Final Review Session

Brahm Capoor, Fall 2018

Logistics

December 10th, 8:30 - 11:30 AM

Last names A-L: Hewlett 200 (where we have lecture)

Last names M-Q: Hewlett 201 (next to where we have lecture)

Last names R-Z: Bishop Auditorium

Come a little early!

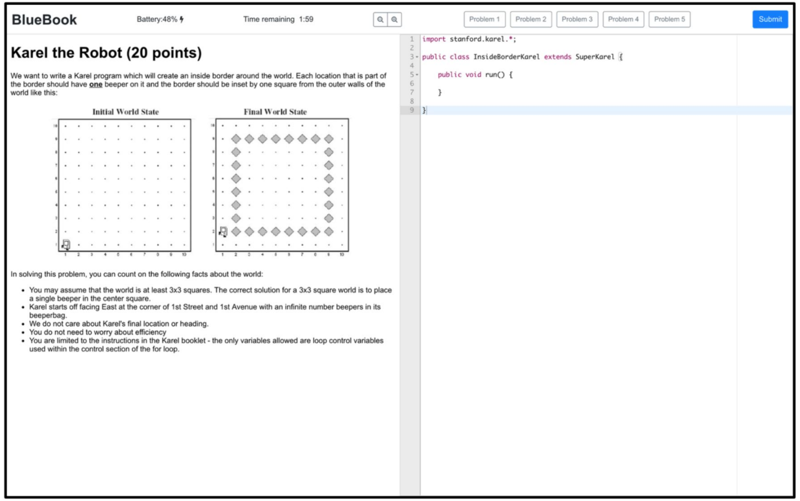
BlueBook

Download for Mac here

Download for Windows here

Handout here

Make sure to have it installed and set up before the exam



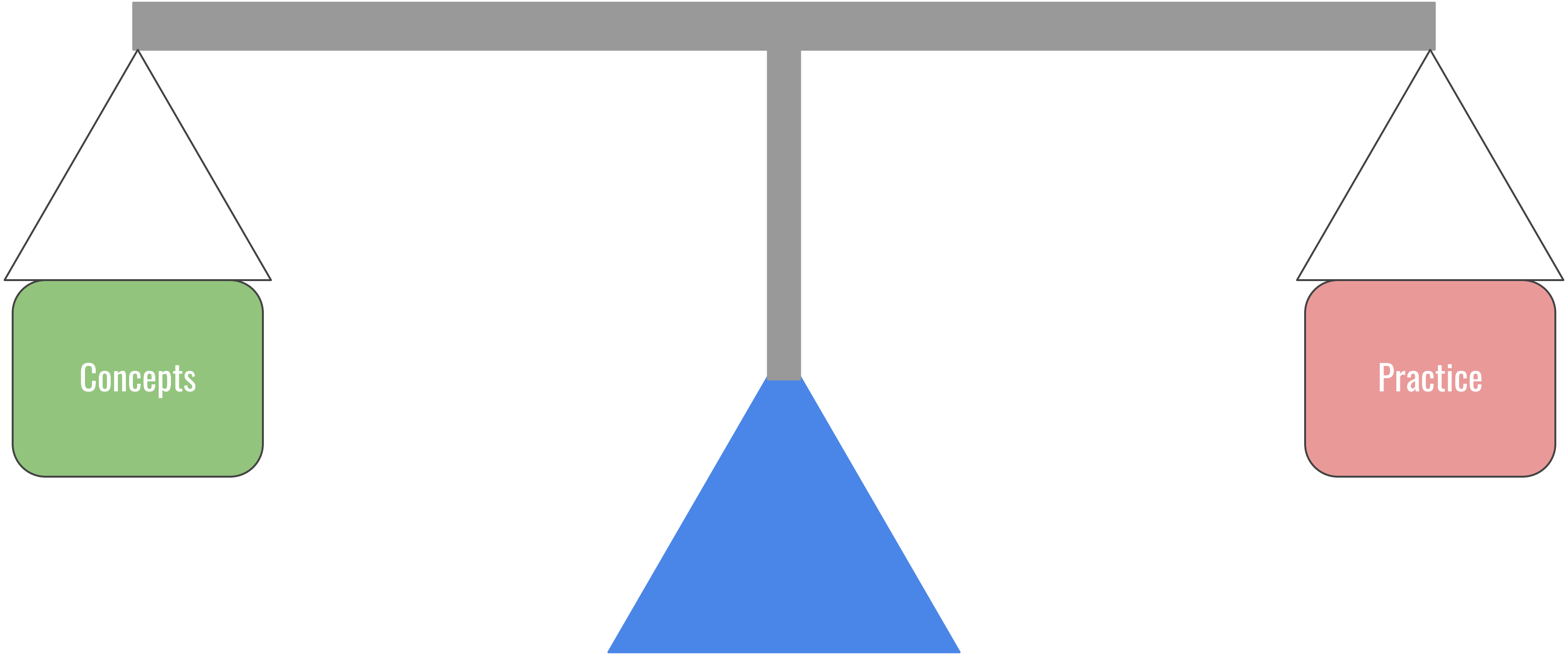
Karel the Robot (20 points)

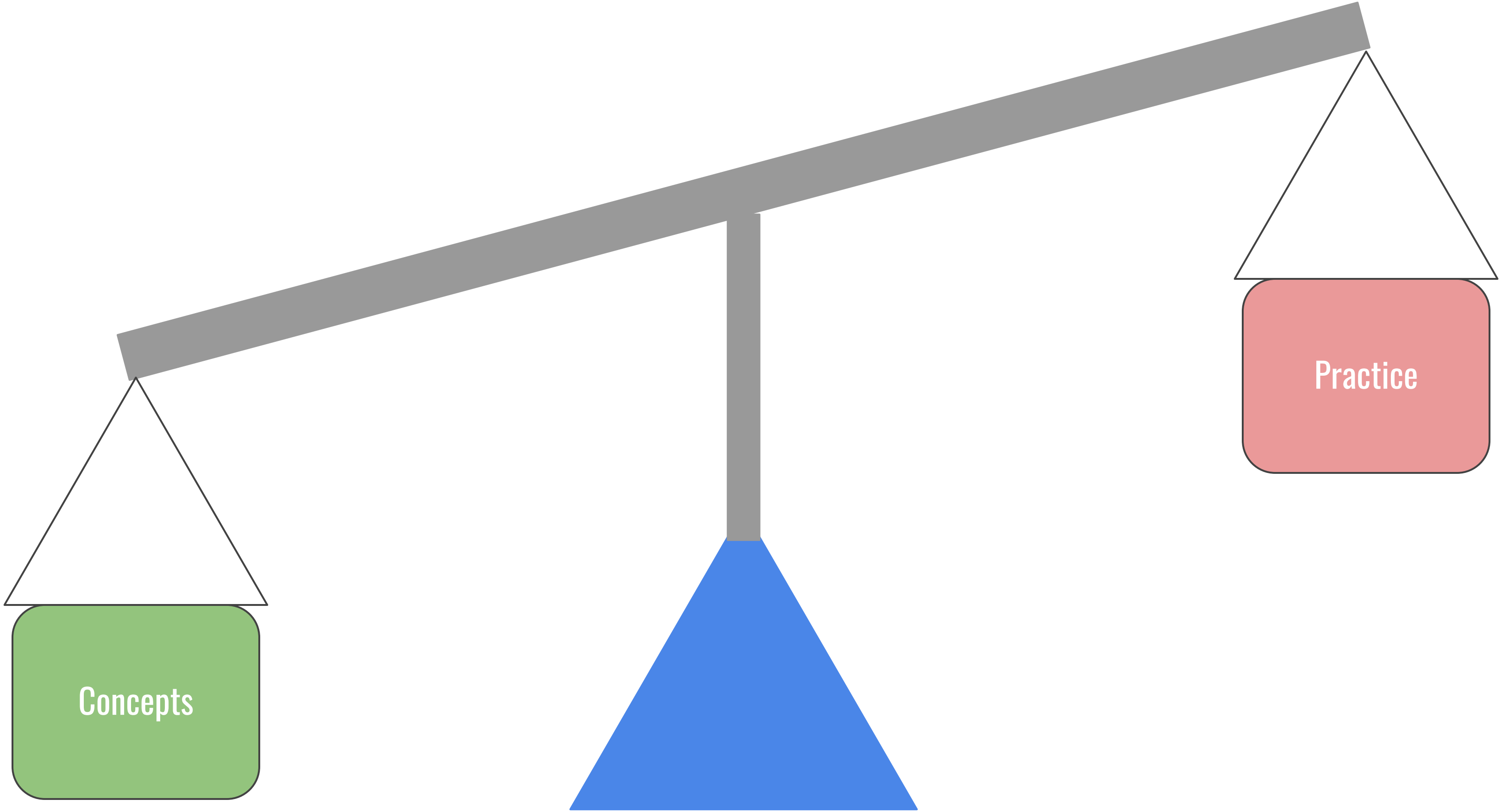
We want to write a Karel program which will create an inside border around the world. Each location that is part of the border should have **one** beeper on it and the border should be inset by one square from the outer walls of the world like this:

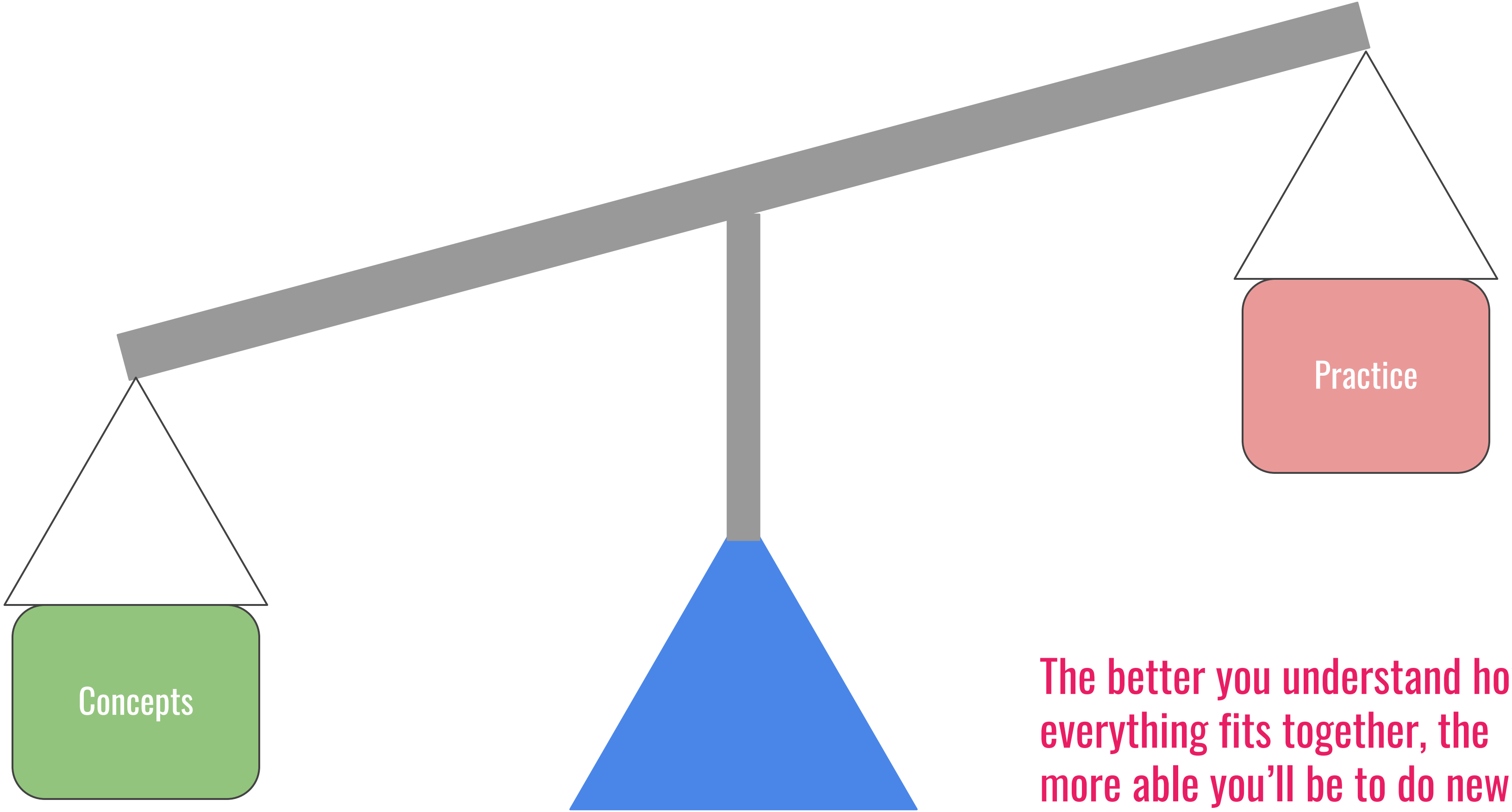
In solving this problem, you can count on the following facts about the world:

- You may assume that the world is at least 3x3 squares. The correct solution for a 3x3 square world is to place a single beeper in the center square.
- Karel starts off facing East at the corner of 1st Street and 1st Avenue with an infinite number beepers in its beeperbag.
- We do not care about Karel's final location or heading.
- You do not need to worry about efficiency
- You are limited to the instructions in the Karel booklet - the only variables allowed are loop control variables used within the control section of the for loop.

```
import stanford.karel.*;

public class InsideBorderKarel extends SuperKarel {

    public void run() {

    }

}
```


Concepts
Practice


Concepts
Practice



The better you understand how everything fits together, the more able you'll be to do new problems!

Where to find practice problems

Section handouts

Practice Final + Additional Practice Problems

CodeStepByStep

Textbook

Scattered throughout these slides

Any logistical questions?

Midterm Greatest Hits

Check out the midterm review for the full collection

Skip to the next section of these slides

Primitive variables

```
int x = 7;   // declare and initialize a variable
x = 9;       // change the value of x
x = x + 1;   // increment (add 1 to) x.  A.K.A. x++
x = x + 2;   // add 2 to x.              A.K.A. x += 2
x /= 2;      // divide x by 2, and truncate result

double d = 3.5;

boolean isThisTrue = true;
isThisTrue = !isThisTrue;  // flip isThisTrue
```

Class variables

```
Type thing = new Type();                   // construct an object
type_1 x = thing.getSomething();           // call a getter method
thing.setSomething(someValue);             // call a setter method
thing.doSomething(argument1, argument2);   // call another method


GRect rect = new GRect(42, 42, 100, 100);
double x = rect.getX();
thing.setLocation(19, 97);
thing.move(20, 25);
```

Class variable types start with capital letters and Primitive variable types start with lowercase letters

Methods

```
private returnType methodName(type param1, type param2, …) {
    // sick code here
}
```

- A method header provides some guarantees about the method (what it returns, how many parameters it takes)
- Parameters and return values generalize the methods we saw in Karel to allow the use of variables
- If a method returns something, that something needs to be stored in a variable

```
returnType storedValue = methodName(/* params */);
```

Primitive variables passed into a method are passed by value

Graphics

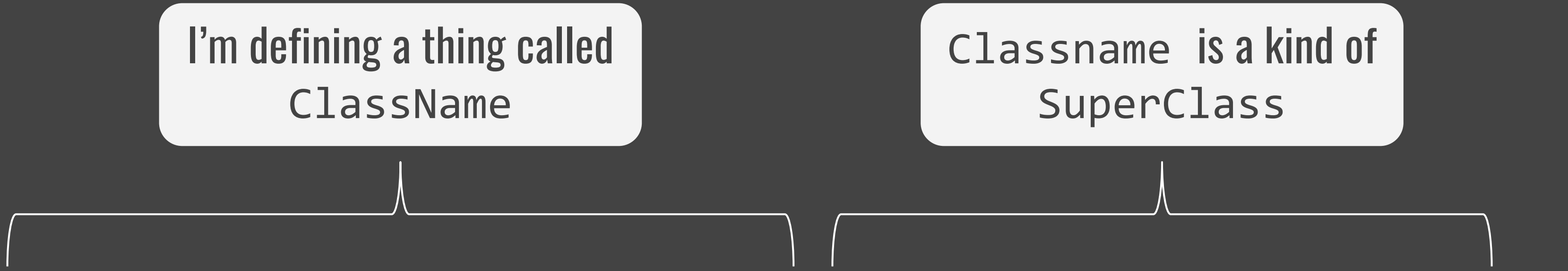
```
GRect rect = new GRect(50, 50, 200, 200);
rect.setFilled(true);
rect.setColor(Color.BLUE);

GOval oval = new GOval(0, 0, getWidth(), getHeight());
oval.setFilled(false);
oval.setColor(Color.GREEN);

GLabel text = new GLabel(“banter”, 200, 10);

add(text);
add(rect);
add(oval);
```

Things to remember

- Coordinates are `double`s
- Coordinates are measured from the top left of the screen
- Coordinates of a shape are coordinates of its top left corner
- Coordinates of a label are coordinates of its bottom left corner
- Remember to `add` objects to the screen!
- Use the online documentation!

I’m defining a thing called `ClassName`

```
public class <ClassName> {

   // sick code here

}
```



I’m defining a thing called ClassName

Classname is a kind of SuperClass

```
public class <ClassName> extends <SuperClass> {

   // sick code here

}
```

Student.java

```java
public class Student {

    // sick code here

}
```

Stanford.java

```java
public void run() {
    Student s1;
    Student s2;
    Student s3;
    // more sick code here
}
```

Creating objects of type `Student`

Instance variables

Defined as part of a class, but not within any particular method

```java
public class Student {

    private String studentName;
    private int studentId;
    private String email;
    private int numUnits;
    private boolean isInternational;
}
```

s1, s2 and s3 all have their own independent properties

```java
public void run() {

    Student s1;
    Student s2;
    Student s3;

}
```

Initializing your instance variables in the constructor

```
public class Student {

    public Student(String name, int id, String email,
                  int numUnits, boolean isInternational) {
        studentName = name;
        studentId = id;
        this.email = email; // to disambiguate between variables
        this.numUnits = numUnits;
        this.isInternational = isInternational;
    }

    /* instance variables go down here */
}
```

Getters and Setters: some notes

```java
public class Student {

    public Student(int unitCount) {
        numUnits = unitCount;
    }

    public int getUnits() {
        return numUnits;
    }

    public void setUnits(int newUnits) {
        numUnits = newUnits;
    }

    private int numUnits;

}
```

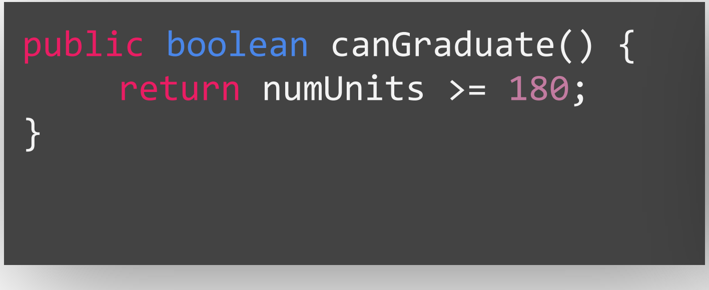
Getter and Setter methods are public (exported) so we can call them in other classes and programs

Define Getters and Setters whenever you want to grant a client access to or control over an instance variable

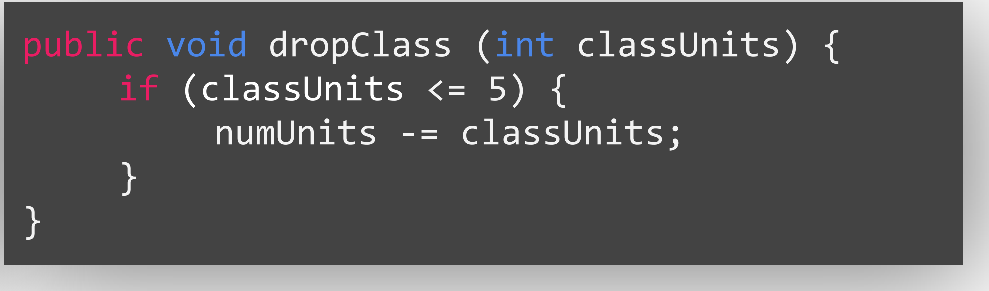
These methods are typically very short

They allow more precise control over the value of a variable:

```java
public void setUnits(int newUnits) {
    if (newUnits >= numUnits) {
        numUnits = newUnits;
    }
}
```



```
public boolean canGraduate() {
    return numUnits >= 180;
}
```



```
public void dropClass (int classUnits) {
    if (classUnits <= 5) {
        numUnits -= classUnits;
    }
}
```

Methods allow us to define **behaviours** for our classes

File Processing

```
try {
     BufferedReader rd = new BufferedReader(new FileReader (filename));
     while (true) {
          String line = rd.nextLine();
          if (line == null) break;
          println(“Just read: ” + line);
     }
     rd.close();
} catch (IOException ex) {
     throw new ErrorException(ex):
}
```

```
try {
    BufferedReader rd = new BufferedReader(new FileReader (filename));
    while (true) {
        String line = rd.nextLine();
        if (line == null) break;
        println(“Just read: ” + line);
    }
    rd.close();
} catch (IOException ex) {
    throw new ErrorException(ex):
}
```

Can only give you the next line in a file

```
try {
     BufferedReader rd = new BufferedReader(new FileReader (filename));
     while (true) {
          String line = rd.nextLine();
          if (line == null) break;
          println(“Just read: ” + line);
     }
     rd.close();
} catch (IOException ex) {
     throw new ErrorException(ex):
}
```

Denotes the end of the file, so we end the loop

```
try {
     BufferedReader rd = new BufferedReader(new FileReader (filename));
     while (true) {
          String line = rd.nextLine();
          if (line == null) break;
          println(“Just read: ” + line);
     }
     rd.close();
} catch (IOException ex) {
     throw new ErrorException(ex):
}
```

Try living dangerously

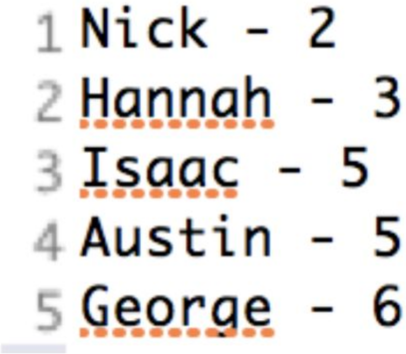
Life insurance

```
public void printFile() {
    try {
        BufferedReader rd = new BufferedReader(new FileReader (filename));
        while (true) {
            String line = rd.nextLine();
            if (line == null) break;
            println(line);
        }
        rd.close();
    } catch (IOException ex) {
        throw new ErrorException(ex):
    }
}
```

A practice problem, courtesy of Nick Troccoli

Skip to next section

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



```
Nick - 2
Hannah - 3
Isaac - 5
Austin - 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.

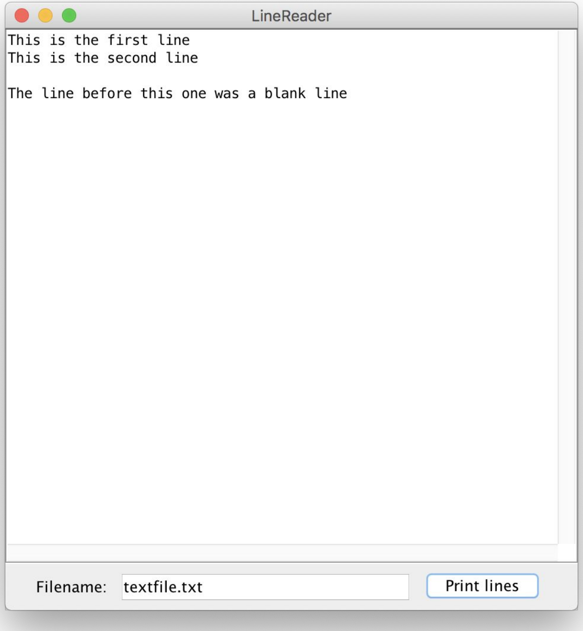
```
String maxName = “”;
int maxGuests = 0;
try {
     BufferedReader rd = new BufferedReader(new
          FileReader(“guestList.txt”));
     while (true) {
          String line = rd.readLine();
          if (line == null) break;
          StringTokenizer t = new StringTokenizer(line, “–“);
      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

Interactors

A problem

Write a program that allows a user to type in a filename in a text field and then upon pressing a button print every line of the file.

- You can assume the file exists
- The file may be any number of lines long
- You may not use any data structures



```
This is the first line
This is the second line

The line before this one was a blank line
```

Filename: textfile.txt

```java
public void init() {
      JLabel label = new JLabel("Filename: ");
      add(label, SOUTH);








}
```

First, add the interactors in init()

```java
private JTextField tf;


public void init() {
      JLabel label = new JLabel("Filename: ");
      add(label, SOUTH);

      tf = new JTextField(20);









}
```

JTextFields are always instance variables

```java
private JTextField tf;

public void init() {
      JLabel label = new JLabel("Filename: ");
      add(label, SOUTH);

      tf = new JTextField(20);
      tf.setActionCommand("Set File");
      tf.addActionListener(this);
      add(tf, SOUTH);






}
```

We always set the action command and add action listeners to text fields

```
private JTextField tf;


public void init() {
      JLabel label = new JLabel("Filename: ");
      add(label, SOUTH);

      tf = new JTextField(20);
      tf.setActionCommand("Set File");
      tf.addActionListener(this);
      add(tf, SOUTH);

      JButton button = new JButton("Print lines");
      add(button, SOUTH);



}
```

Interactors get added to the screen in the order that we define them

```java
private JTextField tf;


public void init() {
    JLabel label = new JLabel("Filename: ");
    add(label, SOUTH);

    tf = new JTextField(20);
    tf.setActionCommand("Set File");
    tf.addActionListener(this);
    add(tf, SOUTH);

    JButton button = new JButton("Print lines");
    add(button, SOUTH);

    addActionListeners();

}
```

Remember to add `ActionListeners` to your program!

```
private JTextField tf;


public void init() {
     JLabel label = new JLabel("Filename: ");
     add(label, SOUTH);

     tf = new JTextField(20);
     tf.setActionCommand("Set File");
     tf.addActionListener(this);
     add(tf, SOUTH);

     JButton button = new JButton("Print lines");
     add(button, SOUTH);

     addActionListeners();
}
```

```
public void actionPerformed(ActionEvent e) {
     String cmd = e.getActionCommand();







}
```

All programs with Action Listeners need an `actionPerformed` method

```
private JTextField tf;
private String filename;

public void init() {
      JLabel label = new JLabel("Filename: ");
      add(label, SOUTH);

      tf = new JTextField(20);
      tf.setActionCommand("Set File");
      tf.addActionListener(this);
      add(tf, SOUTH);

      JButton button = new JButton("Print lines");
      add(button, SOUTH);

      addActionListeners();
}
```

```
public void actionPerformed(ActionEvent e) {
      String cmd = e.getActionCommand();
      if (cmd.equals("Set File")) {
            filename = tf.getText();
      }


}
```

We go through each of the possible action commands

```java
private JTextField tf;
private String filename;

public void init() {
    JLabel label = new JLabel("Filename: ");
    add(label, SOUTH);

    tf = new JTextField(20);
    tf.setActionCommand("Set File");
    tf.addActionListener(this);
    add(tf, SOUTH);

    JButton button = new JButton("Print lines");
    add(button, SOUTH);

    addActionListeners();
}
```

```java
public void actionPerformed(ActionEvent e) {
    String cmd = e.getActionCommand();
    if (cmd.equals("Set File")) {
        filename = tf.getText();
    }
    if (cmd.equals("Print lines")) {
        printFile()
    }
}
```

We call the `printFile` method defined in the last section

```java
private JTextField tf;
private String filename;

public void init() {
    JLabel label = new JLabel("Filename: ");
    add(label, SOUTH);

    tf = new JTextField(20);
    tf.setActionCommand("Set File");
    tf.addActionListener(this);
    add(tf, SOUTH);

    JButton button = new JButton("Print lines");
    add(button, SOUTH);

    addActionListeners();

}
```

```java
public void actionPerformed(ActionEvent e) {
    String cmd = e.getActionCommand();
    if (cmd.equals("Set File")) {
        filename = tf.getText();
    }
    if (cmd.equals("Print lines")) {
        printFile()
    }
}
```

Data structures: `ArrayLists`, `HashMaps` and arrays

Arrays

Fixed size

Store objects or primitives

No methods, only `.length`

Ordered

Arrays	`ArrayLists`
Fixed size	Variable size
Store objects or primitives	Store only objects
No methods, only `.length`	Methods
Ordered	Ordered

Arrays	`ArrayLists`	`HashMaps`
Fixed size	Variable size	Variable size
Store objects or primitives	Store only objects	Store only objects
No methods, only `.length`	Methods	Methods
Ordered	Ordered	Key-Value Associations

Arrays	ArrayLists	HashMaps
Fixed size	Variable size	Variable size
Store objects or primitives	Store only objects	Store only objects
No methods, only `.length`	Methods	Methods
Ordered	Ordered	Key-Value Associations

Wrapper classes

`int`	`Integer`
`double`	`Double`
`boolean`	`Boolean`
`char`	`Character`

Use these instead

A problem:

Suppose we have a bunch of Stanford Students who want to go to a Masquerade Ball, and a bunch of carriages of variable size that can take them there. How can we assign the students to these carriages?

```
ArrayList<String> students = // {“Brahm”, “Kate”, “Zach”, “Jade”, “Vasco”, “Olivia”}
ArrayList<Integer> capacities = {1, 3, 2}
printAssignments(students, capacities);

outputs:
Brahm is in carriage 0, which has Brahm
Kate is in carriage 1, which has Kate, Zach, Jade
Zach is in carriage 1, which has Kate, Zach, Jade
Jade is in carriage 1, which has Kate, Zach, Jade
Vasco is in carriage 2, which has Vasco, Olivia
Olivia is in carriage 2, which has Vasco, Olivia
```

A problem: The Stanford Carriage Pact

Suppose we have a bunch of Stanford Students who want to go to a Masquerade Ball, and a bunch of carriages of variable size that can take them there. How can we assign the students to these carriages?

```
ArrayList<String> students = // {“Brahm”, “Kate”, “Zach”, “Jade”, “Vasco”, “Olivia”}
ArrayList<Integer> capacities = {1, 3, 2}
printAssignments(students, capacities);

outputs:
Brahm is in carriage 0, which has Brahm
Kate is in carriage 1, which has Kate, Zach, Jade
Zach is in carriage 1, which has Kate, Zach, Jade
Jade is in carriage 1, which has Kate, Zach, Jade
Vasco is in carriage 2, which has Vasco, Olivia
Olivia is in carriage 2, which has Vasco, Olivia
```

The Stanford Carriage Pact

Questions I would ask myself about this problem

What information do I need to store?

Questions I would ask myself about this problem

What information do I need to store?

Which carriage each student is in, and which students are in each carriage

Questions I would ask myself about this problem

What information do I need to store?

Which carriage each student is in, and which students are in each carriage

What types are these relationships between?

Questions I would ask myself about this problem

What information do I need to store?

Which carriage each student is in, and which students are in each carriage

What types are these relationships between?

String => int, and int => List of students

Questions I would ask myself about this problem

What information do I need to store?

Which carriage each student is in, and which students are in each carriage

What types are these relationships between?

String => int, and int => List of students

What data structures are best for these relationships?

Questions I would ask myself about this problem

What information do I need to store?

Which carriage each student is in, and which students are in each carriage

What types are these relationships between?

String => int, and int => List of students

What data structures are best for these relationships?

`HashMap<String, Integer>` *and* `ArrayList<ArrayList<String>>`

```
private void printAssignments(ArrayList<String> students, ArrayList<Integer> capacities) {
      HashMap<String, Integer> studentsToCarriages = new HashMap<String, Integer>();
      ArrayList<ArrayList<String>> carriages = new ArrayList<ArrayList<String>>();













}
```

Start by making those data structures

```
private void printAssignments(ArrayList<String> students, ArrayList<Integer> capacities) {
     HashMap<String, Integer> studentsToCarriages = new HashMap<String, Integer>();
     ArrayList<ArrayList<String>> carriages = new ArrayList<ArrayList<String>>();


     int currCarriageIdx = 0;

     for (int i = 0; i < students.size(); i++) {
          String currStudent = students.get(i);
          studentsToCarriages.put(currStudent, currCarriageIdx);






     }







}
```

Optimize for what's easy - let's assume that `currCarriageIdx` is always correct

```
private void printAssignments(ArrayList<String> students, ArrayList<Integer> capacities) {
    HashMap<String, Integer> studentsToCarriages = new HashMap<String, Integer>();
    ArrayList<ArrayList<String>> carriages = new ArrayList<ArrayList<String>>();


    int currCarriageIdx = 0;

    for (int i = 0; i < students.size(); i++) {
        String currStudent = students.get(i);
        studentsToCarriages.put(currStudent, currCarriageIdx);

        if (/* current carriage size */ == capacities.get(currCarriageIdx)) {
            // add current carriage to carriages list
            // make a new current carriage
            currCarriageIdx++;
        }
    }



}
```

Make sure that `currCarriageIdx` is always correct

```java
private void printAssignments(ArrayList<String> students, ArrayList<Integer> capacities) {
    HashMap<String, Integer> studentsToCarriages = new HashMap<String, Integer>();
    ArrayList<ArrayList<String>> carriages = new ArrayList<ArrayList<String>>();

    ArrayList<String> currentCarriage = new ArrayList<String>();
    int currCarriageIdx = 0;

    for (int i = 0; i < students.size(); i++) {
        String currStudent = students.get(i);
        studentsToCarriages.put(currStudent, currCarriageIdx);
        currentCarriage.add(currStudent);
        if (currentCarriage.size() == capacities.get(currCarriageIdx)) {
            carriages.add(currentCarriage);
            // make a new current carriage
            currCarriageIdx++;
        }
    }



}
```

Use an `ArrayList` to represent the currentCarriage

```java
private void printAssignments(ArrayList<String> students, ArrayList<Integer> capacities) {
    HashMap<String, Integer> studentsToCarriages = new HashMap<String, Integer>();
    ArrayList<ArrayList<String>> carriages = new ArrayList<ArrayList<String>>();

    ArrayList<String> currentCarriage = new ArrayList<String>();
    int currCarriageIdx = 0;

    for (int i = 0; i < students.size(); i++) {
        String currStudent = students.get(i);
        studentsToCarriages.put(currStudent, currCarriageIdx);
        currentCarriage.add(currStudent);
        if (currentCarriage.size() == capacities.get(currCarriageIdx)) {
            carriages.add(currentCarriage);
            currentCarriage = new ArrayList<String>();
            currCarriageIdx++;
        }
    }



}
```

Use an `ArrayList` to represent the currentCarriage

```
private void printAssignments(ArrayList<String> students, ArrayList<Integer> capacities) {
    HashMap<String, Integer> studentsToCarriages = new HashMap<String, Integer>();
    ArrayList<ArrayList<String>> carriages = new ArrayList<ArrayList<String>>();

    ArrayList<String> currentCarriage = new ArrayList<String>();
    int currCarriageIdx = 0;

    for (int i = 0; i < students.size(); i++) {
        String currStudent = students.get(i);
        studentsToCarriages.put(currStudent, currCarriageIdx);
        currentCarriage.add(currStudent);
        if (currentCarriage.size() == capacities.get(currCarriageIdx)) {
            carriages.add(currentCarriage);
            currentCarriage = new ArrayList<String>();
            currCarriageIdx++;
        }
    }

    for (int i = 0; i < students.size(); i++) {
        String currStudent = students.get(i);
        int carriage = studentsToCarriages.get(currStudent);
        ArrayList<String> studentsInCarriage = carriages.get(carriage);
        println(currStudent + carriage + studentsInCarriage);
    }
}
```

Output!

```java
private void printAssignments(ArrayList<String> students, ArrayList<Integer> capacities) {
    HashMap<String, Integer> studentsToCarriages = new HashMap<String, Integer>();
    ArrayList<ArrayList<String>> carriages = new ArrayList<ArrayList<String>>();

    ArrayList<String> currentCarriage = new ArrayList<String>();
    int currCarriageIdx = 0;

    for (int i = 0; i < students.size(); i++) {
        String currStudent = students.get(i);
        studentsToCarriages.put(currStudent, currCarriageIdx);
        currentCarriage.add(currStudent);
        if (currentCarriage.size() == capacities.get(currCarriageIdx)) {
            carriages.add(currentCarriage);
            currentCarriage = new ArrayList<String>();
            currCarriageIdx++;
        }
    }

    for (int i = 0; i < students.size(); i++) {
        String currStudent = students.get(i);
        int carriage = studentsToCarriages.get(currStudent);
        ArrayList<String> studentsInCarriage = carriages.get(carriage);
        println(currStudent + carriage + studentsInCarriage);
    }
}
```

Iterators

The key insight

Any collection supports some notion of iteration over its elements

The key insight

Any collection supports some notion of iteration over its elements

There are two important pieces of information when you’re iterating

- Which element you’re currently at

- What the next element is

The key insight

Any collection supports some notion of iteration over its elements

There are two important pieces of information when you’re iterating

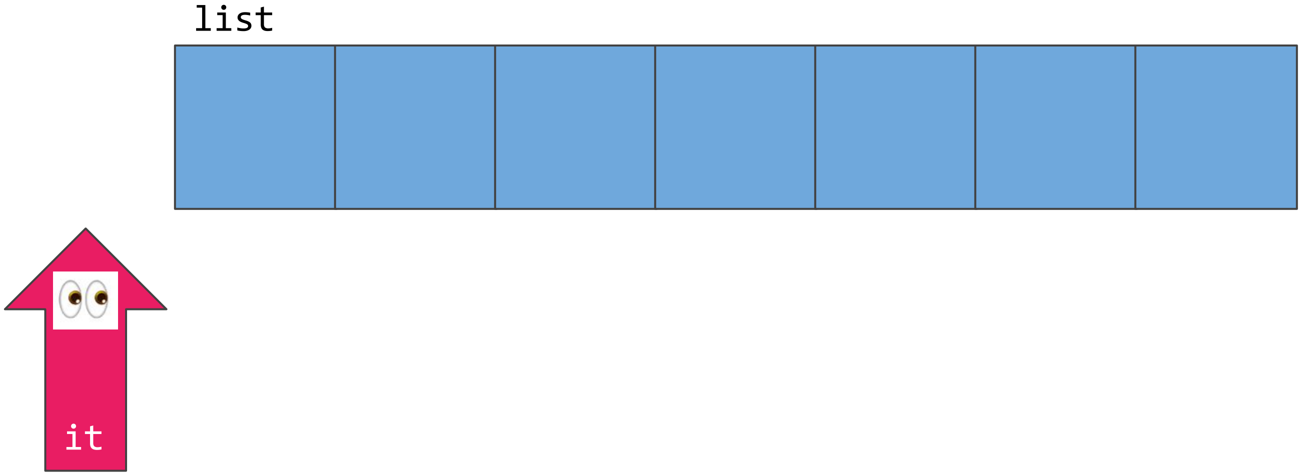
- Which element you’re currently at

- What the next element is

An iterator answers both those questions

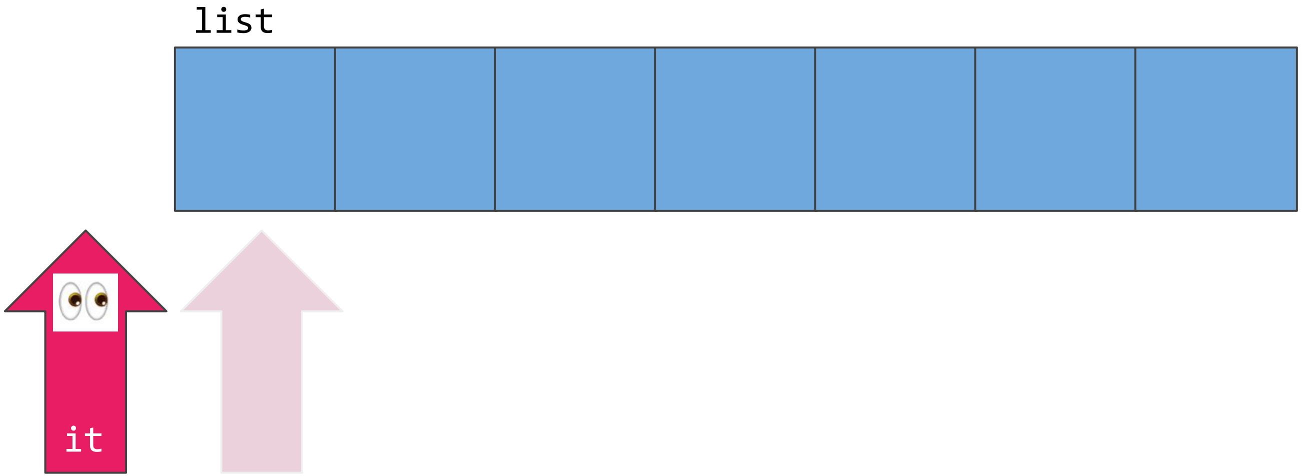
An iterator is an arrow...

```
ArrayList<T> list = new ArrayList<T>();
Iterator<T> it = list.iterator();
```



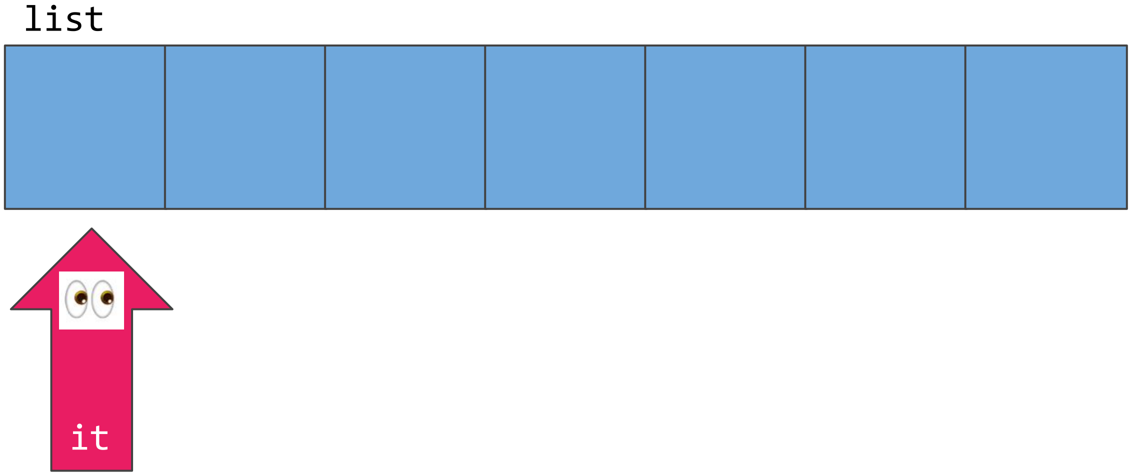
…that can check whether it can move forward…

```
while (it.hasNext()) {
```



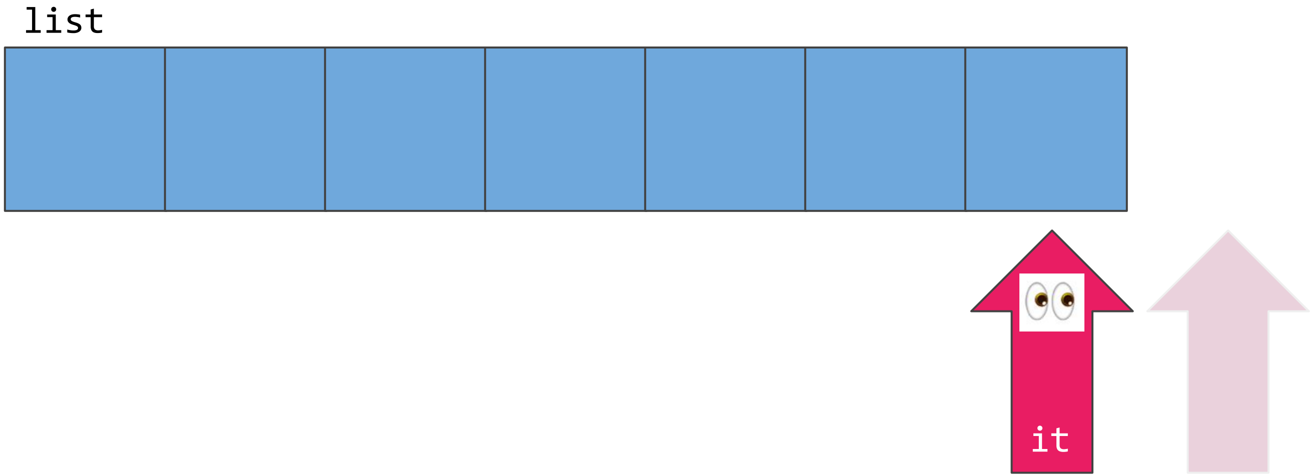
...and then move there.

```
T nextElem = it.next();
```



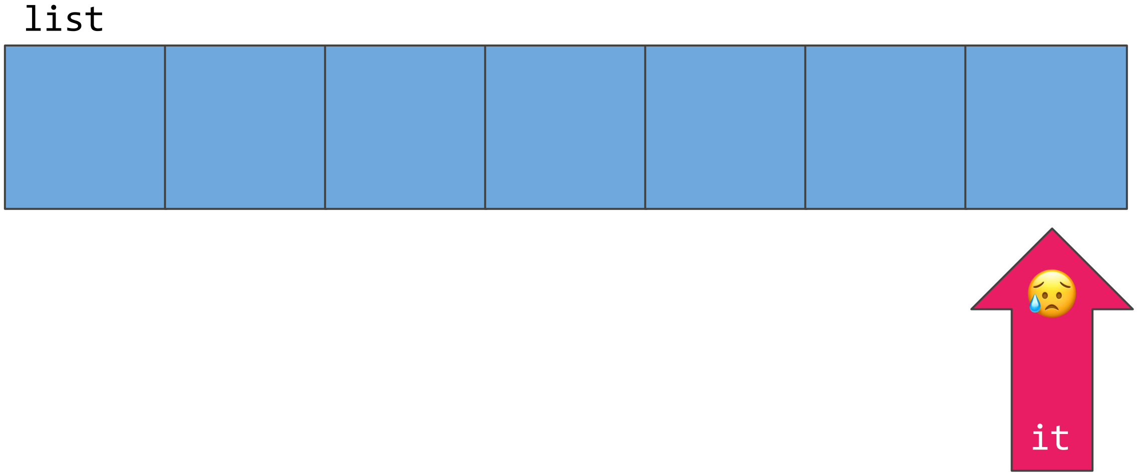
At the end of the list, it can't move to the next spot

```
while (it.hasNext()) {
```



At the end of the list, it can’t move to the next spot

```
    }
```



Implementing 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.

- 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();
}
```

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

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

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

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

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

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

Studying & Exam Strategy

Studying:

Optimize for understanding how everything fits together before how each part works individually

Become familiar with the textbook!

Don’t ask how, ask why a particular solution you see works

In the exam:

Optimize for what’s easy for you at first

Make sure a grader understands your thought processes

Remain calm

After the exam:

You’re done! We’ll take it from here.

Remember:

This exam does not define you.

Good luck!