CS106A Final Exam Review Session

Saturday Dec. 9, 2017

Nick Troccoli

Today's Topic List

- Primitives, Objects and Traces
- Graphics + Animation
- Event-Driven Programs
- Strings + chars
- Classes + Interfaces
- Arrays, ArrayLists and HashMaps
- File Reading

Primitives, Objects and Traces

Primitives and Objects

- **Primitives:** int, double, boolean, char,…
- **Objects**: GRect, GOval, GLine, int[], … (anything with **new**, and that you call methods on)
- Treat Strings as primitives

Parameters

- **When passing parameters, make a copy of whatever is on the stack.**
- **Primitives:** the *actual value* is on the stack (pass by value)
- **Objects:** a *heap address* where the information lives is on the stack. (pass by reference)

Parameters - Primitives

```
public void run() {
      int x = 2;
      addTwo(x);
      println(x);  // x is still 2!
}

private void addTwo(int y) {
      y += 2;
}
```

Parameters - Objects

```
public void run() {
      GRect rect = new Grect(0,0,50,50);
      fillBlue(rect);
      add(rect);     // rect is blue!
}

private void fillBlue(GRect rect) {
      rect.setFilled(true);
      rect.setColor(Color.BLUE);
}
```

Program Traces

- **Approaching program traces**
  - Local variables are *separate* across methods
  - Parameters are just assigned names by the order in which they're passed
  - Write values above variable names as you go through the program (or draw stack frame boxes)
  - Pass-by-reference vs. pass-by-value

```java
private void mystery(int[][] arr) {
    hoHoHo(arr[0]);
    milkAndCookies(arr[1][1]);
}

private void hoHoHo(int[] arr) {
    for (int i = 1; i < arr.length; i++) {
        arr[i] += arr[i - 1];
    }
}

private void milkAndCookies(int num) {
    int milk = 2*num;
    int cookies = num % 2;
    num = milk + cookies;
}
```

0	1	2
3	4	5

Input to **mystery()**
What is **arr** after?

Take 1

```
private void mystery(int[][] arr) {
    hoHoHo(arr[0]);
    milkAndCookies(arr[1][1]);
}

private void hoHoHo(int[] arr) {
    for (int i = 1; i < arr.length; i++) {
        arr[i] += arr[i - 1];
    }
}

private void milkAndCookies(int num) {
    int milk = 2*num;
    int cookies = num % 2;
    num = milk + cookies;
}
```

0	**1**	**2**
3	**4**	**5**

Input to **mystery()**
What is **arr** after?

ANSWER:

0	**1**	**3**
3	**4**	**5**

Take 1

```
private void mystery(int[][] arr) {
    hoHoHo(arr[0]);
    arr[1][1] = milkAndCookies(arr[1][1]);
}

private void hoHoHo(int[] arr) {
    for (int i = 1; i < arr.length; i++) {
        arr[i] += arr[i - 1];
    }
}

private int milkAndCookies(int num) {
    int milk = 2*num;
    int cookies = num % 2;
    num = milk + cookies;
    return num;
}
```

0	1	2
3	4	5

Input to **mystery()**
What is **arr** after?

Take 2

```java
private void mystery(int[][] arr) {
    hoHoHo(arr[0]);
    arr[1][1] = milkAndCookies(arr[1][1]);
}

private void hoHoHo(int[] arr) {
    for (int i = 1; i < arr.length; i++) {
        arr[i] += arr[i - 1];
    }
}

private int milkAndCookies(int num) {
    int milk = 2*num;
    int cookies = num % 2;
    num = milk + cookies;
    return num;
}
```

0	**1**	**2**
3	**4**	**5**

Input to **mystery()**
What is **arr** after?

ANSWER:

0	**1**	**3**
3	**8**	**5**

Take 2

Graphics and Animation

Graphics

- Look at lecture slides for lists of different GObject types and their methods
- Remember: the x and y of GRect, GOval, etc. is their **upper left corner**, but the x and y of GLabel is its **leftmost baseline coordinate**.
- Remember: a label's height is gotten from **getAscent**.

Animation

Standard format for animation code:

```
while (CONDITION) {
      updateGraphics();
      performChecks();
      pause(PAUSE_TIME);
}
```

Event-Driven Programs

Events

- **Three** types of events: keyboard, mouse, interactors
- **Two** ways for Java to execute your code: from run() and from event handlers (mouseClicked, mouseMoved, actionPerformed, etc.)
- These programs are **asynchronous**; your code is not run in order anymore, since you don't know when the user will interact with your program!

Mouse + Key Events

1. Sign up for notifications for key or mouse events
2. Implement the method corresponding to what event you care about (e.g. mousePressed, mouseMoved)
3. Java will call that method whenever the corresponding event occurs.

Interactors

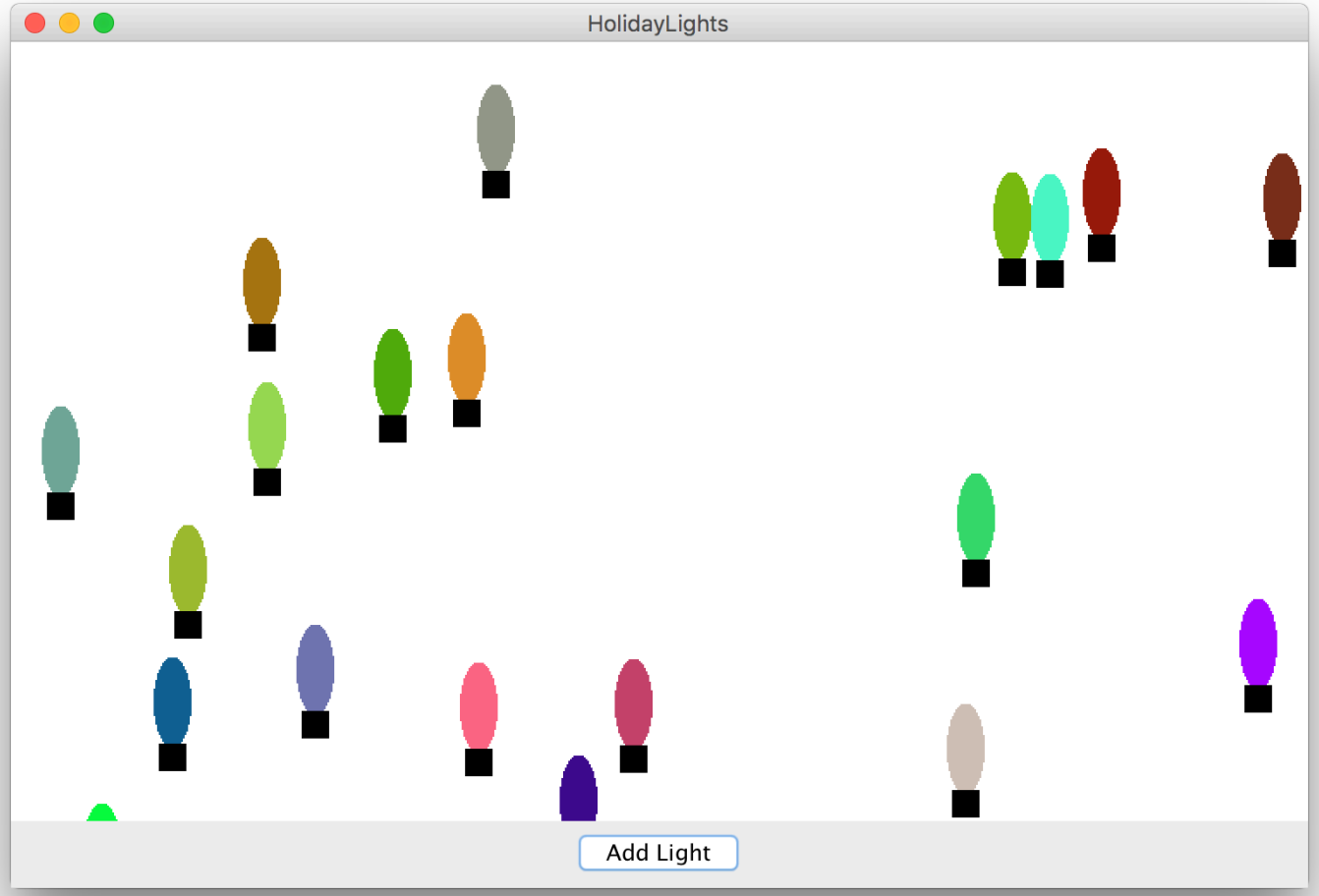
1. Add interactors in init()
2. addActionListeners() to listen for button presses
3. .addActionListener(this) on text fields for ENTER
4. Implement actionPerformed
5. Java will call actionPerformed whenever an action event occurs.

Interactors

```
public void actionPerformed(ActionEvent e) {
      if (e.getActionCommand().equals("My Button")) {
            ...
      }
}

// ... equivalent to ...
public void actionPerformed(ActionEvent e) {
      if (e.getSource() == myJButton) {
            ...
      }
}
```

Holiday Lights!



Holiday Lights

- Add a new holiday light by clicking “Add Light”
- Every second, every light on screen changes to a new random color

```
private static final int BULB_V_RADIUS = 25;
private static final int BULB_H_RADIUS = 10;
private static final int BULB_BASE_SIZE = 15;
private static final int PAUSE_TIME = 1000;
```

Instance Variables

```
// List of all bulbs on the screen
private ArrayList<GOval> bulbs = new ArrayList<GOval>();

// To change the bulb color and position bulbs
private RandomGenerator rgen = RandomGenerator.getInstance();
```

init()

```
public void init() {
    add(new JButton("Add Light"), SOUTH);
    addActionListeners();
}
```

run()

```
public void run() {
    while (true) {
        for (int i = 0; i < bulbs.size(); i++) {
            bulbs.get(i).setColor(rgen.nextColor());
        }
        pause(PAUSE_TIME);
    }
}
```

```
public void actionPerformed(ActionEvent e) {
      if (e.getActionCommand().equals("Add Light")) {
            GOval bulb = new GOval(2*BULB_H_RADIUS, 2*BULB_V_RADIUS);
            bulb.setFilled(true);
            bulb.setColor(Color.RED);

            GRect base = new GRect(BULB_BASE_SIZE, BULB_BASE_SIZE);
            base.setFilled(true);
            base.setColor(Color.BLACK);

            int x = rgen.nextInt(0, getWidth());
            int y = rgen.nextInt(0, getHeight());
            add(bulb, x – BULB_H_RADIUS, y);
            add(base, x - BULB_BASE_SIZE / 2.0, y + 2*BULB_V_RADIUS);
            bulbs.add(bulb);
      }
}
```

```
public void actionPerformed(ActionEvent e) {
    if (e.getActionCommand().equals("Add Light")) {
        GOval bulb = new GOval(2*BULB_H_RADIUS, 2*BULB_V_RADIUS);
        bulb.setFilled(true);
        bulb.setColor(Color.RED);

        GRect base = new GRect(BULB_BASE_SIZE, BULB_BASE_SIZE);
        base.setFilled(true);
        base.setColor(Color.BLACK);

        int x = rgen.nextInt(0, getWidth());
        int y = rgen.nextInt(0, getHeight());
        add(bulb, x – BULB_H_RADIUS, y);
        add(base, x - BULB_BASE_SIZE / 2.0, y + 2*BULB_V_RADIUS);
        bulbs.add(bulb);
    }
}
```

```
public void actionPerformed(ActionEvent e) {
      if (e.getActionCommand().equals("Add Light")) {
            GOval bulb = new GOval(2*BULB_H_RADIUS, 2*BULB_V_RADIUS);
            bulb.setFilled(true);
            bulb.setColor(Color.RED);

            GRect base = new GRect(BULB_BASE_SIZE, BULB_BASE_SIZE);
            base.setFilled(true);
            base.setColor(Color.BLACK);

            int x = rgen.nextInt(0, getWidth());
            int y = rgen.nextInt(0, getHeight());
            add(bulb, x – BULB_H_RADIUS, y);
            add(base, x - BULB_BASE_SIZE / 2.0, y + 2*BULB_V_RADIUS);
            bulbs.add(bulb);
      }
}
```

```
public void actionPerformed(ActionEvent e) {
      if (e.getActionCommand().equals("Add Light”)) {
            GOval bulb = new GOval(2*BULB_H_RADIUS, 2*BULB_V_RADIUS);
            bulb.setFilled(true);
            bulb.setColor(Color.RED);

            GRect base = new GRect(BULB_BASE_SIZE, BULB_BASE_SIZE);
            base.setFilled(true);
            base.setColor(Color.BLACK);

            int x = rgen.nextInt(0, getWidth());
            int y = rgen.nextInt(0, getHeight());
            add(bulb, x – BULB_H_RADIUS, y);
            add(base, x - BULB_BASE_SIZE / 2.0, y + 2*BULB_V_RADIUS);
            bulbs.add(bulb);
      }
}
```

```
public void actionPerformed(ActionEvent e) {
      if (e.getActionCommand().equals("Add Light”)) {
            GOval bulb = new GOval(2*BULB_H_RADIUS, 2*BULB_V_RADIUS);
            bulb.setFilled(true);
            bulb.setColor(Color.RED);

            GRect base = new GRect(BULB_BASE_SIZE, BULB_BASE_SIZE);
            base.setFilled(true);
            base.setColor(Color.BLACK);

            int x = rgen.nextInt(0, getWidth());
            int y = rgen.nextInt(0, getHeight());
            add(bulb, x – BULB_H_RADIUS, y);
            add(base, x - BULB_BASE_SIZE / 2.0, y + 2*BULB_V_RADIUS);
            bulbs.add(bulb);
      }
}
```

```
public void actionPerformed(ActionEvent e) {
      if (e.getActionCommand().equals("Add Light")) {
            GOval bulb = new GOval(2*BULB_H_RADIUS, 2*BULB_V_RADIUS);
            bulb.setFilled(true);
            bulb.setColor(Color.RED);

            GRect base = new GRect(BULB_BASE_SIZE, BULB_BASE_SIZE);
            base.setFilled(true);
            base.setColor(Color.BLACK);

            int x = rgen.nextInt(0, getWidth());
            int y = rgen.nextInt(0, getHeight());
            add(bulb, x – BULB_H_RADIUS, y);
            add(base, x - BULB_BASE_SIZE / 2.0, y + 2*BULB_V_RADIUS);
            bulbs.add(bulb);
      }
}
```

```
public void actionPerformed(ActionEvent e) {
      if (e.getActionCommand().equals("Add Light”)) {
            GOval bulb = new GOval(2*BULB_H_RADIUS, 2*BULB_V_RADIUS);
            bulb.setFilled(true);
            bulb.setColor(Color.RED);

            GRect base = new GRect(BULB_BASE_SIZE, BULB_BASE_SIZE);
            base.setFilled(true);
            base.setColor(Color.BLACK);

            int x = rgen.nextInt(0, getWidth());
            int y = rgen.nextInt(0, getHeight());
            add(bulb, x – BULB_H_RADIUS, y);
            add(base, x - BULB_BASE_SIZE / 2.0, y + 2*BULB_V_RADIUS);
            bulbs.add(bulb);
      }
}
```

```
public void actionPerformed(ActionEvent e) {
      if (e.getActionCommand().equals("Add Light")) {
            GOval bulb = new GOval(2*BULB_H_RADIUS, 2*BULB_V_RADIUS);
            bulb.setFilled(true);
            bulb.setColor(Color.RED);

            GRect base = new GRect(BULB_BASE_SIZE, BULB_BASE_SIZE);
            base.setFilled(true);
            base.setColor(Color.BLACK);

            int x = rgen.nextInt(0, getWidth());
            int y = rgen.nextInt(0, getHeight());
            add(bulb, x – BULB_H_RADIUS, y);
            add(base, x - BULB_BASE_SIZE / 2.0, y + 2*BULB_V_RADIUS);
            bulbs.add(bulb);
      }
}
```

Strings and chars

Strings and chars

- A **char** is a primitive type that represents a single letter, digit, or symbol. Uses single quotes (‘’).
- Computers represent **chars** as numbers under the hood (ASCII encoding scheme).
- A string is an immutable object that represents a sequence of characters. Uses double quotes (“”).

chars

```
char uppercaseA = 'A';
char uppercaseB = (char)(uppercaseA + 1);

int lettersInAlphabet = 'Z' – 'A' + 1;
// equivalent: 'z' – 'a' + 1
// A to Z and a to z are sequential numbers.
```

Characters

- **Problem:** we can't call methods on chars or store them in collections since they're primitives!
- **Solution:** the Character class
- Call methods on the Character class and pass the char as a *parameter*.

Characters

Useful Methods in the `Character` Class

`static boolean isDigit(char ch)` Determines if the specified character is a digit.
`static boolean isLetter(char ch)` Determines if the specified character is a letter.
`static boolean isLetterOrDigit(char ch)` Determines if the specified character is a letter or a digit.
`static boolean isLowerCase(char ch)` Determines if the specified character is a lowercase letter.
`static boolean isUpperCase(char ch)` Determines if the specified character is an uppercase letter.
`static boolean isWhitespace(char ch)` Determines if the specified character is **whitespace** (spaces and tabs).
`static char toLowerCase(char ch)` Converts **`ch`** to its lowercase equivalent, if any. If not, **`ch`** is returned unchanged.
`static char toUpperCase(char ch)` Converts **`ch`** to its uppercase equivalent, if any. If not, **`ch`** is returned unchanged.

Characters and chars

```
char ch = 'a';
Character.toUpperCase(ch);     // does nothing!
ch.toUpperCase();              // won't compile!
ch = Character.toUpperCase(ch);    // ✔

if (Character.isUpperCase(ch)) {
    println(ch + " is upper case!");
}
```

Strings

- Strings are (immutable) objects. This means we can call methods on them!
- We *cannot change a string after creating it.*
- Strings can be combined with ints, doubles, chars, etc.

Strings are Immutable!

```
String str = “Hello”;
str += “ world”;

// what is actually happening:
String str = “Hello”;
str = str + “ world”;

// This makes a NEW string on the right, and
// then re-assigns str to equal it!
```

Helpful String Methods

Useful Methods in the `String` Class

`int length()` Returns the length of the string
`char charAt(int index)` Returns the character at the specified index. Note: Strings indexed starting at 0.
`String substring(int p1, int p2)` Returns the substring beginning at **`p1`** and extending up to but not including **`p2`**
`String substring(int p1)` Returns substring beginning at **`p1`** and extending through end of string.
`boolean equals(String s2)` Returns true if string **`s2`** is equal to the receiver string. This is case sensitive.
`int compareTo(String s2)` Returns integer whose sign indicates how strings compare in lexicographic order
`int indexOf(char ch)` *or* **`int indexOf(String s)`** Returns index of first occurrence of the character or the string, or -1 if not found
`String toLowerCase()` *or* **`String toUpperCase()`** Returns a lowercase or uppercase version of the receiver string

Using portions of slides by Eric Roberts

Helpful String Methods

```
String str = "Hello world!";
str.toUpperCase();                  // does nothing!
str = str.toUpperCase();            // ✔

for (int i = 0; i < str.length(); i++) {
    println(str.charAt(i));
}
// prints each char on its own line
```

Putting it All Together

```
String str = “happy holidays”;
String newStr = “”;

for (int i = 0; i < str.length(); i++) {
    char ch = str.charAt(i);
    if (i % 2 == 0) ch = Character.toUpperCase(ch);
    newStr += ch;
}
// newStr = HaPpY HoLiDaYs”
```

Type Conversion

```
println("B" + 8 + 4);
// prints "B84"
println("B" + (8 + 4));
// prints "B12"
println('A' + 5 + "ella");
// prints "70ella (note: 'A' corresponds to 65)"
println((char)('A' + 5) + "ella");
// prints "Fella"
```

Type Conversion

- This seems nonsensical - but it's not! (kind of)
- **Just use precedence rules** and keep track of the type along the way.

```
println('A' + 5 + "ella");
// 'A' + 5 is int (70), int + "ella" is string
println((char)('A' + 5) + "ella");
// 'A' + 5 is char ('F'), char + "ella" is string
```

String Tokenizers

- Chop up a string according to certain **delimiters**.
- Use **hasMoreTokens()** to check if there are more tokens.
- Use **nextToken()** to get the next token.
- Optional last parameter in constructor says whether or not to *include delimiters as tokens too*.

```
StringTokenizer t = new StringTokenizer(str, ";");
StringTokenizer t = new StringTokenizer(str, ";", true);
```

Strings Practice

- Lets write a method that, given a string, returns its **acronym**.
  - "In Real Life" -> I.R.L.
  - "All Day I Dream About Soccer " -> A.D.I.D.A.S.
  - "Come Late And Start Sleeping " -> C.L.A.S.S.
  - "Come Late And then you Start Sleeping " -> C.L.A.S.S.
- Every **capitalized word** contributes one letter to the acronym.

Practice: Acronyms

```
private String acronym(String str) {
      String acronym = "";
      StringTokenizer tokenizer = new StringTokenizer(str);

      while (tokenizer.hasMoreTokens()) {
            String token = tokenizer.nextToken();
            if (Character.isUpperCase(token.charAt(0))) {
                  acronym += token.charAt(0) + ".";
            }
      }
      return acronym;
}
```

Practice: Acronyms

```
private String acronym(String str) {
      String acronym = "";
      StringTokenizer tokenizer = new StringTokenizer(str);

      while (tokenizer.hasMoreTokens()) {
            String token = tokenizer.nextToken();
            if (Character.isUpperCase(token.charAt(0))) {
                  acronym += token.charAt(0) + ".";
            }
      }
      return acronym;
}
```

Practice: Acronyms

```
private String acronym(String str) {
      String acronym = "";
      StringTokenizer tokenizer = new StringTokenizer(str);

      while (tokenizer.hasMoreTokens()) {
            String token = tokenizer.nextToken();
            if (Character.isUpperCase(token.charAt(0))) {
                  acronym += token.charAt(0) + ".";
            }
      }
      return acronym;
}
```

Practice: Acronyms

```
private String acronym(String str) {
      String acronym = "";
      StringTokenizer tokenizer = new StringTokenizer(str);

      while (tokenizer.hasMoreTokens()) {
            String token = tokenizer.nextToken();
            if (Character.isUpperCase(token.charAt(0))) {
                  acronym += token.charAt(0) + ".";
            }
      }
      return acronym;
}
```

Practice: Acronyms

```
private String acronym(String str) {
      String acronym = "";
      StringTokenizer tokenizer = new StringTokenizer(str);

      while (tokenizer.hasMoreTokens()) {
            String token = tokenizer.nextToken();
            if (Character.isUpperCase(token.charAt(0))) {
                  acronym += token.charAt(0) + ".";
            }
      }
      return acronym;
}
```

Practice: Acronyms

```
private String acronym(String str) {
      String acronym = "";
      StringTokenizer tokenizer = new StringTokenizer(str);

      while (tokenizer.hasMoreTokens()) {
            String token = tokenizer.nextToken();
            if (Character.isUpperCase(token.charAt(0))) {
                  acronym += token.charAt(0) + ".";
            }
      }
      return acronym;
}
```

Practice: Acronyms

```
private String acronym(String str) {
      String acronym = "";
      StringTokenizer tokenizer = new StringTokenizer(str);

      while (tokenizer.hasMoreTokens()) {
            String token = tokenizer.nextToken();
            if (Character.isUpperCase(token.charAt(0))) {
                  acronym += token.charAt(0) + ".";
            }
      }
      return acronym;
}
```

Practice: Acronyms

```
private String acronym(String str) {
      String acronym = "";
      StringTokenizer tokenizer = new StringTokenizer(str);

      while (tokenizer.hasMoreTokens()) {
            String token = tokenizer.nextToken();
            if (Character.isUpperCase(token.charAt(0))) {
                  acronym += token.charAt(0) + ".";
            }
      }
      return acronym;
}
```

Classes + Interfaces

Classes

- A **class** is a way to tell Java about a new variable type, AKA define a new “class” of object.
- To do that, you must define what information that type of object contains (**instance variable(s)**), what it can do (**method(s)**) and how to create one (**constructor(s)**).
- E.g. NameSurferEntry, HangmanCanvas

Constructors

- Every class has (at least) one **constructor.**
- A **constructor** is the special method called when a new instance of your class is first created.
- You can have multiple constructors that take different arguments. (If you don't write one, Java implicitly provides one that does nothing).

```
GRect rect = new GRect(0, 0, 50, 50);
GRect rect2 = new GRect(50, 50);
```

Interfaces

- An **interface** is a list of method names (no implementations!).
- Any class can **implement** an interface, which means they provide an implementation of every method in the interface.
- Interfaces let different classes tell Java they implement the same behavior. (e.g. GFillable)
- Interfaces let each class implement methods their own way.

Practice: Airplane!

- Let's write a class **Airplane** that implements the **Boardable** interface. **Airplane** is initialized with its capacity. Don't worry about error-checking.

```
public interface Boardable {
    /** Boards a single passenger, at front or back **/
    public void boardPassenger(String name, boolean priority);
    /** Returns whether the vehicle is full **/
    public boolean isFull();
    /** Unboards/returns next passenger **/
    public String unboardPassenger();
}
```

Practice: Airplane!

- Need an ArrayList of passenger names
- Need an int to store the maximum capacity

Practice: Airplane!

```
public class Airplane implements Boardable {
    private ArrayList<String> passengers;
    private int capacity;

    public Airplane(int numSeats) {
        passengers = new ArrayList<String>();
        capacity = numSeats;
    }
    ...
```

boardPassenger

```
public void boardPassenger(String name, boolean priority) {
      if (priority) {
            passengers.add(0, name);
      } else {
            passengers.add(name);
      }
}
...
```

isFull

```
public boolean isFull() {
    return capacity == passengers.size();
}
...
```

unboardPassenger

```
    public String unboardPassenger() {
        return passengers.remove(0);
    }
}
```

Practice: Airplane!

https://youtu.be/wvdNCDVmRPo

Arrays, ArrayLists and HashMaps

Arrays

- An **array** is a fixed-length list of a single type of thing.
- An array can store **primitives** and **objects**.
- You *cannot call methods on it*. I.e. no array.contains().
- Get the length by saying *.length*. (No parentheses!)

ArrayLists

- An **ArrayList** is a flexible-length list of a single type of thing.
- An ArrayList can only store **objects**.
- Uses wrapper classes to store primitives (e.g. Character, Integer, Double).
- An ArrayList has a variety of methods you can use, including *.contains*, *.get*, *.remove*, *.add*, etc.

HashMaps

- A **HashMap** is an unordered bag of key-value pairs.
- A HashMap can only store **objects** as keys and values.
- Uses wrapper classes to store primitives (e.g. Character, Integer, Double).
- A HashMap's keys are unique.
- A HashMap has a variety of methods you can use, including *.containsKey*, *.put*, *.get*, etc.

Which Do I Use?

- Ordered list?
- Any mapping from one thing to another
- Expandable list

Which Do I Use?

- Ordered list? -> array or ArrayList
- Any mapping from one thing to another -> HashMap
- Expandable list ArrayList

Iterators

- An iterator is an object that lets you iterate over every value in a collection (e.g. ArrayList, HashMap).
- Similar to StringTokenizer: hasNext(), next()
- Lets you iterate over a list without having access to the list itself.

Iterators - ArrayList

```
Iterator<String> it = myList.iterator();
while (it.hasNext()) {
    println(it.next());
}

// Note: in order
```

Iterators - HashMap

```
Iterator<String> it = myMap.keySet().iterator();
while (it.hasNext()) {
     String key = it.next();
     println(key + " -> " + myMap.get(key));
}

// Note: unknown order!
```

Collections Practice

- Let's write a console program to keep track of what gifts you're getting your friends.
- The program should allow a user to enter a name, followed by one gift to get for them. You may only add one gift at a time, but may add a gift for a person multiple times.
- Once the user is done, you should print out each person followed by what gifts to get them.

Practice: GiftList

```
public void run() {
    HashMap<String, ArrayList<String>> giftMap =
        new HashMap<String, ArrayList<String>>();
    println("Welcome to GiftList!  Enter a name,");
    println("followed by a gift.  Just ENTER to quit.");

    ...
```

Practice: GiftList

```
while (true) {
        String name = readLine("Name? ");
        if (name.equals("")) break;
        String gift = readLine("Gift? ");

        ArrayList<String> giftList;
        if (giftMap.containsKey(name)) {
            giftList = giftMap.get(name);
        } else giftList = new ArrayList<String>();
...
```

Practice: GiftList

```
while (true) {
        String name = readLine("Name? ");
        if (name.equals("")) break;
        String gift = readLine("Gift? ");

        ArrayList<String> giftList;
        if (giftMap.containsKey(name)) {
            giftList = giftMap.get(name);
        } else giftList = new ArrayList<String>();
...
```

Practice: GiftList

```
while (true) {
    String name = readLine("Name? ");
    if (name.equals("")) break;
    String gift = readLine("Gift? ");

    ArrayList<String> giftList;
    if (giftMap.containsKey(name)) {
        giftList = giftMap.get(name);
    } else giftList = new ArrayList<String>();
...
```

Practice: GiftList

```
while (true) {
    String name = readLine("Name? ");
    if (name.equals("")) break;
    String gift = readLine("Gift? ");

    ArrayList<String> giftList;
    if (giftMap.containsKey(name)) {
        giftList = giftMap.get(name);
    } else giftList = new ArrayList<String>();
...
```

Practice: GiftList

```
            giftList.add(gift);
            giftMap.put(name, giftList);
        }
        println(giftMap.toString());
}
```

File Reading

File Reading

- Use a **BufferedReader** to read in a text file.
- try/catch – “try this, and catch the error if something happens”
- BufferedReaders are like **StringTokenizers**; they can only give you the *next line* in the file.
- Remember to close your BufferedReader when you’re done!

File Reading

```
try {
      BufferedReader rd =
            new BufferedReader(new FileReader(filename));
      while (true) {
            String line = rd.readLine();
            if (line == null) break;
            println("Read line: [" + line + "]");
      }
      rd.close();
} catch (IOException ex) {
      throw new ErrorException(ex);
}
```

Practice: GuestList

- Let's say we're given a guest list for a party. The guest list is formatted as follows:

```
1 Nick - 2
2 Hannah - 3
3 Isaac - 5
4 Austin - 5
5 George - 6
```

- Specifically, each line has the name of a friend, and how many people *they* are bringing. Print out the friend bringing the most people.

```
String maxName = "";
int maxGuests = 0;
try {
      BufferedReader rd = new BufferedReader(new
            FileReader("guestList.txt"));
      while (true) {
            String line = rd.readLine();
            if (line == null) break;
            StringTokenizer t = new StringTokenizer(line, "—");
            String name = t.nextToken().trim();
            int numGuests = Integer.parseInt(t.nextToken().trim());
            if (numGuests > maxGuests) {
                  maxGuests = numGuests;
                  maxName = name;
            }
      }...
```

```
String maxName = "";
int maxGuests = 0;
try {
    BufferedReader rd = new BufferedReader(new
        FileReader("guestList.txt"));
    while (true) {
        String line = rd.readLine();
        if (line == null) break;
        StringTokenizer t = new StringTokenizer(line, "-");
        String name = t.nextToken().trim();
        int numGuests = Integer.parseInt(t.nextToken().trim());
        if (numGuests > maxGuests) {
            maxGuests = numGuests;
            maxName = name;
        }
    }...
```

```
String maxName = “”;
int maxGuests = 0;
try {
     BufferedReader rd = new BufferedReader(new
          FileReader(“guestList.txt”));
     while (true) {
          String line = rd.readLine();
          if (line == null) break;
          StringTokenizer t = new StringTokenizer(line, “–“);
          String name = t.nextToken().trim();
          int numGuests = Integer.parseInt(t.nextToken().trim());
          if (numGuests > maxGuests) {
               maxGuests = numGuests;
               maxName = name;
          }
     }...
```

```
String maxName = "";
int maxGuests = 0;
try {
    BufferedReader rd = new BufferedReader(new
        FileReader("guestList.txt"));
    while (true) {
        String line = rd.readLine();
        if (line == null) break;
        StringTokenizer t = new StringTokenizer(line, "—");
        String name = t.nextToken().trim();
        int numGuests = Integer.parseInt(t.nextToken().trim());
        if (numGuests > maxGuests) {
            maxGuests = numGuests;
            maxName = name;
        }
    }...
```

```
String maxName = "";
int maxGuests = 0;
try {
    BufferedReader rd = new BufferedReader(new
        FileReader("guestList.txt"));
    while (true) {
        String line = rd.readLine();
        if (line == null) break;
        StringTokenizer t = new StringTokenizer(line, "—");
        String name = t.nextToken().trim();
        int numGuests = Integer.parseInt(t.nextToken().trim());
        if (numGuests > maxGuests) {
            maxGuests = numGuests;
            maxName = name;
        }
    }...
```

```
String maxName = "";
int maxGuests = 0;
try {
     BufferedReader rd = new BufferedReader(new
          FileReader("guestList.txt"));
     while (true) {
          String line = rd.readLine();
          if (line == null) break;
          StringTokenizer t = new StringTokenizer(line, "—");
          String name = t.nextToken().trim();
          int numGuests = Integer.parseInt(t.nextToken().trim());
          if (numGuests > maxGuests) {
               maxGuests = numGuests;
               maxName = name;
          }
     }...
```

```
String maxName = “”;
int maxGuests = 0;
try {
     BufferedReader rd = new BufferedReader(new
          FileReader(“guestList.txt”));
     while (true) {
          String line = rd.readLine();
          if (line == null) break;
          StringTokenizer t = new StringTokenizer(line, “–“);
          String name = t.nextToken().trim();
          int numGuests = Integer.parseInt(t.nextToken().trim());
          if (numGuests > maxGuests) {
               maxGuests = numGuests;
               maxName = name;
          }
     }...
```

```
        rd.close();
        println(maxName + " is bringing the most guests.");
    } catch (IOException ex) {
        throw new ErrorException(ex);
    }
```

Wrap-up

Wrap-up

- Try to get to every problem
- Don’t rush to coding too quickly
- Pseudocode!
- Look over practice final, extra problems, section handouts, and book for more practice
- Think about how much you’ve learned in 10 weeks!