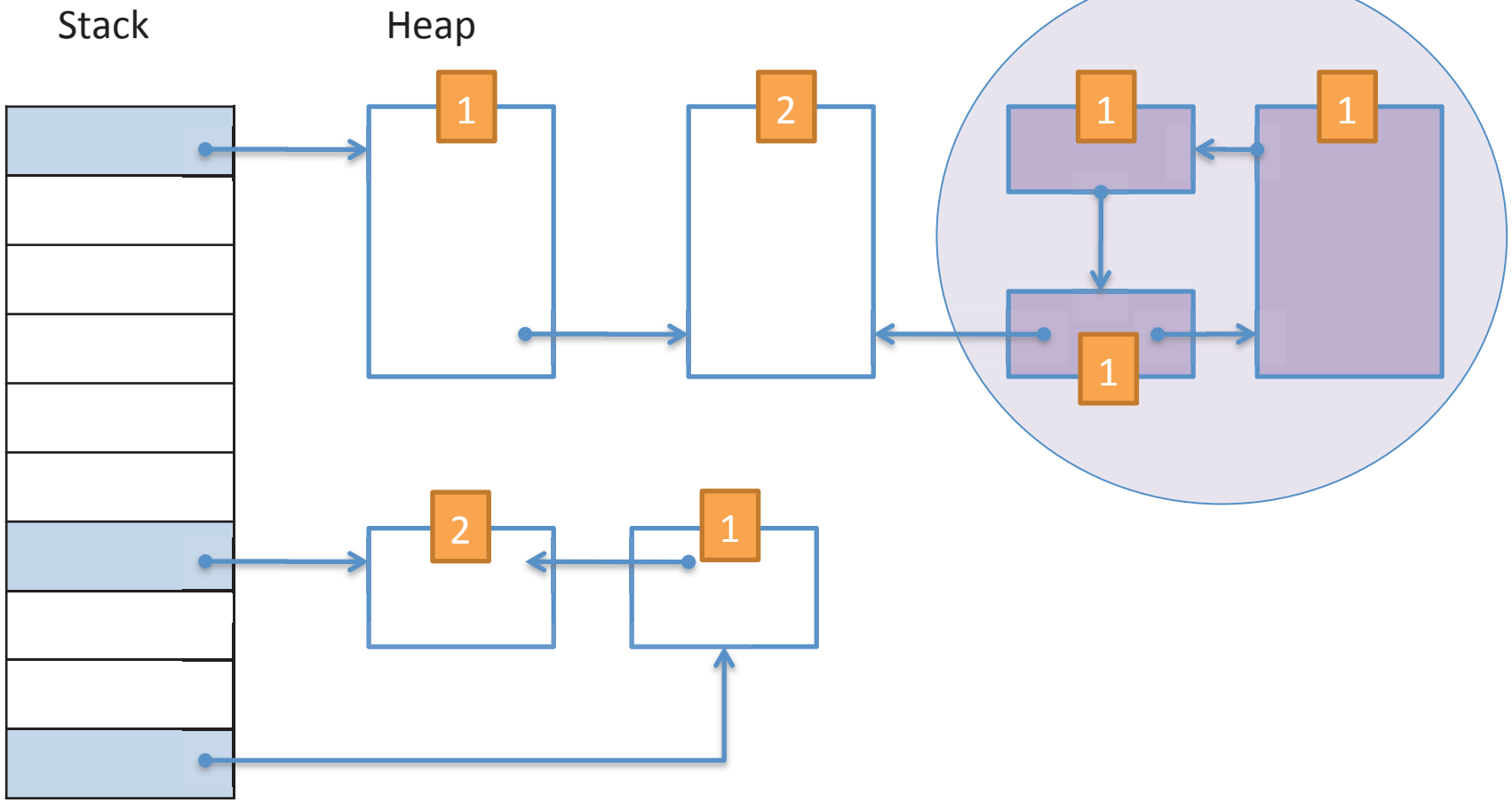
Reference Couting

- When reference count goes to 0, reclaim that space
  - and decrement counts for objects pointed to by that object



Problem: Cyclic Data

- Cycles of data will never decrement to 0!
  - *Can lead to "space leaks"*



Dealing with Cycles

- Option 1: Require programmers to explicitly null-out references to break cycles.

- Option 2: Periodically run mark & sweep GC to collect cycles

- Option 3: Require programmers to distinguish "weak pointers" from "strong pointers"
  - *weak pointers*: if all references to an object are "weak" then the object can be freed even with non-zero reference count.
  - "Back edges" in the object graph should be designated as weak
  - (Aside: weak pointers useful in GC settings too.)

Mark & Sweep / Copying

Traverse the Heap

Memory Use & Reachability

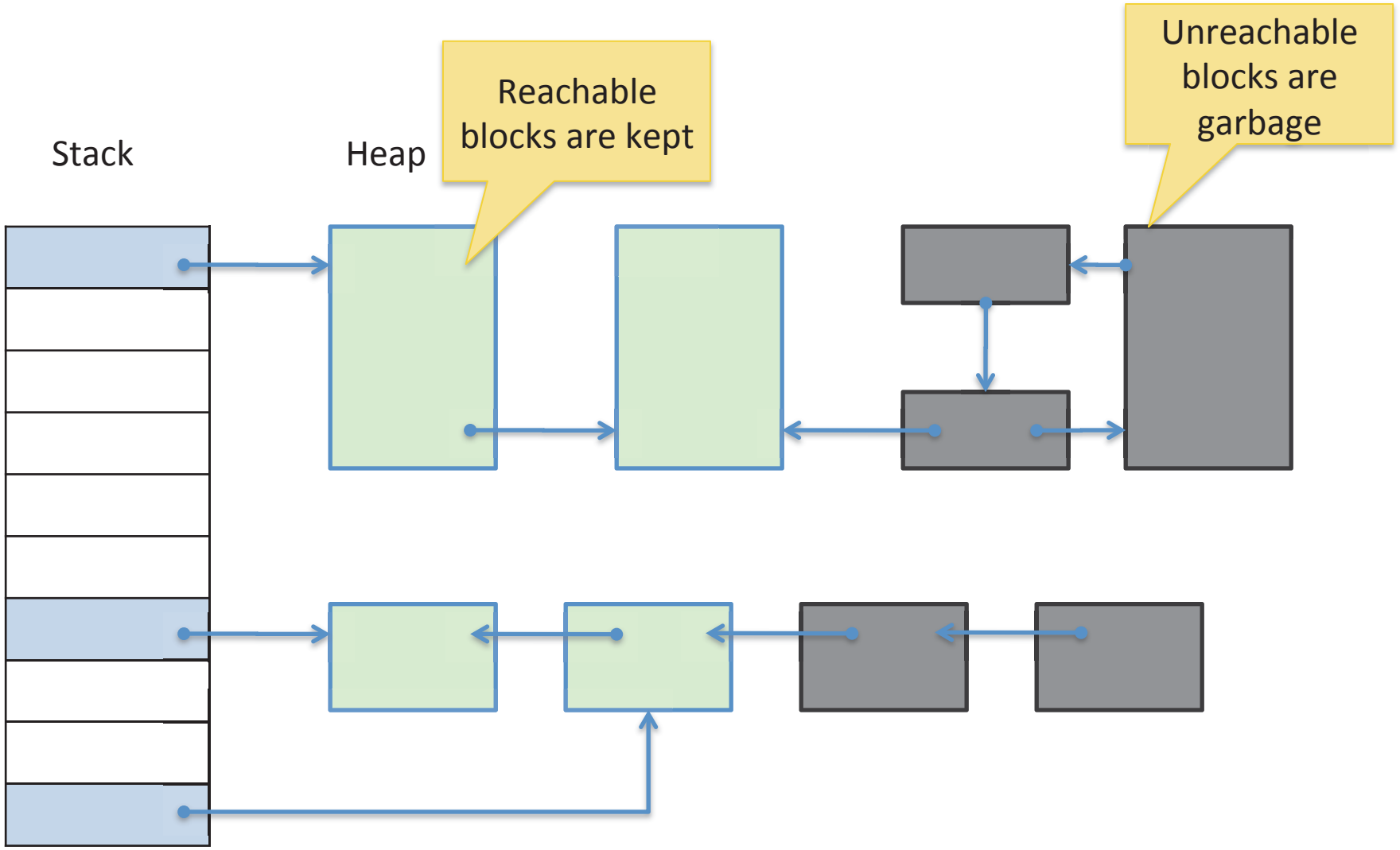
- When is a chunk of memory no longer needed?
  - In general, this problem is undecidable.

- We can approximate this information by freeing memory that can't be reached from any *root* references.
  - A *root reference* is one that might be accessible directly from the program (i.e. they're not in the heap).
  - Root references include (global) static fields and references in the stack.
- If an object can be reached by traversing pointers from a root, it is *live*.
- It is safe to reclaim all heap allocations not reachable from a root (such objects are *garbage* or *dead* objects).

Mark and Sweep Garbage Collection

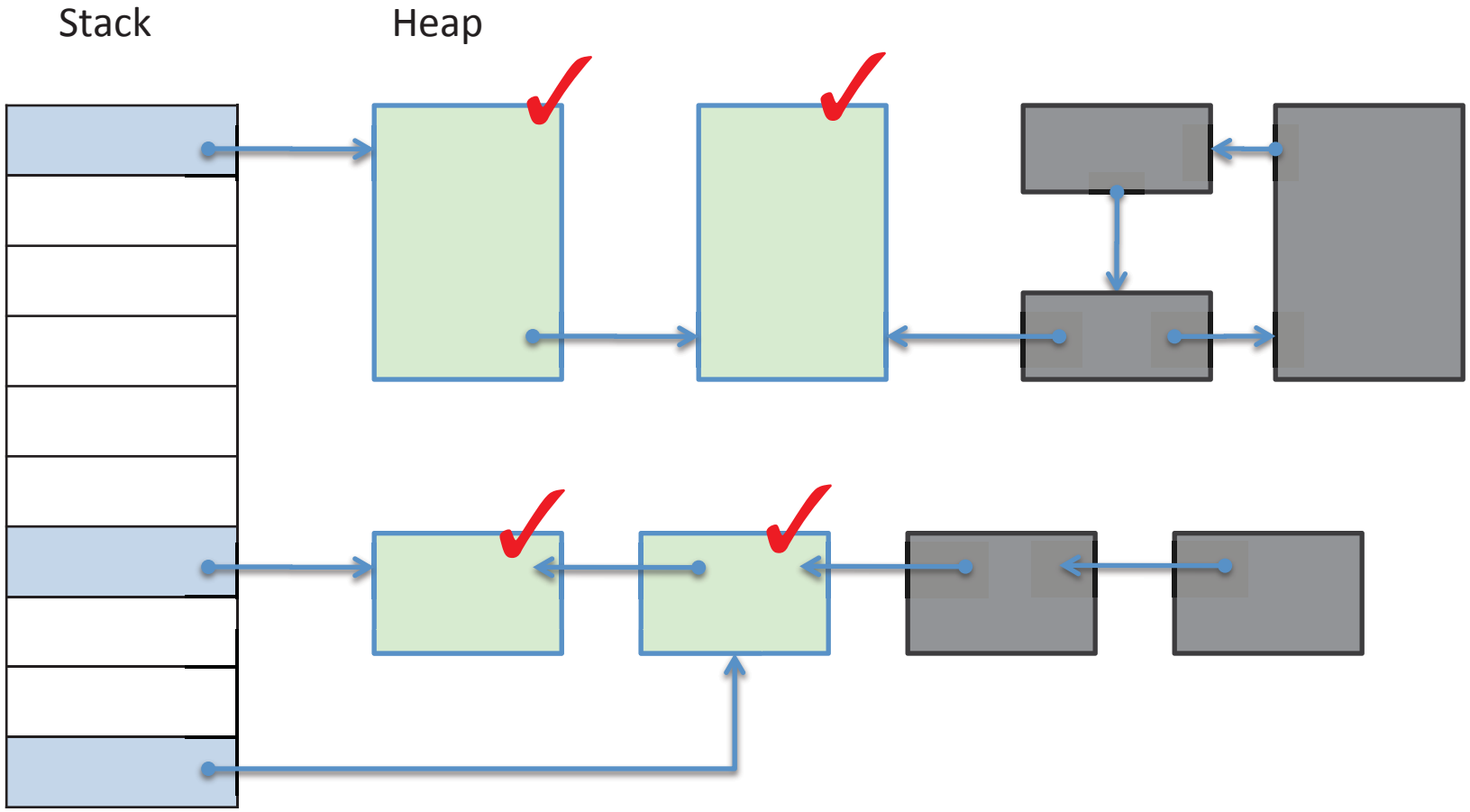
- Classic algorithm with two phases:
- Phase 1: Mark
  - Start from the roots
  - Do depth-first traversal, marking every object reached.
- Phase 2: Sweep
  - Walk over *all* allocated objects and check for marks.
  - Unmarked objects are reclaimed.
  - Marked objects have their marks cleared.
  - Optional: compact all live objects in heap by moving them adjacent to one another. (Needs extra work & indirection to “patch up” references)
- (In practice much more complex: "generational GC")

Results of Marking Graph



Second Phase: Drop "Unreachable"

- Sweep over all objects, dropping the ones marked as unreachable and keeping the ones marked reachable.



Costs & Implications

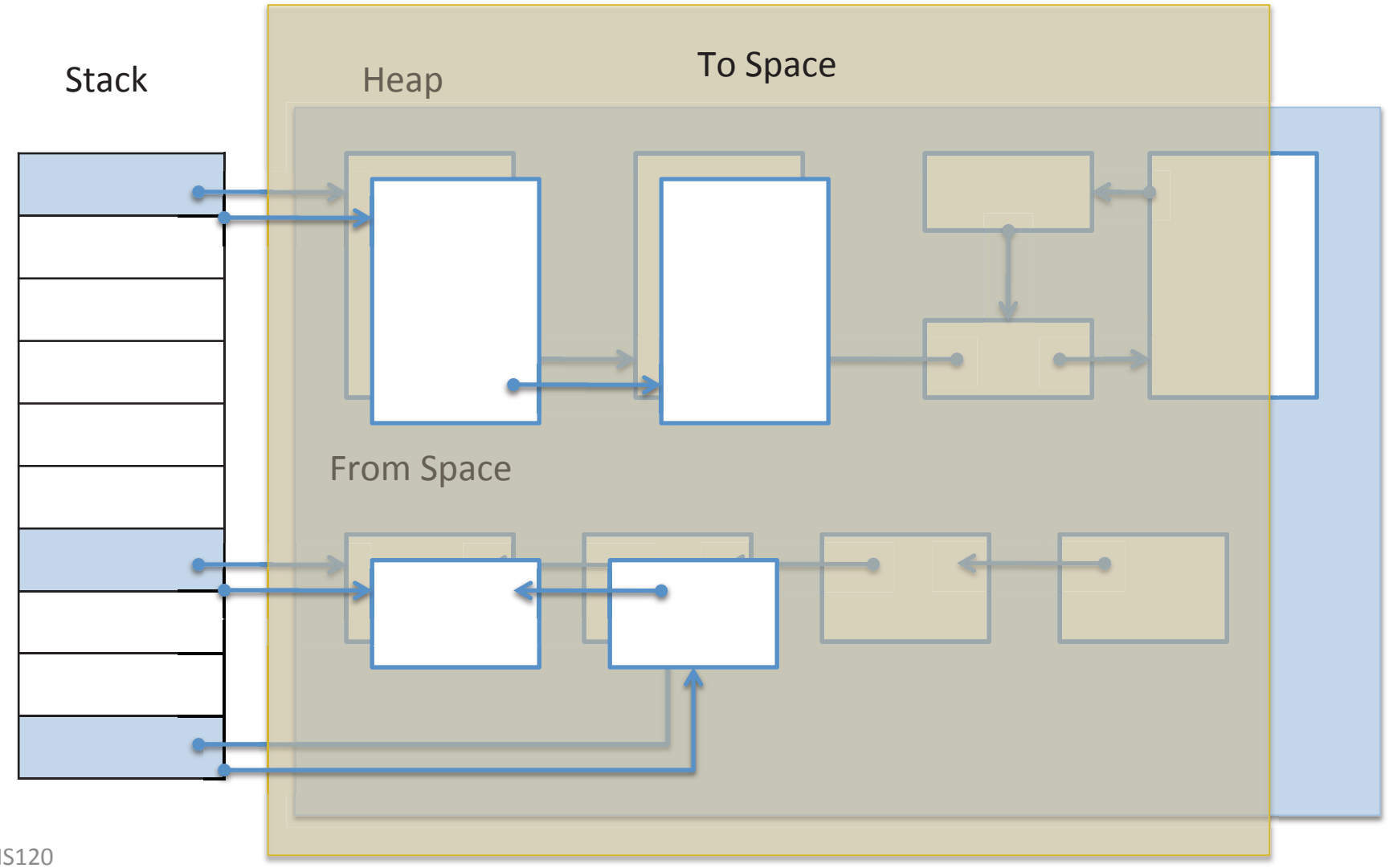
- Need to generalize to account for objects that have multiple outgoing pointers.
- Mark & Sweep algorithm reads all memory in use by the program (even if it's garbage!)
  - Running time is proportional to the total amount of allocated memory (both live and garbage).
  - Can pause the programs for long times during garbage collection.

Copying Garbage Collection

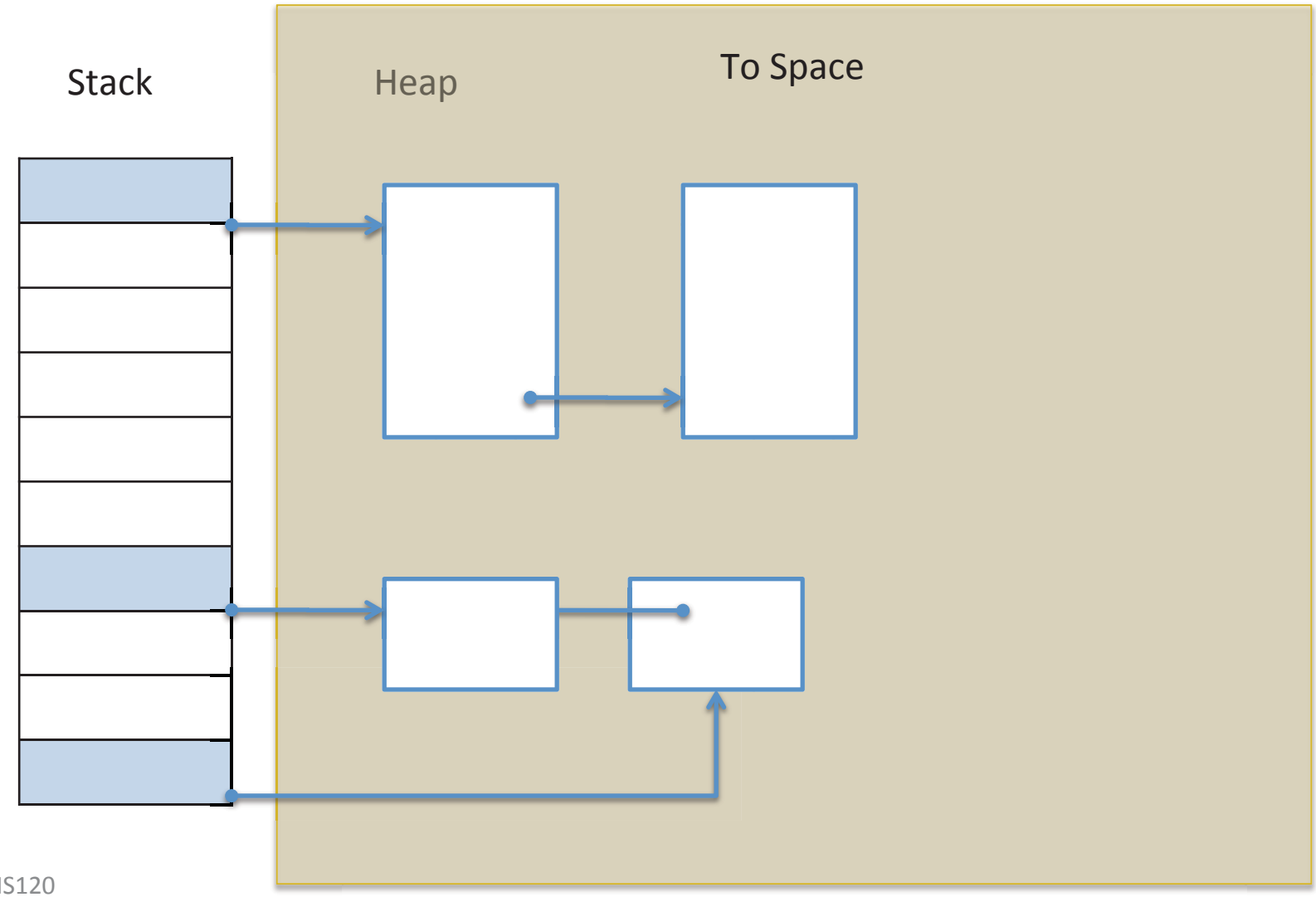
- Like mark & sweep: collects all garbage.
- Basic idea: use *two* regions of memory
  - One region is the memory in use by the program. New allocation happens in this region.
  - Other region is idle until the GC requires it.

- Garbage collection algorithm:
  - Traverse over live objects in the active region (called the "*from-space*"), copying them to the idle region (called the "*to-space*").
  - After copying all reachable data, switch the roles of the from-space and to-space.
  - All dead objects in the (old) from-space are discarded en masse.
  - A side effect of copying is that all live objects are compacted together.

Copy from "From" to "To"



Discard the "From Space"



GCDemo

See GCTest.java

Garbage Collection Take Aways

- Big idea: the Java runtime system tries to free-up as much memory as it can automatically.
  - Almost always a big win, in terms of convenience and reliability

- Sometimes can affect performance:
  - Lots of dead objects might take a long time to collect
  - When garbage collection will be triggered can be hard to predict, so there can be "pauses" (modern GC implementations try to avoid this!)
  - Global data structures can have references to "zombie" objects that won't be used, but are still reachable ⇒ "space leak".

- There are many advanced programming techniques to address these issues:
  - Configuring the GC parameters
  - Explicitly triggering a GC phase
  - "Weak" references

Programming Languages and Techniques (CIS120)

Lecture 39

December 8th, 2017

Java Miscellany: GC & Memory Management, Java 1.8 Functional Programming and Lambdas

Announcements

- HW09: Game of your own
  - Due Monday, December 11th at 11:59pm
  - No late submissions!

Final Exam:

- Friday, December 15th 6:00-8:00 PM
- 3 Locations:
  - Towne 100 Last Names A – G
  - Stiteler B6 Last Names H – Pa
  - Cohen G17 Last Names Pe -- Z

- Mock Exam:

  Wednesday, December 13th 6:00-8:00PM

  Towne 100

Advanced Java Miscellany

- Hashing: HashSets & HashMaps
- Threads & Synchronization
- Garbage Collection
- Java 1.8 Lambdas

We'll touch on these.

- 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 nitty-gritty details:
Java Language Specification
http://docs.oracle.com/javase/specs/

Garbage Collection & Memory Management

Cleaning up the Heap

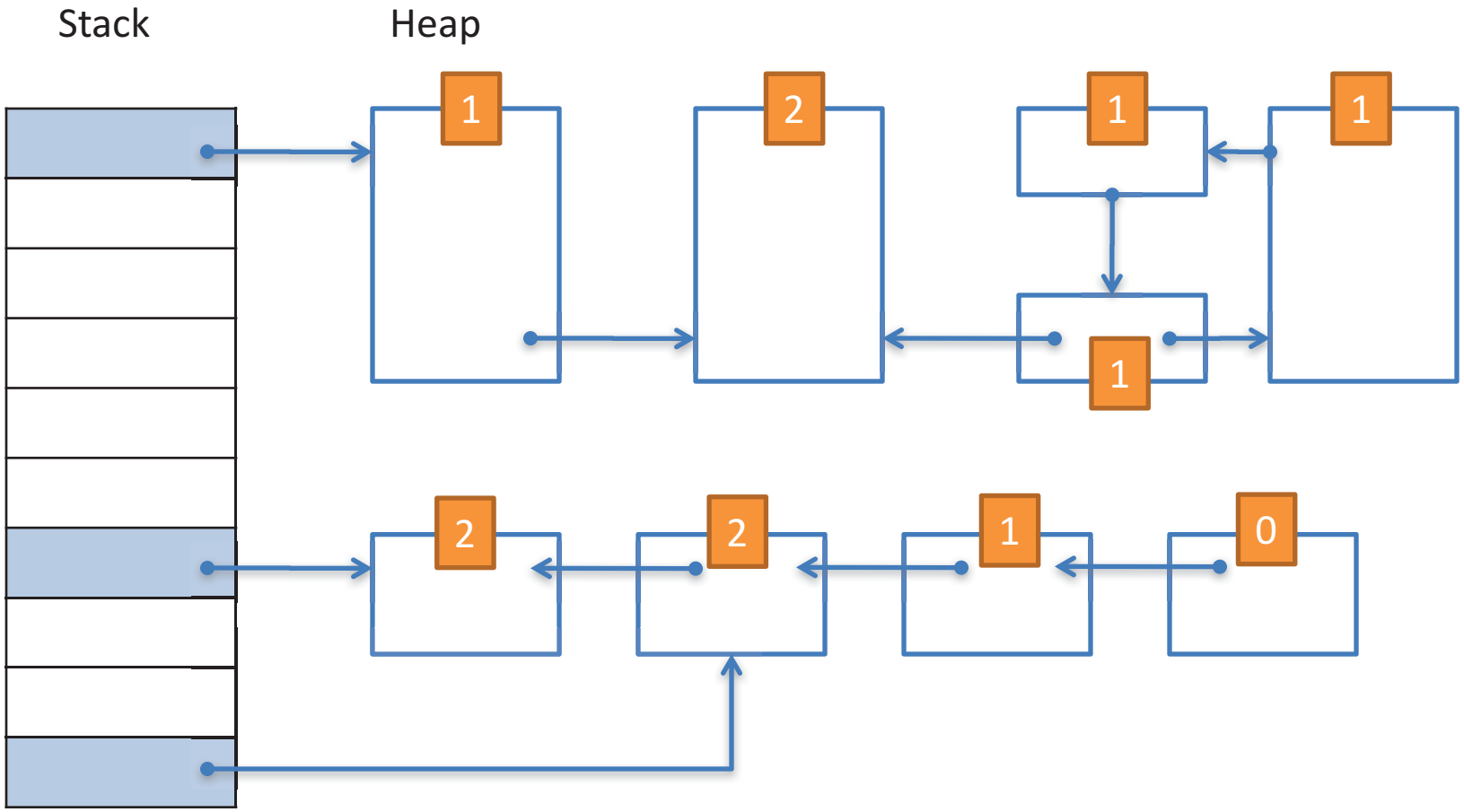
Memory Management

- The Java Abstract Machine stores all objects in the heap.
  - We imagine that the heap has limitless space…
    … but: real machines have limited amounts of memory
- *Manual memory management*
  - C and C++
  - The programmer explicitly allocates heap objects (`malloc` / `new`)
  - The programmer explicitly de-allocates the objects (`free` / `delete`)
- *Automatic memory management (garbage collection)*
  - Reference Counting: Objective C, Swift, Python, many scripting languages
  - Mark & sweep/Copying GC: Java, OCaml, C#, Haskell (and most other 'managed' languages)

Reference Counting

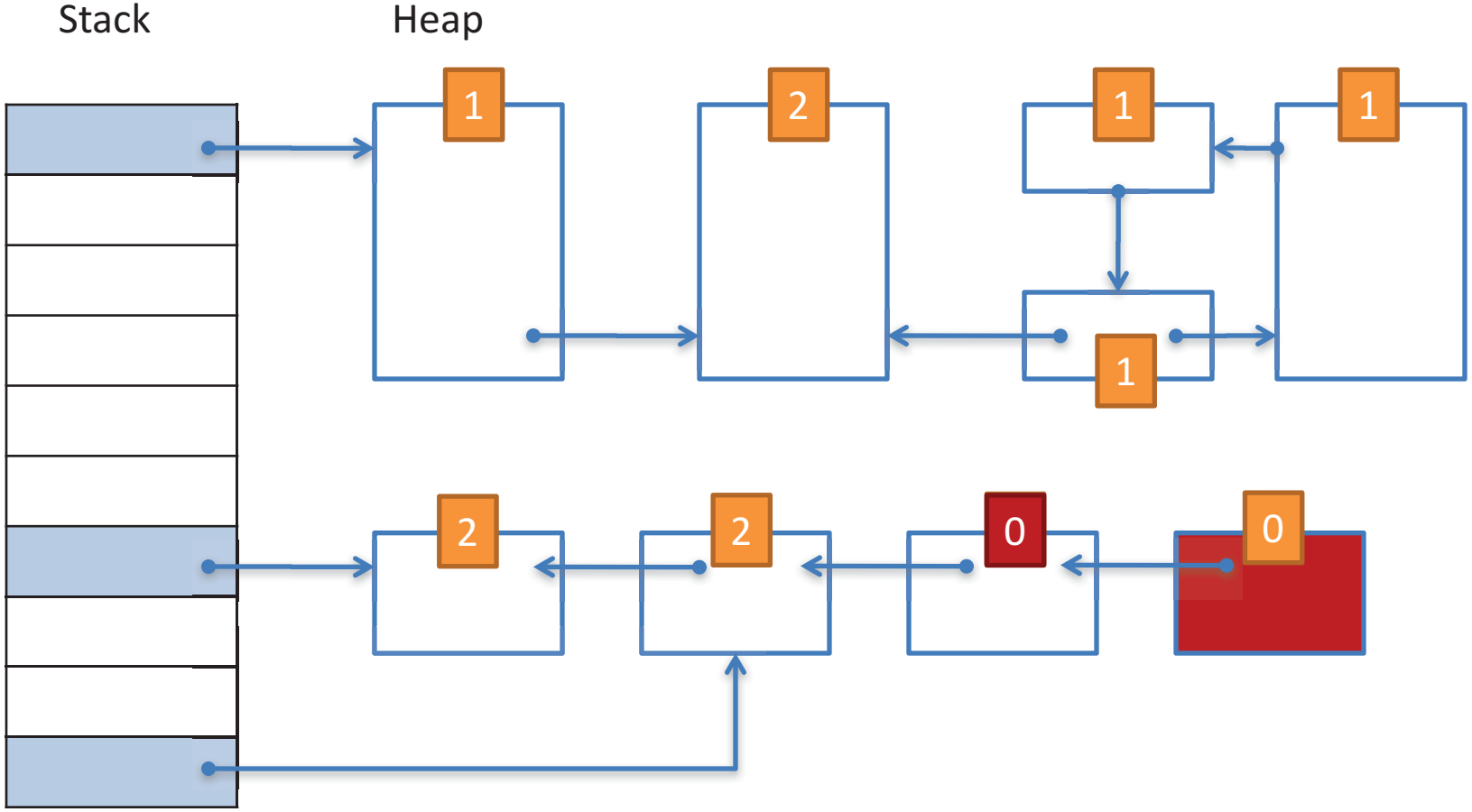
Reference Counting

- Each heap object tracks how many references point to it:



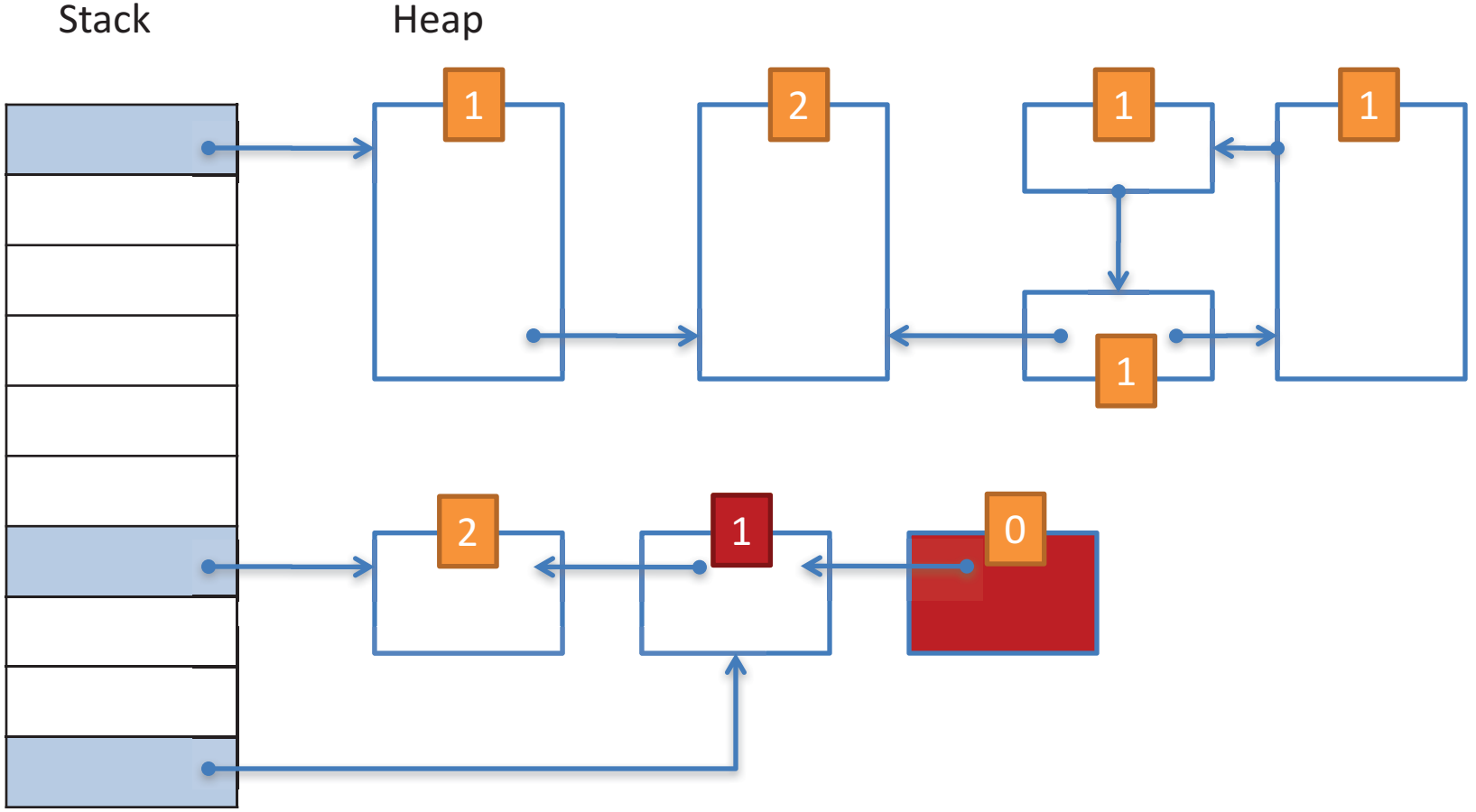
Reference Couting

- When reference count goes to 0, reclaim that space
  - and decrement counts for objects pointed to by that object



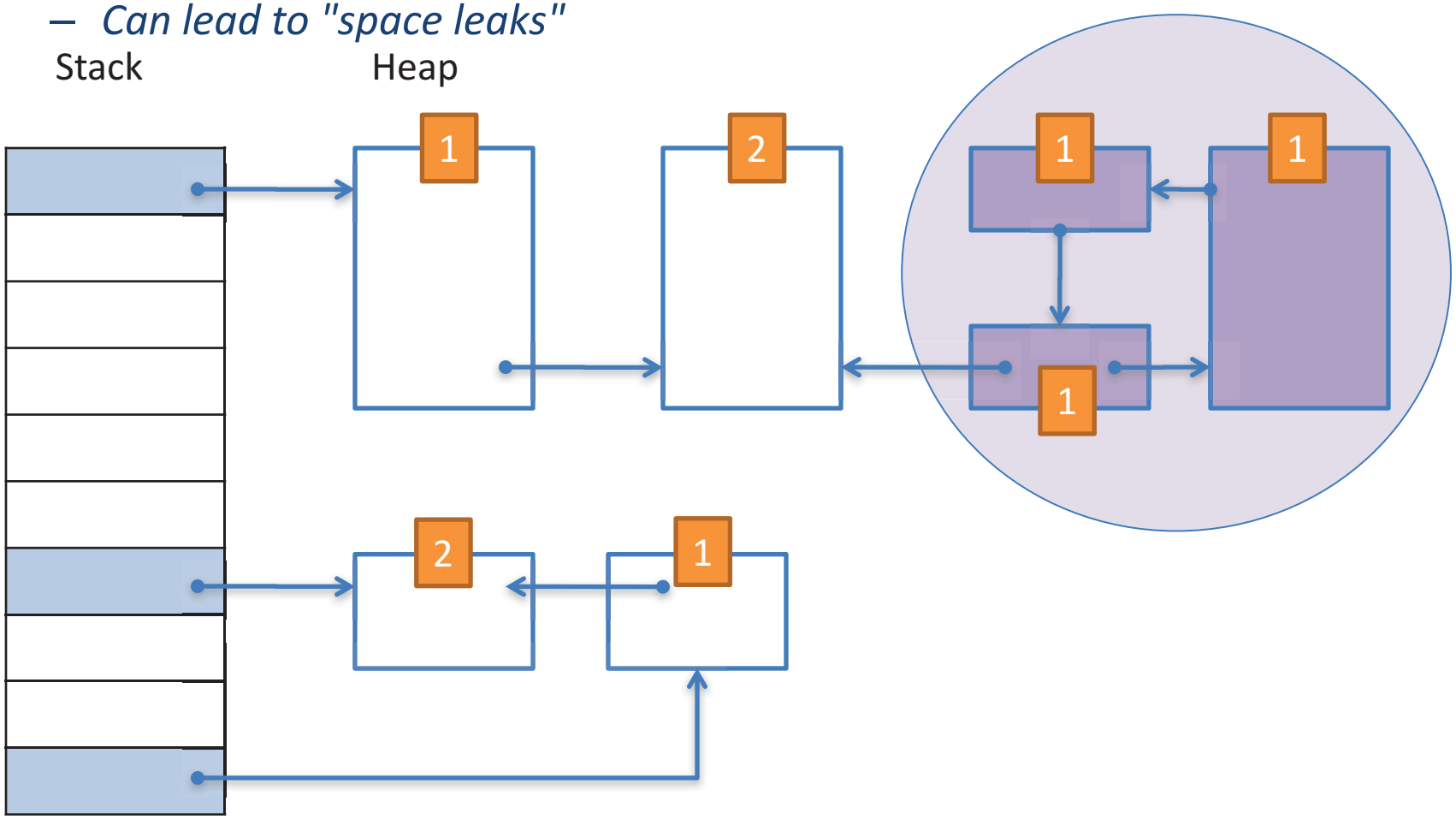
Reference Couting

- When reference count goes to 0, reclaim that space
  - and decrement counts for objects pointed to by that object



Problem: Cyclic Data

- Cycles of data will never decrement to 0!
  - *Can lead to "space leaks"*



Dealing with Cycles

- Option 1: Require programmers to explicitly null-out references to break cycles.

- Option 2: Periodically run mark & sweep GC to collect cycles

- Option 3: Require programmers to distinguish "weak pointers" from "strong pointers"
  - *weak pointers*: if all references to an object are "weak" then the object can be freed even with non-zero reference count.
  - "Back edges" in the object graph should be designated as weak
  - (Aside: weak pointers useful in GC settings too.)

Mark & Sweep / Copying

Traverse the Heap

Memory Use & Reachability

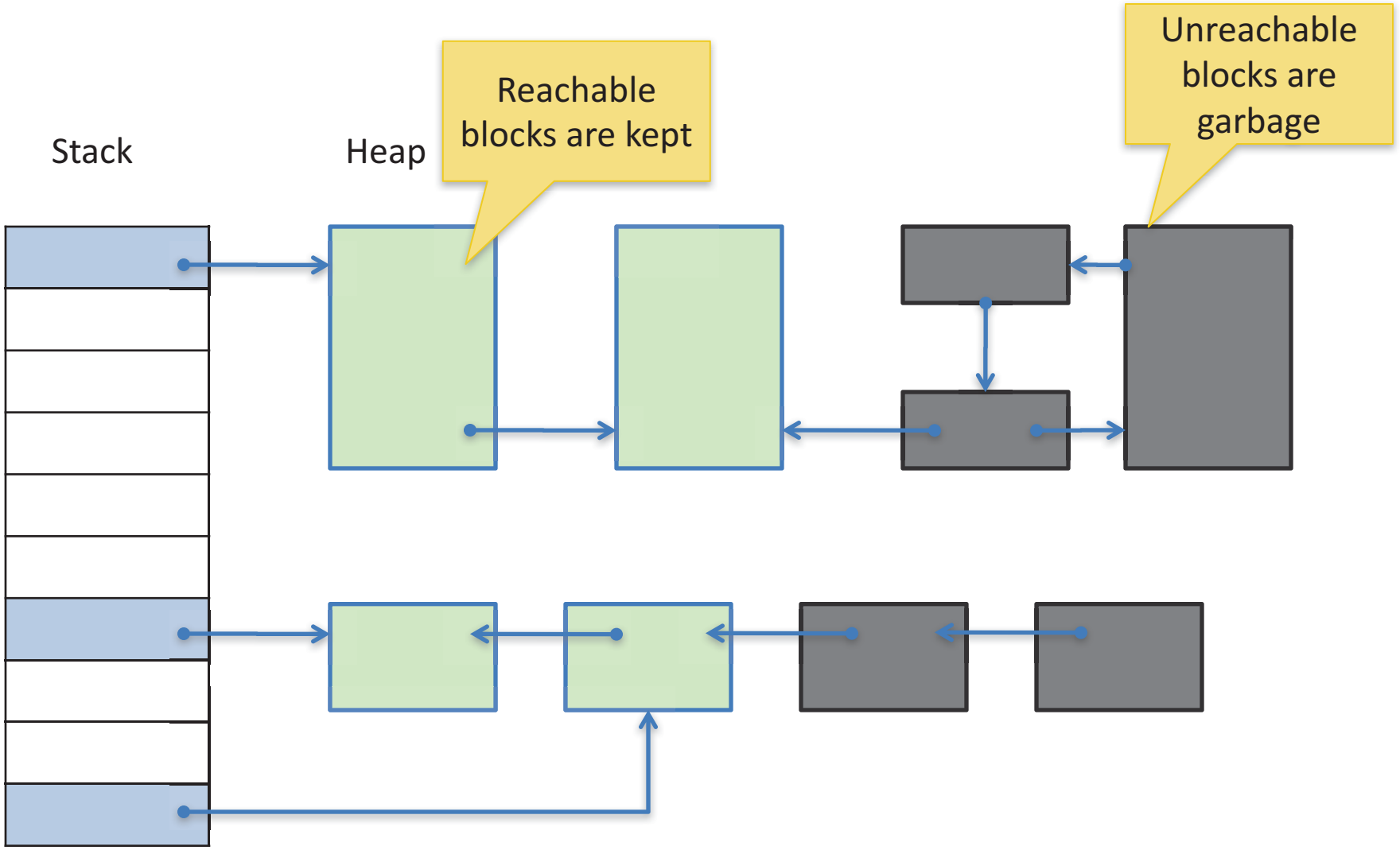
- When is a chunk of memory no longer needed?
  - In general, this problem is undecidable.

- We can approximate this information by freeing memory that can't be reached from any *root* references.
  - A *root reference* is one that might be accessible directly from the program (i.e. they're not in the heap).
  - Root references include (global) static fields and references in the stack.
- If an object can be reached by traversing pointers from a root, it is *live*.
- It is safe to reclaim all heap allocations not reachable from a root (such objects are *garbage* or *dead* objects).

Mark and Sweep Garbage Collection

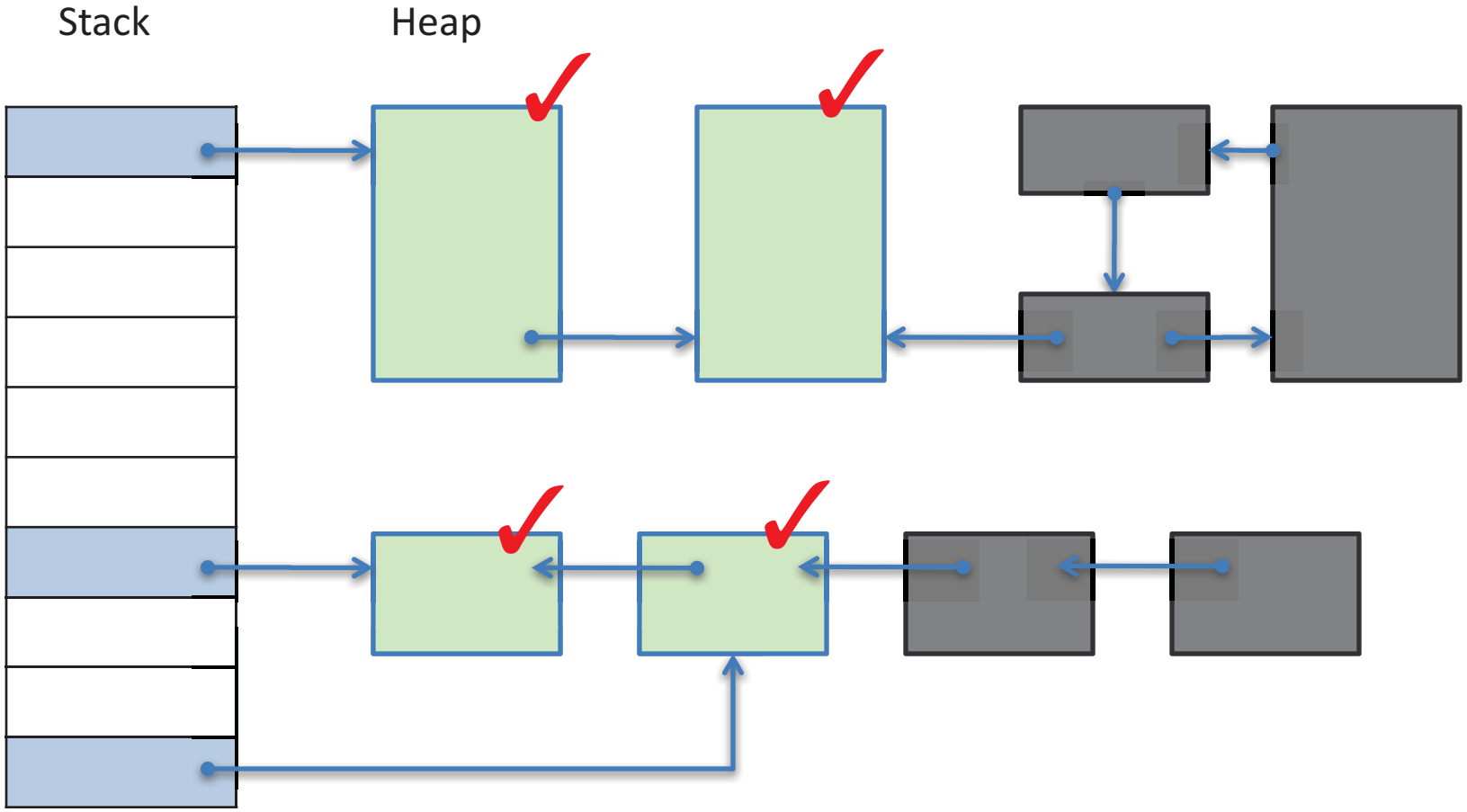
- Classic algorithm with two phases:
- Phase 1: Mark
  - Start from the roots
  - Do depth-first traversal, marking every object reached.
- Phase 2: Sweep
  - Walk over *all* allocated objects and check for marks.
  - Unmarked objects are reclaimed.
  - Marked objects have their marks cleared.
  - Optional: compact all live objects in heap by moving them adjacent to one another. (Needs extra work & indirection to “patch up” references)
- (In practice much more complex: "generational GC")

Results of Marking Graph



Second Phase: Drop "Unreachable"

- Sweep over all objects, dropping the ones marked as unreachable and keeping the ones marked reachable.



Costs & Implications

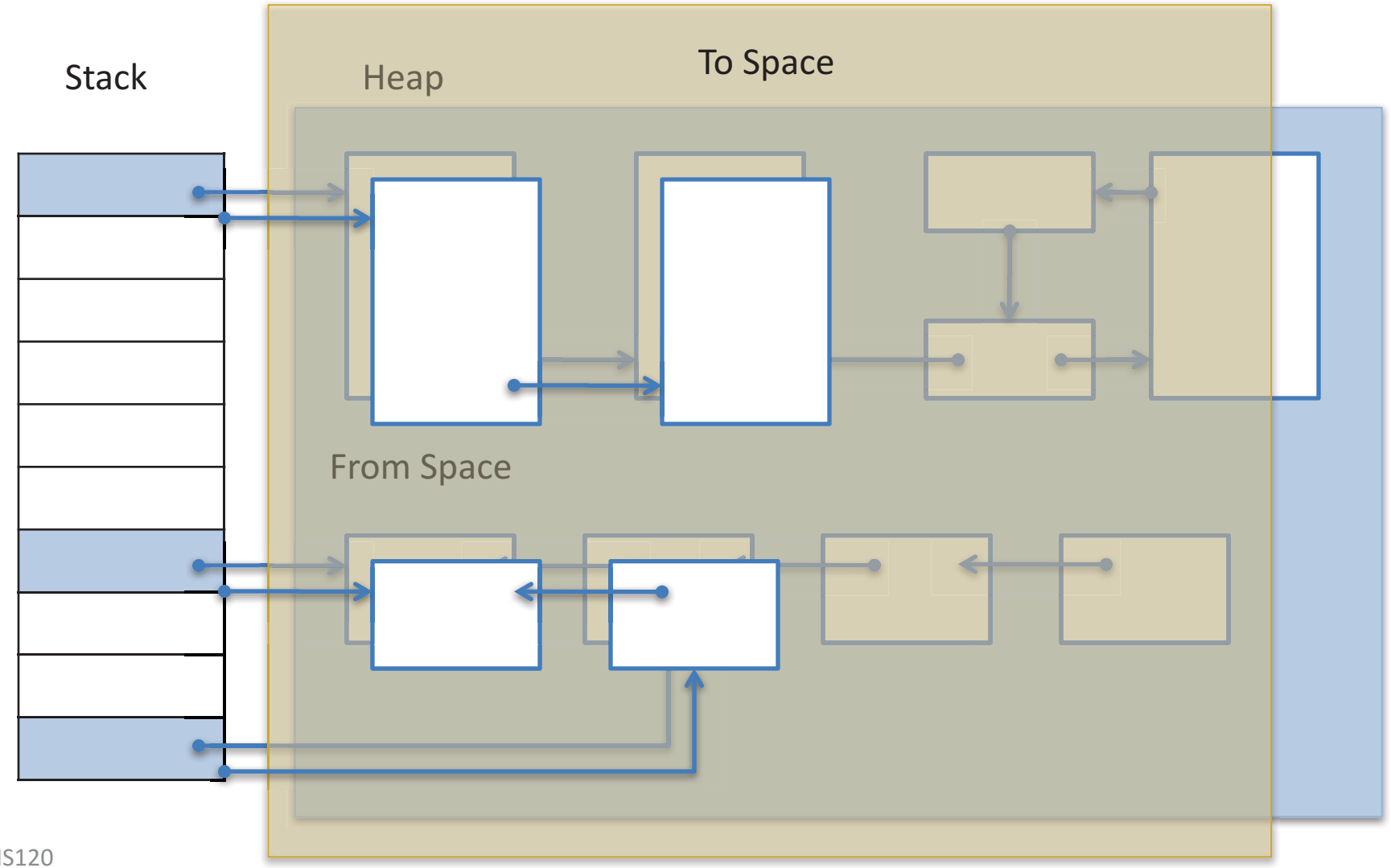
- Need to generalize to account for objects that have multiple outgoing pointers.
- Mark & Sweep algorithm reads all memory in use by the program (even if it's garbage!)
  - Running time is proportional to the total amount of allocated memory (both live and garbage).
  - Can pause the programs for long times during garbage collection.

Copying Garbage Collection

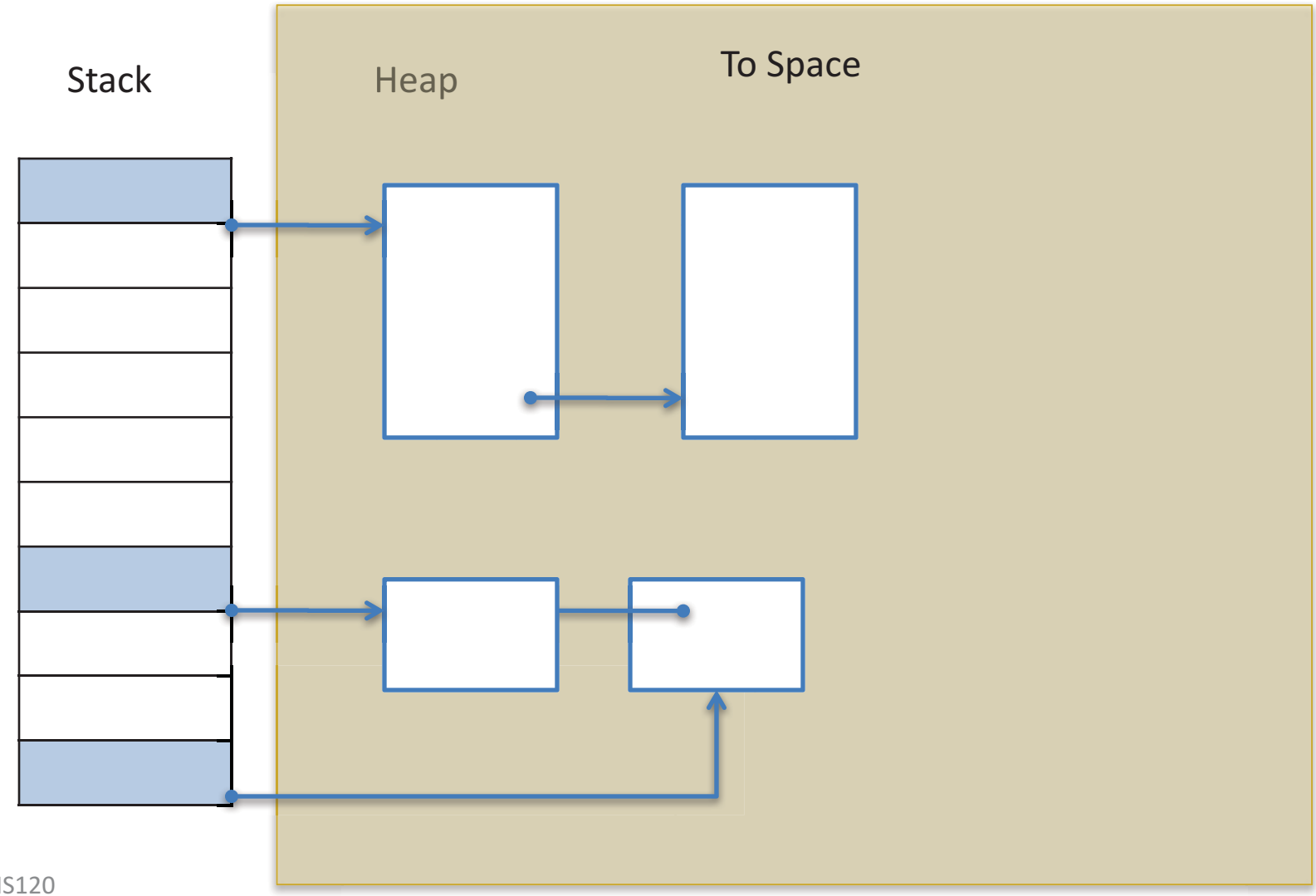
- Like mark & sweep: collects all garbage.
- Basic idea: use *two* regions of memory
  - One region is the memory in use by the program. New allocation happens in this region.
  - Other region is idle until the GC requires it.

- Garbage collection algorithm:
  - Traverse over live objects in the active region (called the "*from-space*"), copying them to the idle region (called the "*to-space*").
  - After copying all reachable data, switch the roles of the from-space and to-space.
  - All dead objects in the (old) from-space are discarded en masse.
  - A side effect of copying is that all live objects are compacted together.

Copy from "From" to "To"



Discard the "From Space"



GCDemo

See GCTest.java

Garbage Collection Take Aways

- Big idea: the Java runtime system tries to free-up as much memory as it can automatically.
  - Almost always a big win, in terms of convenience and reliability

- Sometimes can affect performance:
  - Lots of dead objects might take a long time to collect
  - When garbage collection will be triggered can be hard to predict, so there can be "pauses" (modern GC implementations try to avoid this!)
  - Global data structures can have references to "zombie" objects that won't be used, but are still reachable $\Rightarrow$ "space leak".

- There are many advanced programming techniques to address these issues:
  - Configuring the GC parameters
  - Explicitly triggering a GC phase
  - "Weak" references

Java 1.8 Functional Programming and Lambdas

or: How I Learned to Stop Worrying and Love Functional Programming in Java

(See PaintF.java)

Problem – Boilerplate Code in Java

- Using anonymous inner classes (Not great, but better than named classes)

```
quit.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }
});
```

- Using Lambdas (Much better!)

```
quit.addActionListener(e ->
        System.exit(0)
);
```

Lambdas – Why and What?

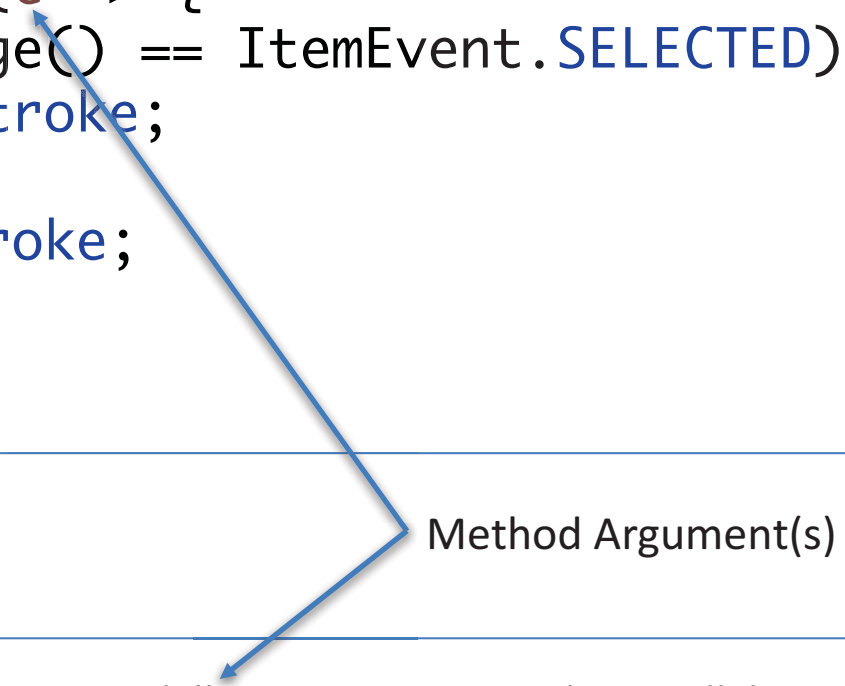
- Often implementation of anonymous classes is simple (e.g., an interface that contains only one method)
- Usually trying to pass functionality as an argument to another method, such as what action should be taken when someone clicks a button.
- Lambda expressions enable you to do this, to treat functionality as method argument, or code as data.

- https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

Lambdas – Why and What?

- Helps create instances of single-method classes more easily
- Think of them as anonymous methods

```
thick.addItemListener(e -> {
    if (e.getStateChange() == ItemEvent.SELECTED) {
        stroke = thickStroke;
    } else {
        stroke = thinStroke;
});
```



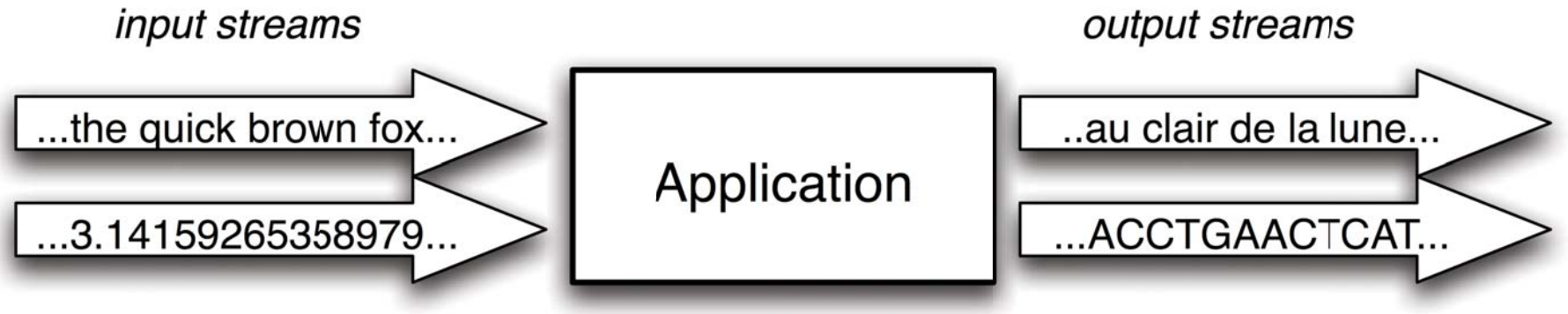

```
SwingUtilities.invokeLater(() -> new PaintF());
```

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)
  - Laziness-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(Object[]);
- The lines of a file can be obtained from BufferedReader.lines();
- Streams of file paths can be obtained from methods in Files;
- Streams of random numbers can be obtained from Random.ints();
- From static factory methods on the stream classes, such as Stream.of(Object[]), IntStream.range(int, int) or Stream.iterate(Object, UnaryOperator);
- Numerous other stream-bearing methods in the JDK, including BitSet.stream(), Pattern.splitAsStream(java.lang.CharSequence), and JarFile.stream().

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!)

Streams Operations

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

Lambdas, Streams, Pipelines

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

```
roster.stream()
      .filter(p ->
              p.getGender() == Person.Sex.MALE
              && 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++) {
   sum += list.get(i);
}
```

- 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)

- Less likely to have bug, much easier to parallelize

Programming Languages and Techniques (CIS120)

Lecture 40

December 11th, 2017

Semester Recap

Announcements 1

- Game Project Complete Code
  - Due: TONIGHT at midnight
- Schedule Grading Demo with your TAs!
  - You should have received an email
  - *30 point penalty if you miss your appointment and need to reschedule!*
  - *No points if you miss your appointment and do not reschedule.*
- Demo Days
  - Tuesday, December 12th noon – 2:00
  - Towne Building
  - Come take a look & you're welcome to demo your game!

Announcements 2

Final Exam:

- Friday, December 15th 6:00-8:00 PM
- 3 Locations:
  - Towne 100 Last Names A – G
  - Stiteler B6 Last Names H – Pa
  - Cohen G17 Last Names Pe -- Z

Exam Preparation

- *Comprehensive* exam over course concepts:
  - OCaml material (though we won't worry much about syntax)
  - All Java material (emphasizing material since midterm 2)
  - all course content *except lecture 33 (Code is data)*
- Closed book, but:
  - You may use one letter-sized, two-sided, *handwritten* sheet of notes during the exam.

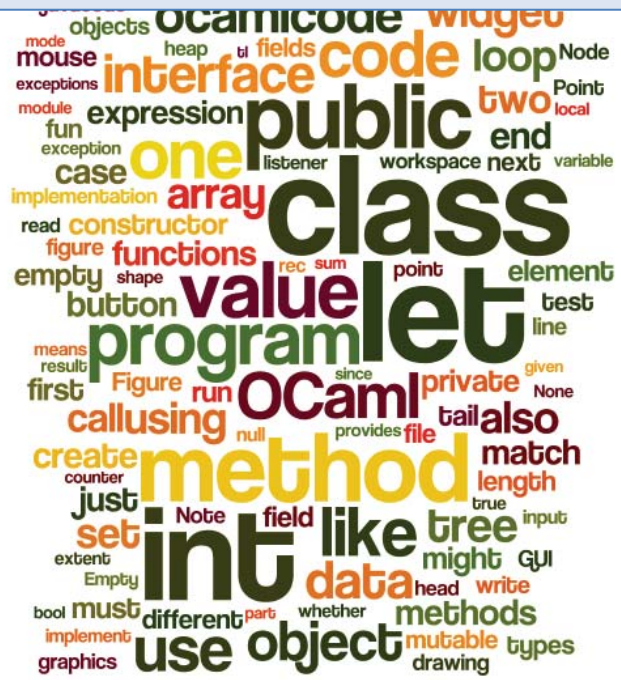
- Mock Exam:
  - Wednesday, December 13th 6:00-8:00PM
  - Towne 100
  - See Piazza

CIS 120 Recap

From Day 1

- CIS 120 is a course in **program design**
- Practical skills:
  - ability to write larger (~1000 lines) programs
  - increased independence ("working without a recipe")
  - test-driven development, principled debugging
- Conceptual foundations:
  - common data structures and algorithms
  - several different programming idioms
  - focus on modularity and compositionality
  - derived from first principles throughout
- It will be fun!

Promise: A *challenging* but *rewarding* course.



Which assignment was the most *challenging*?

1. OCaml finger exercises
2. DNA
3. Sets and Maps
4. Queues
5. GUI
6. Images
7. Chat
8. SpellChecker
9. Game

Which assignment was the most *rewarding*?

1. OCaml finger exercises
2. DNA
3. Sets and Maps
4. Queues
5. GUI
6. Images
7. Chat
8. SpellChecker
9. Game

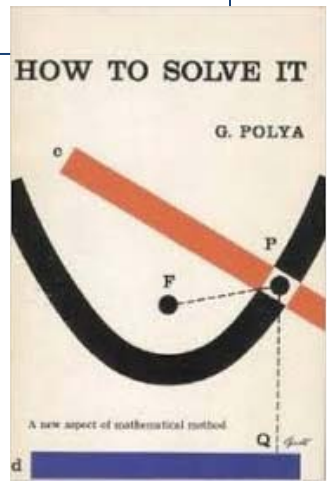
CIS 120 Concepts

13 concepts in 39 lectures

Concept: Design Recipe

1. Understand the problem
   What are the relevant concepts and how do they relate?
2. Formalize the interface
   How should the program interact with its environment?
3. Write test cases
   How does the program behave on typical inputs? On unusual ones? On erroneous ones?
4. Implement the required behavior
   Often by decomposing the problem into simpler ones and applying the same recipe to each

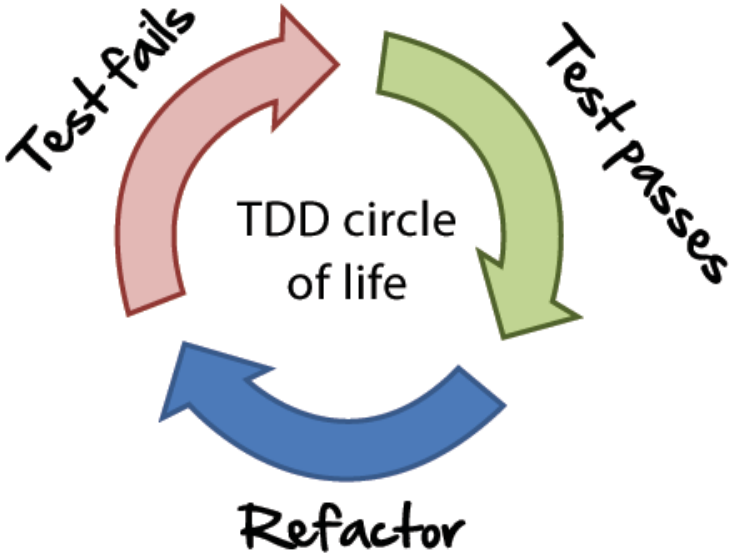
"Solving problems", wrote Polya, "is a practical art, like swimming, or skiing, or playing the piano: You can learn it only by imitation and practice."



Testing

- Concept: Write tests *before* coding
  - *"test first"* methodology

- Examples:
  - Simple assertions for declarative programs (or subprograms)
  - Longer (and more) tests for stateful programs / subprograms
  - Informal tests for GUIs (can be automated through tools)

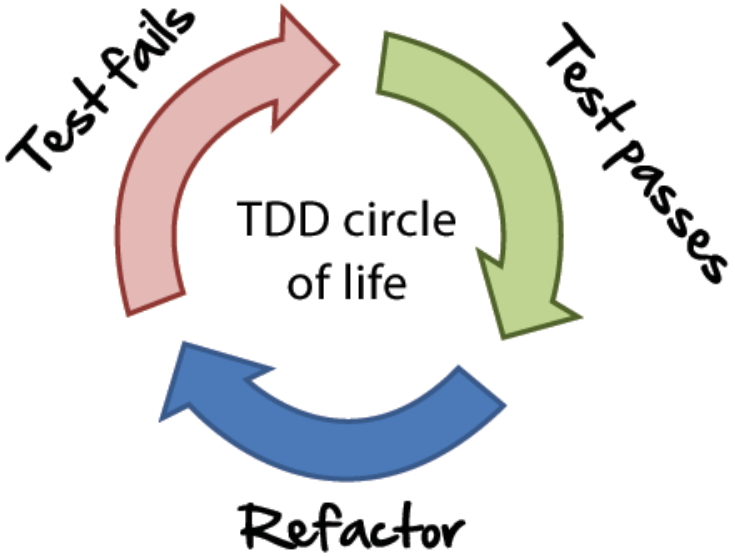


- Why?
  - Tests clarify the specification of the problem
  - Helps you understand the *invariants*
  - Thinking about tests informs the implementation
  - Tests help with extending and refactoring code later
  - Industry practice; useful for coordinating teams

Testing

- Concept: Write tests *before* coding
  - *"test first"* methodology

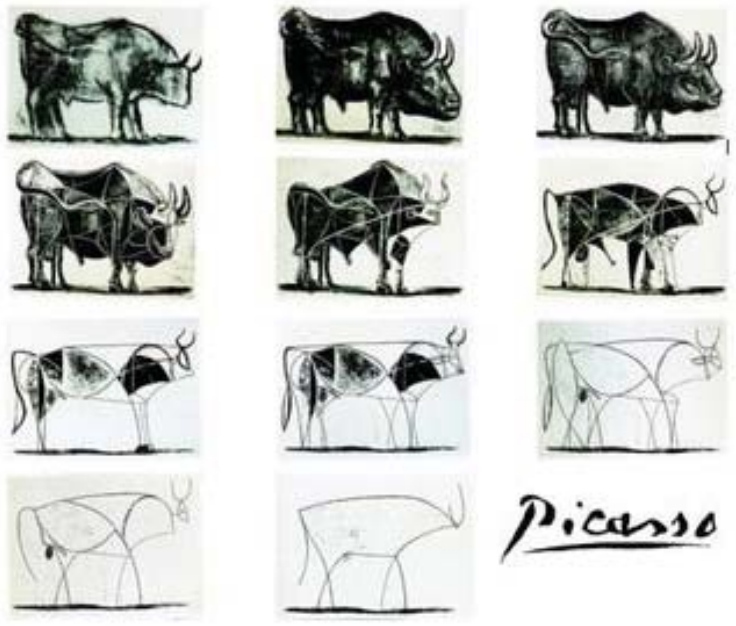
- Examples:
  - Simple assertions for declarative programs (or subprograms)
  - Longer (and more) tests for stateful programs / subprograms
  - Informal tests for GUIs (can be automated through tools)



- Why?
  - Tests clarify the specification of the problem
  - Helps you understand the *invariants*
  - Thinking about tests informs the implementation
  - Tests help with extending and refactoring code later
  - Industry practice; useful for coordinating teams

Functional/Procedural Abstraction

- Concept: *Don't Repeat Yourself!*
  - Find ways to generalize code so it can be reused in multiple situations



Pablo Picasso, Bull (plates I - XI) 1945

- Examples: Functions/methods, generics, higher-order functions, interfaces, subtyping, abstract classes

- Why?
  - Duplicated functionality = duplicated bugs
  - Duplicated functionality = more bugs waiting to happen
  - Good abstractions make code easier to read, modify, maintain

Persistent data structures

- Concept: Store data in *persistent, immutable* structures; implement computation as *tra* structures
- Examples: immutable lists and images, Strings, Streams in Jav
- Why?
  - Simple model of comp
  - Simple interface: D ave to re communication ween various parts of the program, all interfaces are explicit)
  - *Recursion* amenable to mathematical analysis (CIS 160/121)
  - Plays well with parallelism

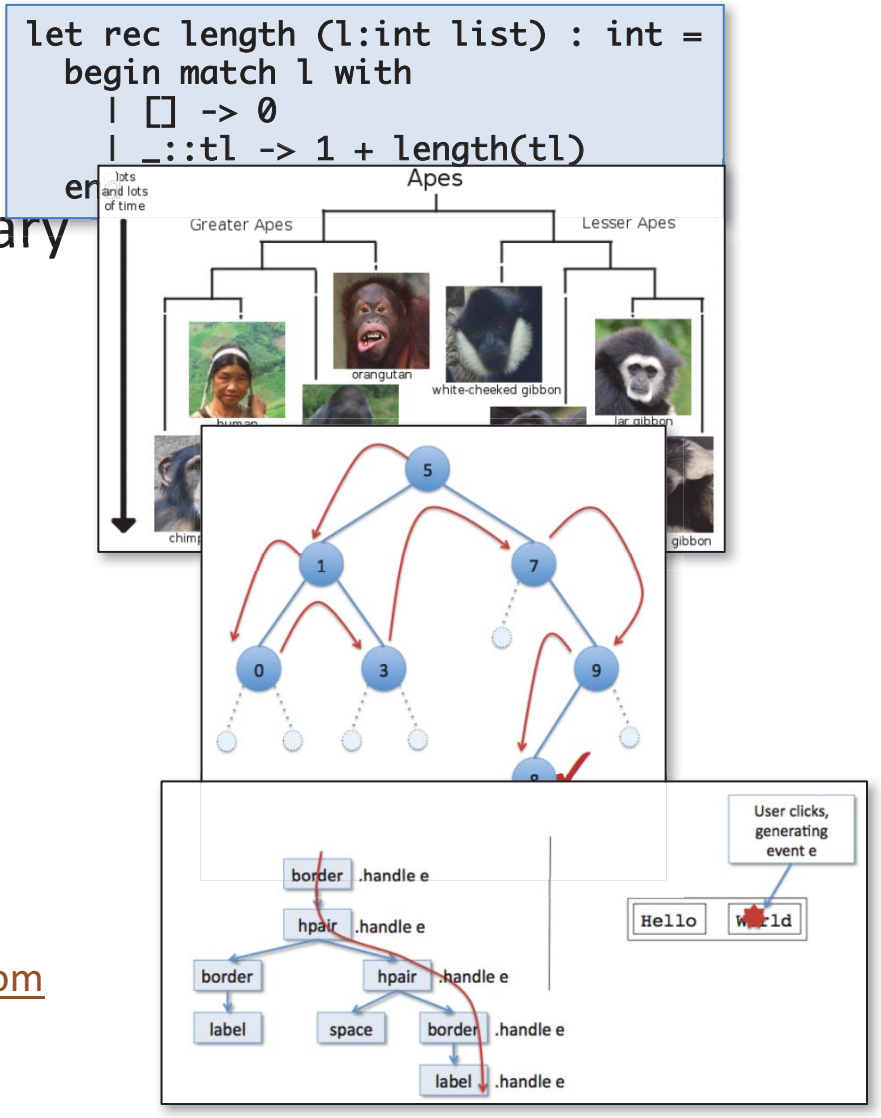
Recursion is the natural way of computing a function $f(t)$ when t belongs to an inductive data type:

1. Determine the value of f for the base case(s).
2. Compute f for larger cases by combining the results of recursively calling f on smaller cases.
3. Same idea as mathematical induction (a la CIS 160)

Concept: Tree Structured data

- Lists (i.e. “unary” trees)
- Simple binary trees
- Trees with invariants: e.g. binary search trees
- Widget trees: screen layout + event routing
- Swing components

- Why? Trees are ubiquitous in CS!
  - file system organization
  - languages, compilers
  - domain name hierarchy www.google.com



First-class computation

- Concept: *code is a form of data* that can be defined by functions, methods, or objects (including anonymous ones), stored in data structures, and passed to other functions
- Examples: map, filter, fold (HW4), pixel transformers (HW6), event listeners (HW5, 7, 9)

```
cell.addMouseListener(new MouseAdapter() {
    public voi
        select
    }
});
```

```
cell.addMouseListener(e -> {
            selectCell(cell);
});
```

- Why?
  - Powerful tool for abstraction: can factor out design patterns that differ only in certain computations
  - Heavily used for reactive programming, where data structures store "reactions" to various events

Types, Generics, and Subtyping

- Concept: *Static type systems* prevent errors. Every expression has a static type, and OCaml/Java use the types to rule out buggy programs. *Generics* and *subtyping* make types more flexible and allow for better code reuse.

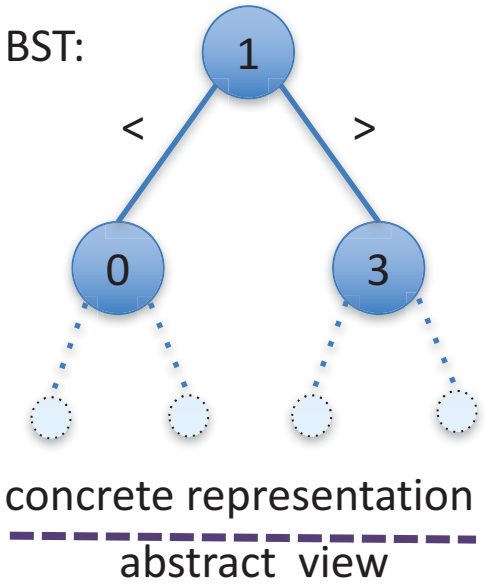
```
let rec contains (x:'a) (l:'a list) : bool =
  begin match l with
    | [] -> false
    | h::tl -> x = a || (contains x tl)
  end
```

- Why?
  - Easier to fix problems indicated by a type error than to write a test case and then figure out why the test case fails
  - Promotes refactoring: type checking ensures that basic invariants about the program are maintained

Abstract types and encapsulation

- Concept: *Type abstraction* hides the actual implementation of a data structure, describes a data structure by its interface (what it does vs. how it is represented), supports reasoning with invariants

- Examples: Set/Map interface (HW3), queues in …and access

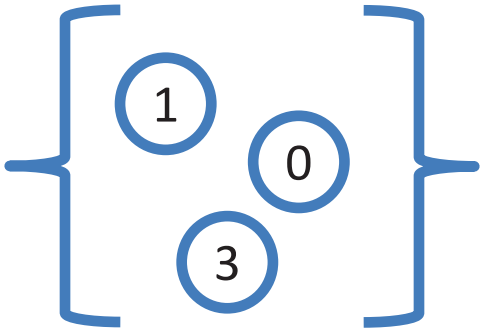


Invariants are a crucial tool for reasoning about data structures:

1. *Establish* the invariants when you create the structure.
2. *Preserve* the invariants when you modify the structure.

…entation without

… about the

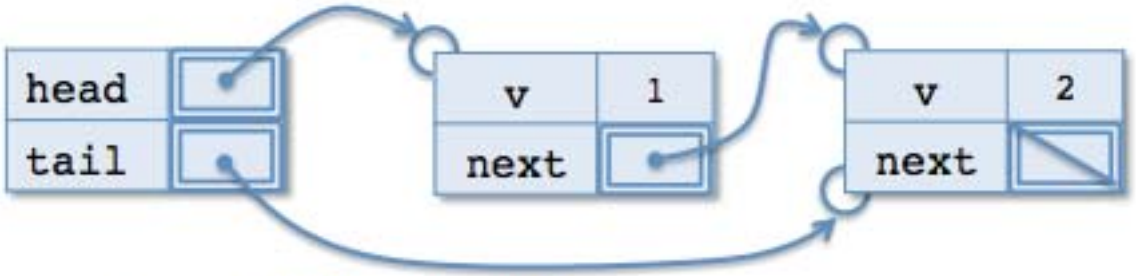


Mutable data

- Concept: Some data structures are *ephemeral*: computations mutate them over time

- Examples: queues, deques (HW4), GUI state (HW5, 9), arrays (HW 6), dynamic arrays, dictionaries (HW8)

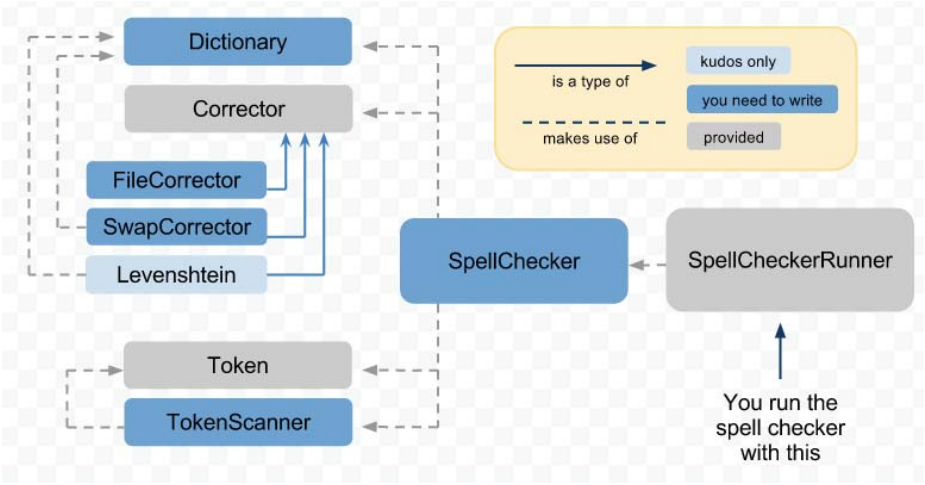
- Why?
  - Common in OO programming, which simulates the transformations that objects undergo when interacting with their environment
  - Heavily used for event-based programming, where different parts of the application communicate via shared state
  - Default style for Java libraries (collections, etc.)



A queue with two elements

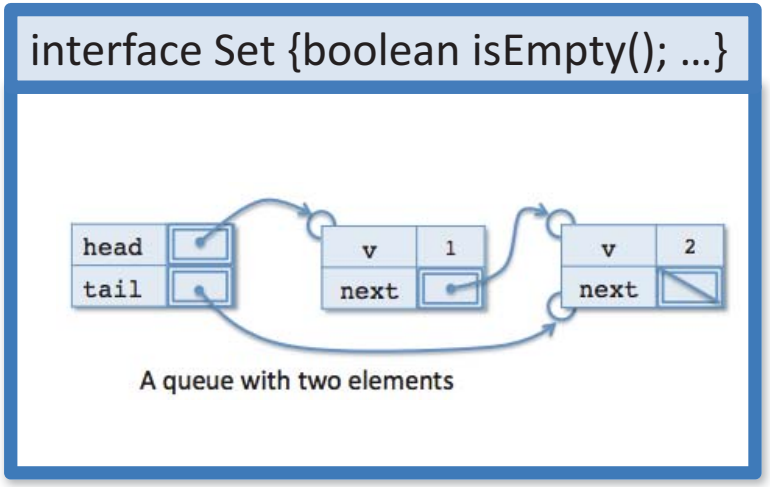
Sequences, Sets, Maps

- Concept: Specific **abstract data types** of sequences, sets, and finite maps
- Examples: HW3, Java Collections, HW 7, 8
- Why?
  - These abstract data types come up again and again
  - Need aggregate data structures (collections) no matter what language you are programming in
  - Need to be able to choose the data structure with the right semantics



Lists, Trees, BSTs, Queues, and Arrays

- Concept: There are *implementation trade-offs* for abstract types
- Examples:
  - Binary Search Trees vs. Lists vs. Hashing for sets and maps
  - Linked lists vs. Arrays for sequential data
- Why?
  - Abstract types have multiple implementations
  - Different implementations have different trade-offs. Need to understand these trade-offs to use them well.
  - For example: BSTs use their invariants to speed up lookup operations compared to linked lists.



Lookup 'a'

Lookup 'a'

Lookup 'b'

Lookup 'b'

Do the Function call

Save Workspace; push l1, l2

Lookup l1

Lookup l1

Match Expression

Nil case Doesn't Match

Cons case *Does* Match

Simplify the Branch: push h, t

Lookup 'h'

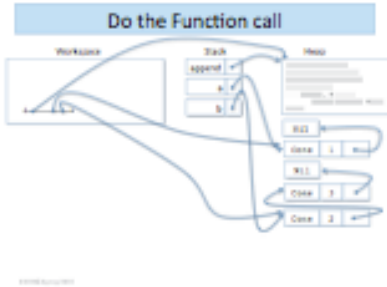
Lookup 'h'

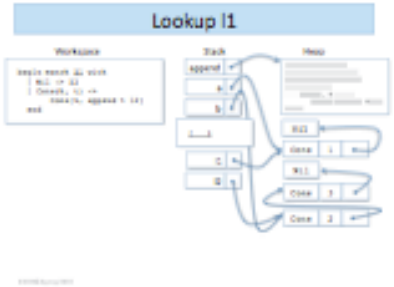
Lookup 'append'

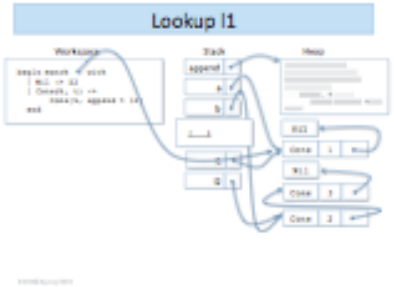
Lookup 'append'

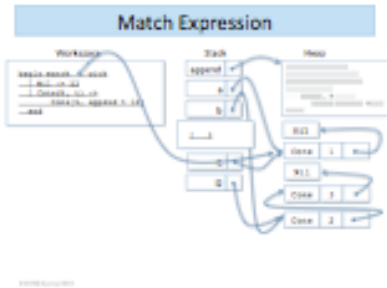
Abstract Stack Machine

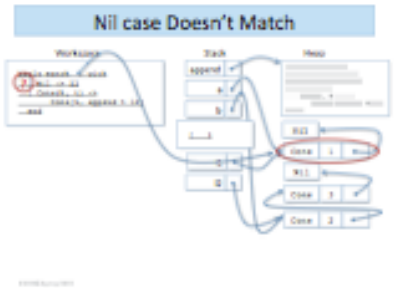
- Concept: The *Abstract Stack Machine* is a detailed model of the execution of OCaml/Java

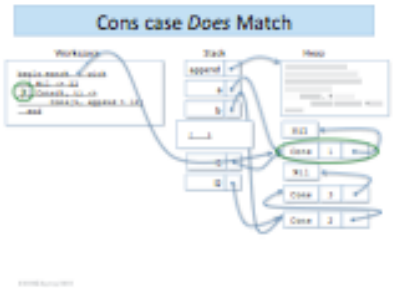


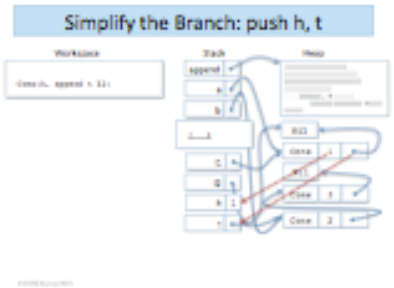


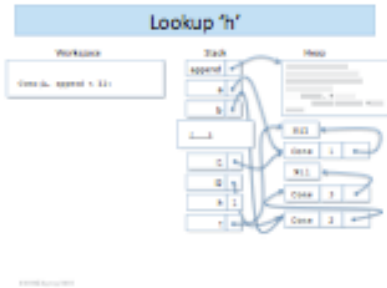


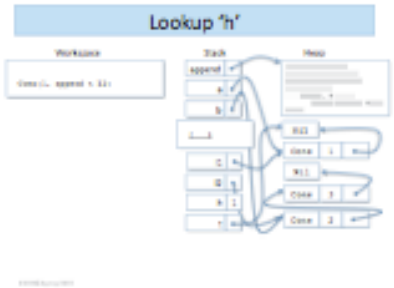


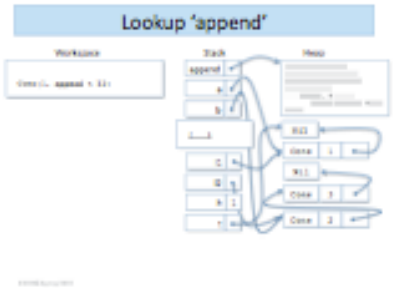


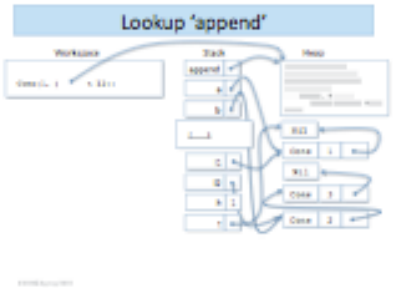












Abstract Stack Machine

- Concept: The *Abstract Stack Machine* is a detailed model of the execution of OCaml/Java

- Example: throughout the semester!

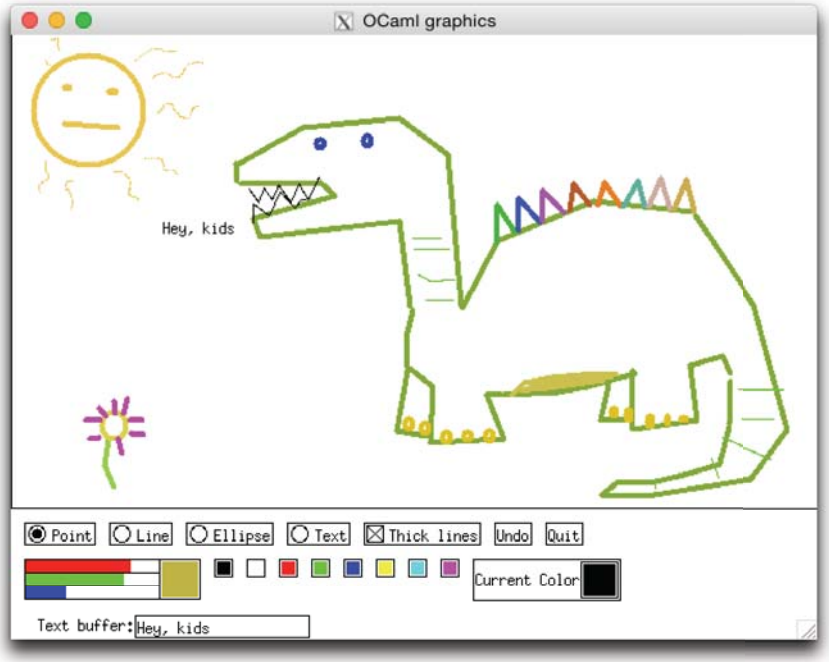
- Why?
  - To know what your program does without running it
  - To understand tricky features of Java/OCaml language (aliasing, first-class functions, exceptions, dynamic dispatch)
  - To help understand the programming models of other languages: Javascript, Python, C++, C#, …
  - To predict performance and space usage behaviors

Event-Driven programming

- Concept: Structure a program by associating "handlers" that *react to program events*. Handlers typically interact with the rest of the program by modifying shared state.

- Examples: GUI programming in OCaml and Java

- Why?
  - Practice with reasoning about shared state
  - Practice with first-class functions
  - Necessary for programming with Swing
  - Common in GUI applications



Why OCaml?

Why some other language than Java?

- Level playing field for students with varying backgrounds coming into the same class
- Two points of comparison allow us to emphasize language-independent concepts
- Learn concepts that generalize *across* diverse languages.

…but, why specifically OCaml?

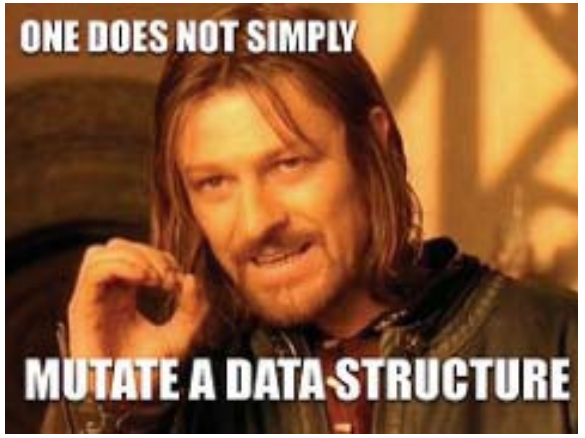
Rich, orthogonal vocabulary

- In Java: int, A[], Object, Interfaces
- In OCaml:
  - primitives
  - arrays
  - objects
  - datatypes (including lists, trees, and options)
  - records
  - refs
  - first-class functions
  - abstract types
- All of the above *can* be implemented in Java, but untangling various use cases of objects is subtle
- Concepts (like generics) can be studied in isolation, fewer intricate interactions with the rest of the language

Functional Programming

- In Java, every reference is mutable and optional by default
- In OCaml, persistent data structures are the default. Furthermore, the type system keeps track of what is and is not mutable, and what is and is not optional

- Advantages of immutable/persistent data structures
  - Don't have to keep track of aliasing. Interface to the data structure is simpler
  - Often easier to think in terms of "transforming" data structures than "modifying" data structures
  - Simpler implementation (Compare lists and trees to queues and deques)
  - Powerful evaluation model (substitution + recursion).



Why Java?

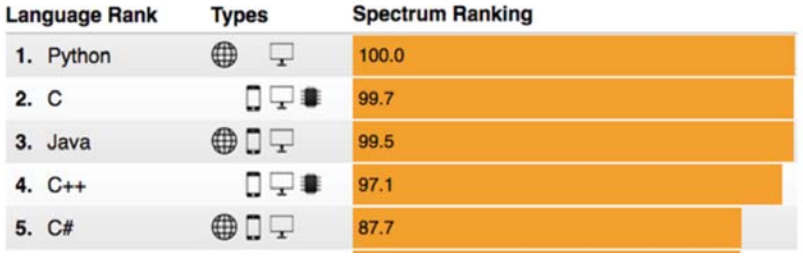
Object Oriented Programming

- Provides a different way of decomposing programs
- Basic principles:
  - Encapsulation of local, mutable state
  - Inheritance to share code
  - Dynamic dispatch to select which code gets run

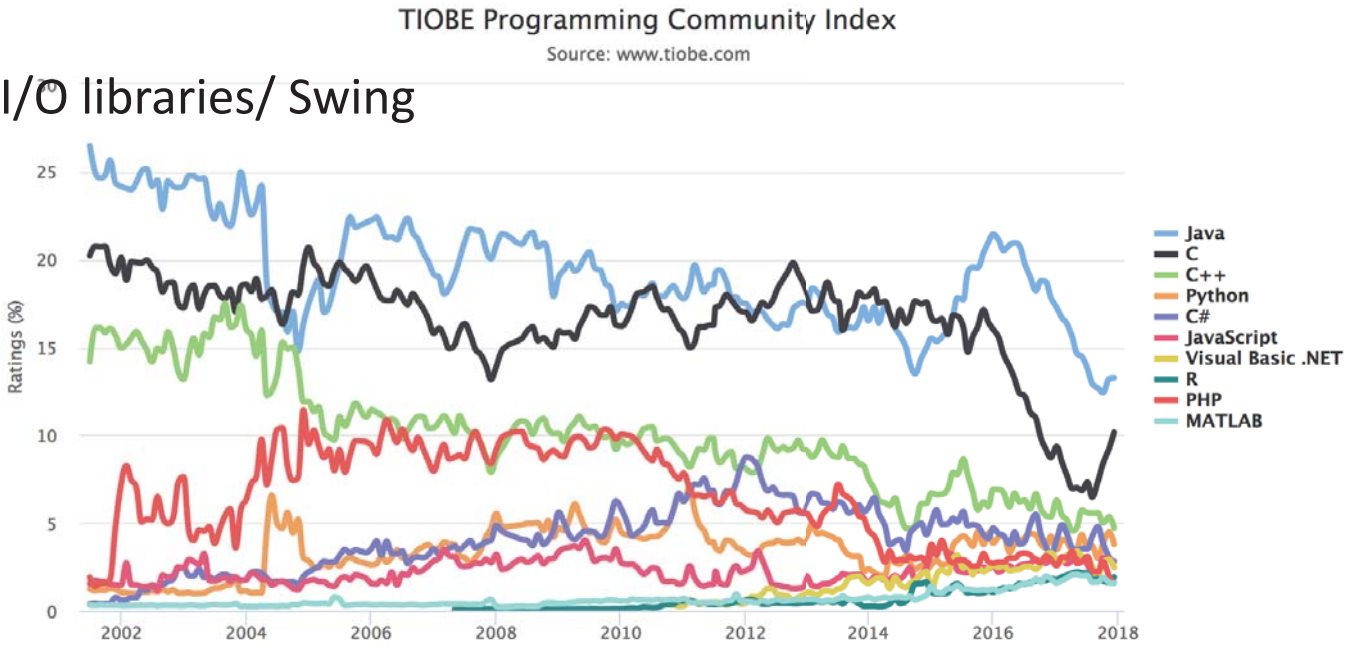
- but why specifically Java?

Important Ecosystem

- Canonical example of OO language design
- Widely used: Desktop / Server / Android / etc.
- Industrial strength tools
  - Eclipse
  - JUnit testing framework
  - Profilers, debuggers, …
- Libraries:
  - Collections / I/O libraries/ Swing
  - …
- In-demand job skill



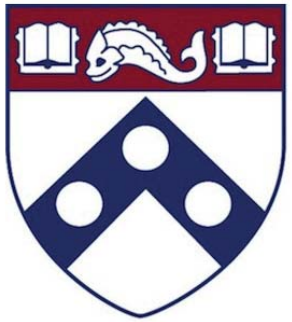
Language Rank	Types	Spectrum Ranking
1. Python		100.0
2. C		99.7
3. Java		99.5
4. C++		97.1
5. C#		87.7



Onward…

What Next?

- Classes:
  - CIS 121, 262, 320 – data structures, performance, computational complexity
  - CIS 19x – programming languages
    - C++, C#, Python, Haskell, Ruby on Rails, iPhone programming
  - CIS 240 – lower-level: hardware, gates, assembly, C programming
  - CIS 341 – compilers (projects in OCaml)
  - CIS 371, 380 – hardware and OS's
  - CIS 552 – advanced programming in Haskell
  - And many more!

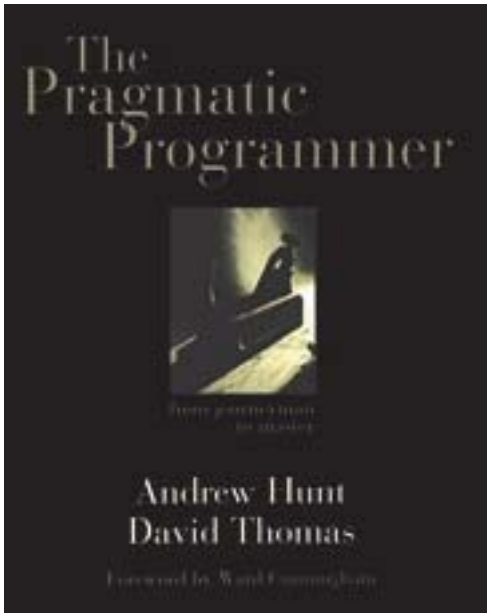


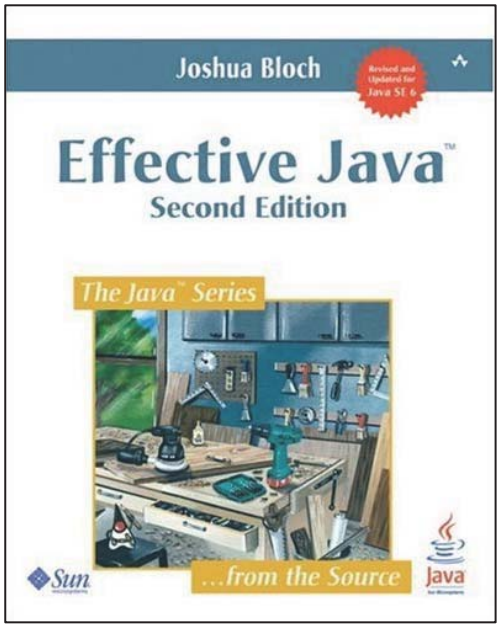
Penn Engineering

The Craft of Programming

- *The Pragmatic Programmer:*
  From Journeyman to Master

  by Andrew Hunt and David Thomas
  - Not about a particular programming language, it covers style, effective use of tools, and good practices for developing programs.

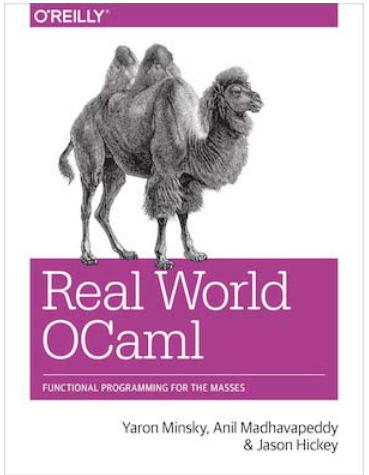




- *Effective Java*
  by Joshua Bloch
  - Technical advice and wisdom about using Java for building software. The views we have espoused in this course share much of the same design philosophy.

Functional Programming

- *Real World OCaml*
  by Yaron Minsky, Anil Madhavpeddy,
  and Jason Hickey
  - Using OCaml in practice: learn how to leverage its rich types, module system, libraries, and tools to build reliable, efficient software.
  - https://realworldocaml.org/



- Explore related Languages:

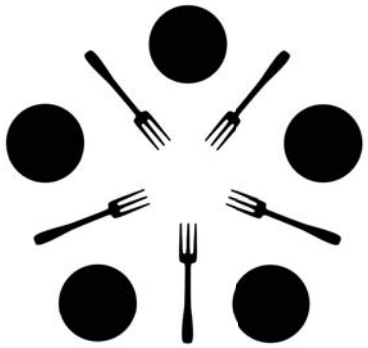
Haskell Scala Clojure

Conferences / Videos / Blogs

- curry-on.org
- cufp.org Commercial Users of Functional Programming
  - See e.g. Manuel Chakravarty's talk "A Type is Worth a Thousand Tests"
- Yaron Minsky's Jane Street Tech Blog
  - Ocaml in practice
- PHASE – Philly Area Scala Enthusiasts

- Join us! Penn's PL Club plclub.org

Ways to get Involved



dining philosophers

UNIVERSITY OF PENNSYLVANIA | COMPUTER SCIENCE CLUB

PENN APPS

PENN PL

plclub.org

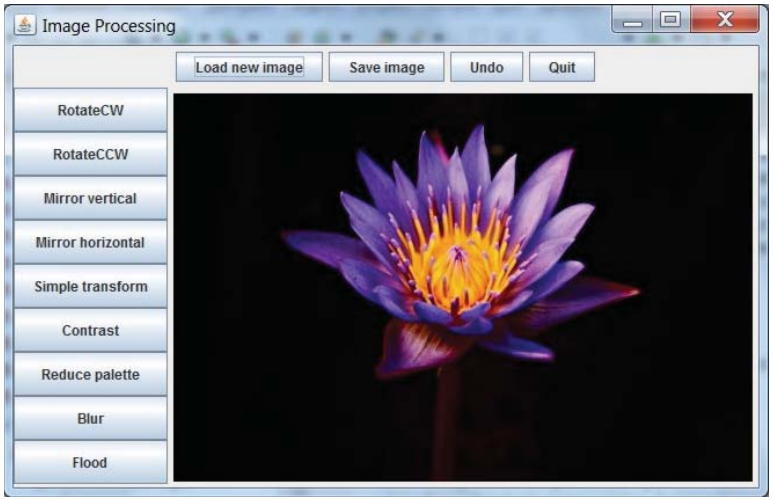
WOMEN IN CS

Become a TA!

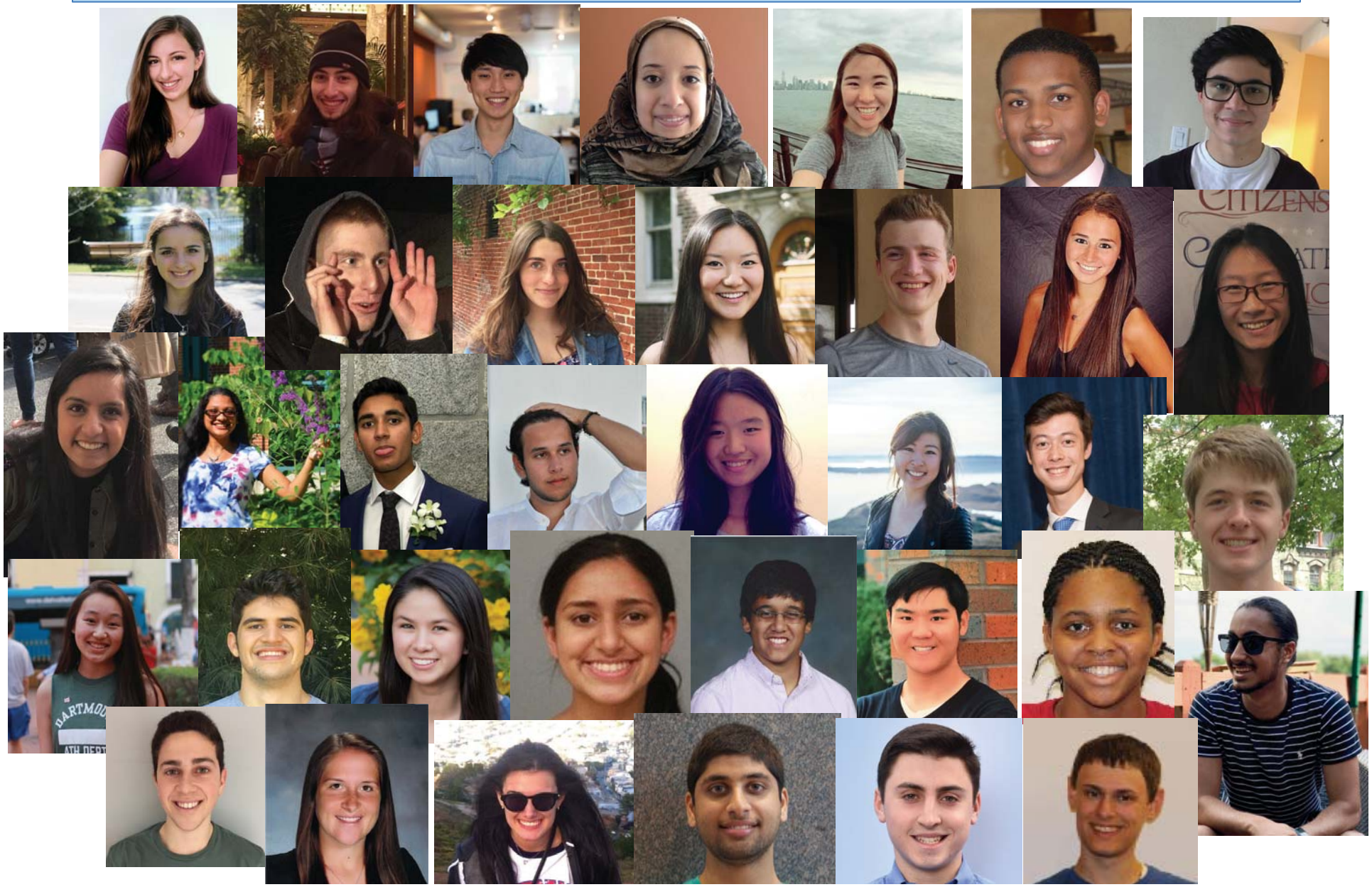
Undergraduate Research

Parting Thoughts

- Improve CIS 120:
  - End-of-term survey will be sent soon
  - Penn Course evaluations also provide useful feedback
  - We take them seriously: please complete them!

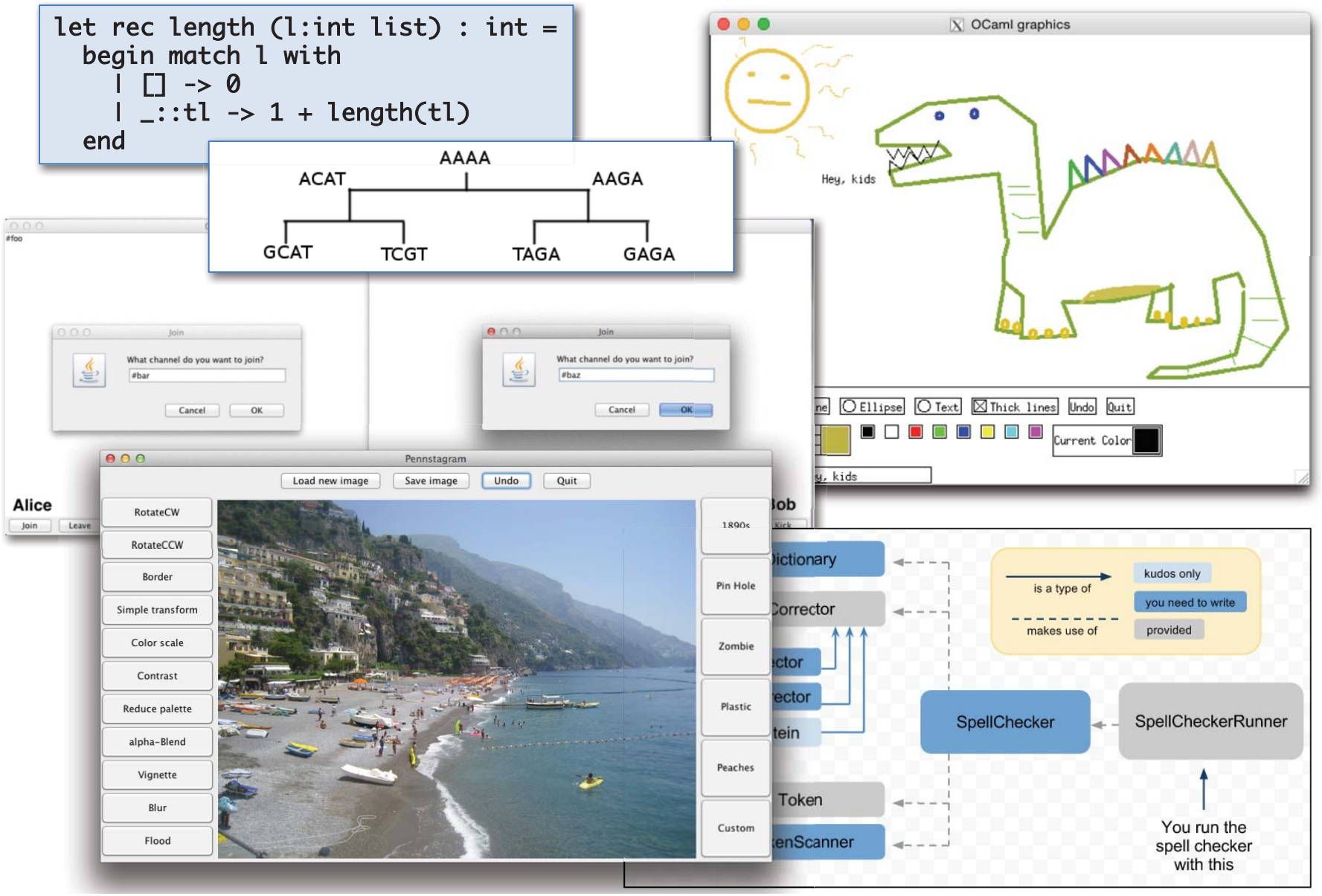


Many Thanks to the Tas!!!!!



Thanks!

```
let rec length (l:int list) : int =
  begin match l with
    | [] -> 0
    | _::tl -> 1 + length(tl)
  end
```



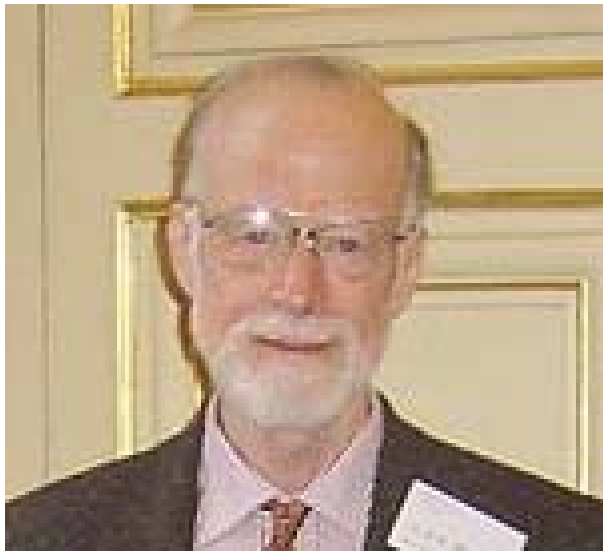
Who uses OCaml?

Did you attend class today?

1. yes
2. yes
3. yes
4. yes
5. maybe

The Billion Dollar Mistake

"***I call it my billion-dollar mistake. It was the invention of the null reference in 1965.*** *At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement.* ***This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.*** *In recent years, a number of program analysers like PREfix and PREfast in Microsoft have been used to check references, and give warnings if there is a risk they may be non-null. More recent programming languages like Spec# have introduced* ***declarations for non-null references****. This is the solution, which I rejected in 1965. "*



Sir Tony Hoare, QCon, London 2009

Better interfaces: Optional values

- In Java, optional values are the default. *Any* reference type could be null.

- In OCaml, references are non-null by default and optional values must be specified by the programmer. Only values of type 'a option can be None.

- In Java, every method must specify what it does if its arguments are null. Many of them don't.
- In OCaml, the type of a method tells you whether an argument may be null. We didn't have to think about optional values until homework 5!

Fundamental abstract types

- An *abstract* type is defined by its *interface* not its *implementation*.
- Flexibility: interface can change without modification to clients
- Security: implementation invariants can be preserved

- In OCaml, direct expression of abstract data types through modules and signatures
- In Java, make types abstract via access modifiers (private), provide flexibility through interfaces