Final Review Session

Brahm Capoor

Logistics

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

Last names A-G: 320-105

Last names H-O: 420-040

Last names P-Z: Bishop Auditorium

Come a little early!

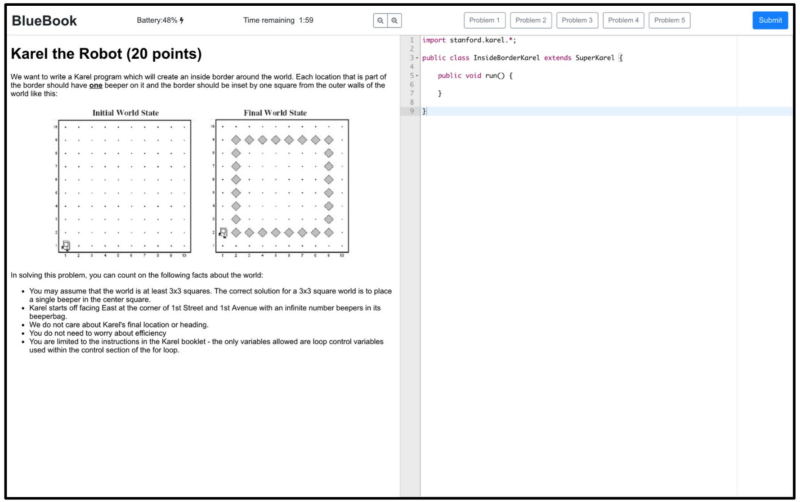
BlueBook

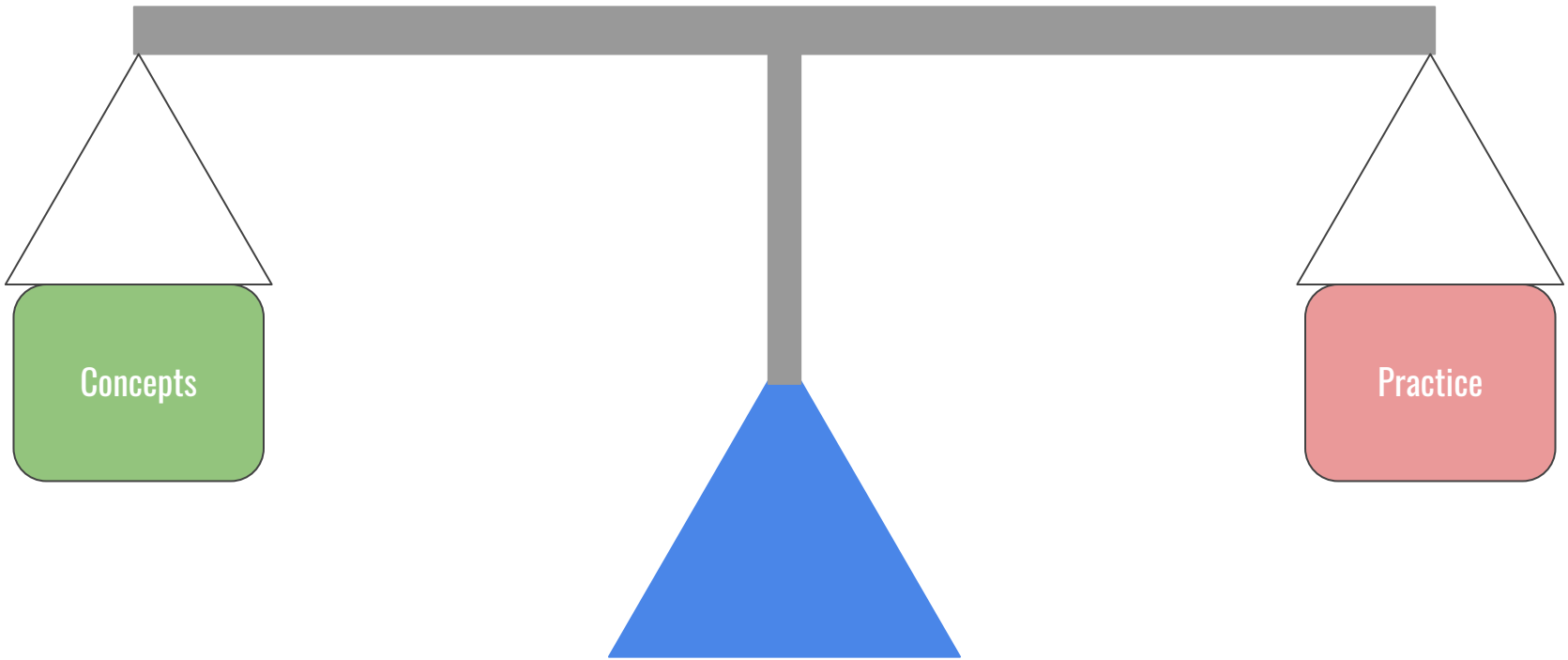
Download for Mac here

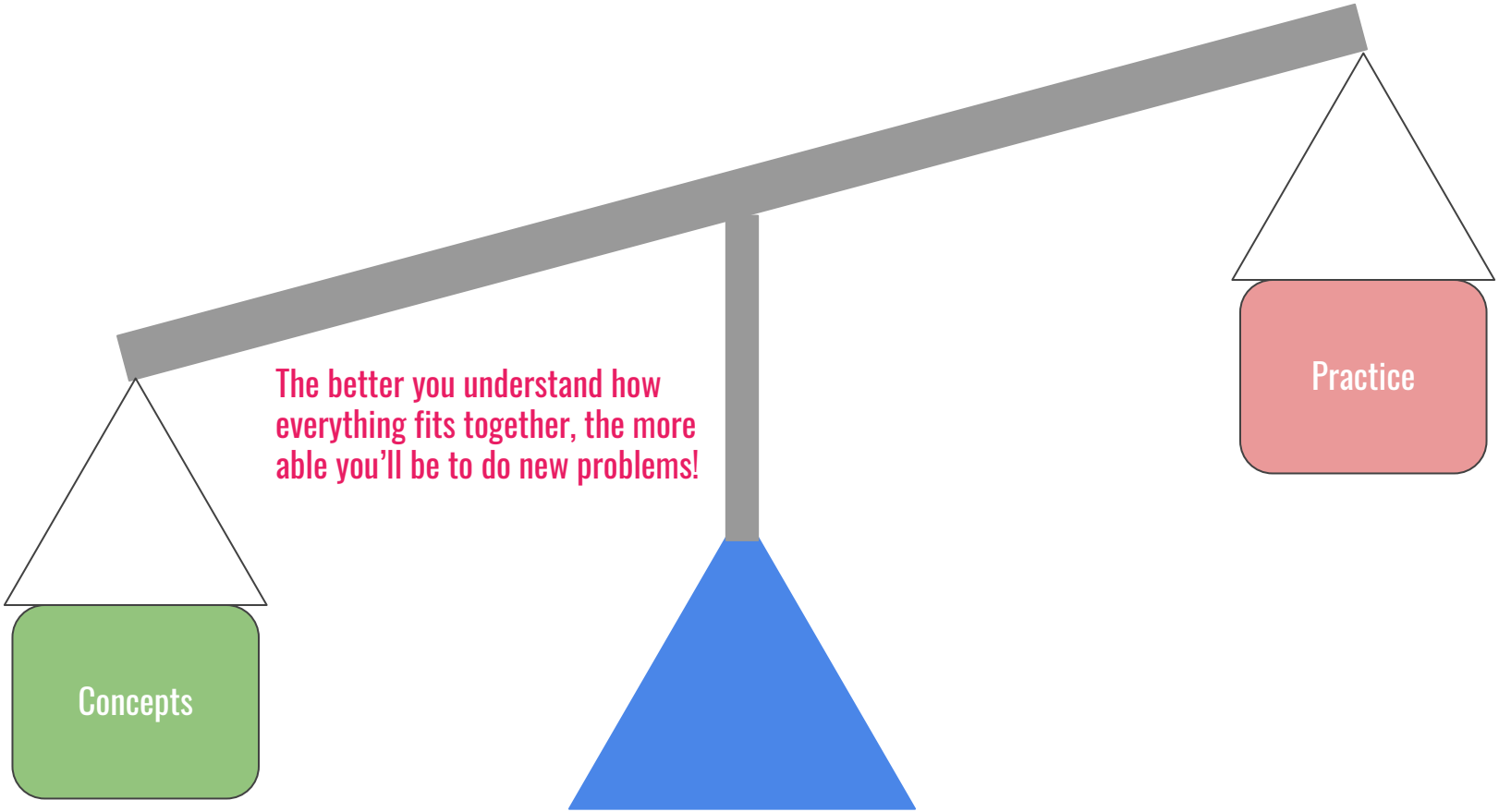
Download for Windows here

Handout here

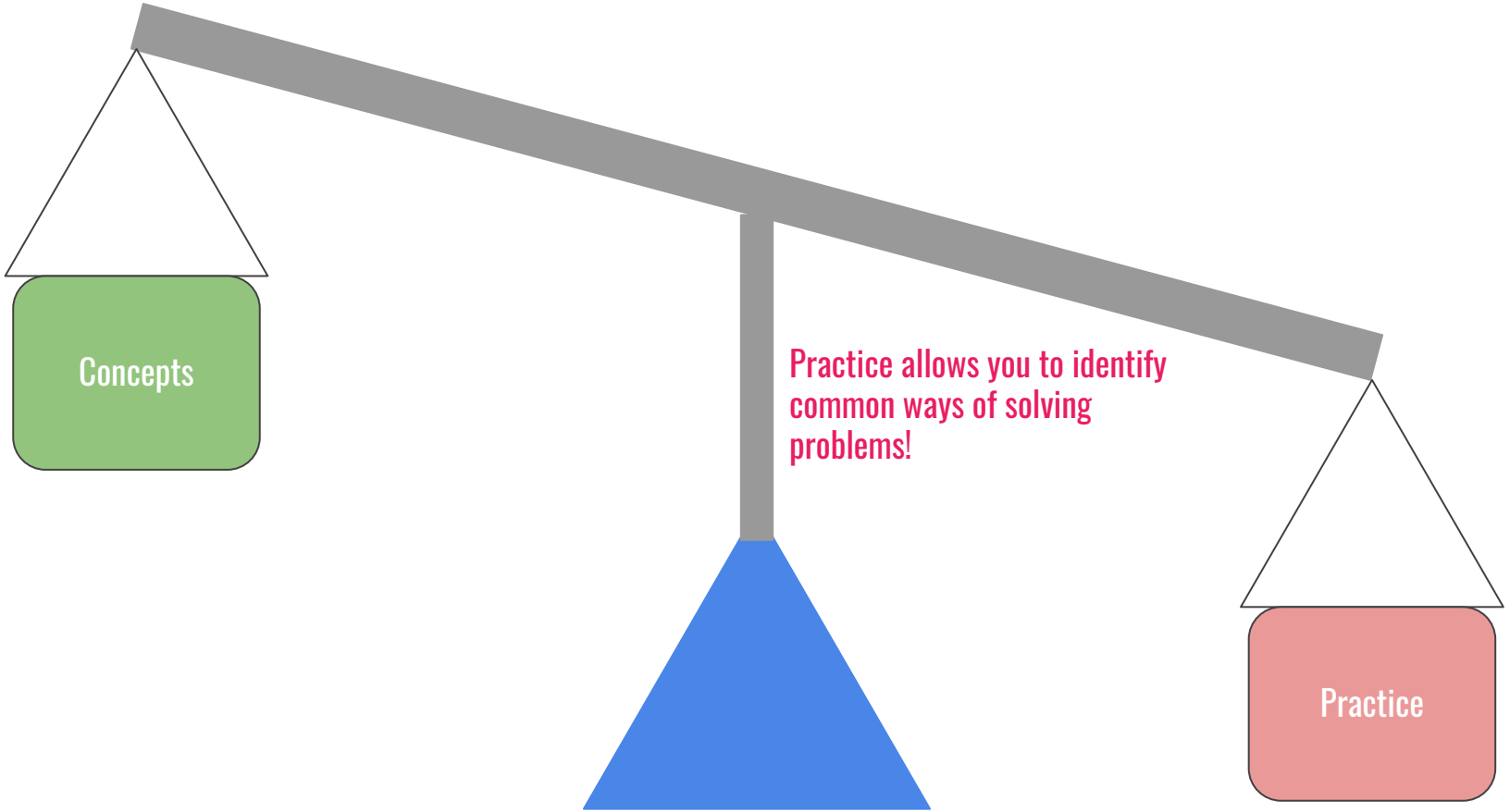
Make sure to have it installed and set up before the exam




Concepts
Practice



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


Concepts
Practice allows you to identify common ways of solving problems!
Practice

Where to find practice problems

Section handouts

Practice Final + Additional Practice Problems

CodeStepByStep

Textbook

Scattered throughout these slides

The Game Plan

Midterm Review

File Processing

Interactors

Collections

Classes

Server/Client

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 `doubles`
- 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!

What's a Character?

A `char` is a variable that represents a single letter, number or symbol.

Under the hood, it's a number (as specified by ASCII)

```
char upperA = 'A';

char upperB = (char)(uppercaseA + 1);

int numLetters = 'z' - 'a' + 1;
```

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

What can we do with a Character?

`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.

```
char c = 'b';
char upper = Character.toUpperCase(c);
boolean isDigit = Character.isDigit(c);
```

Characters are primitives, so we have a helper class with all these methods

What's a String?

A `String` is a variable that contains arbitrary text data

It consists of a series of `chars`, in order

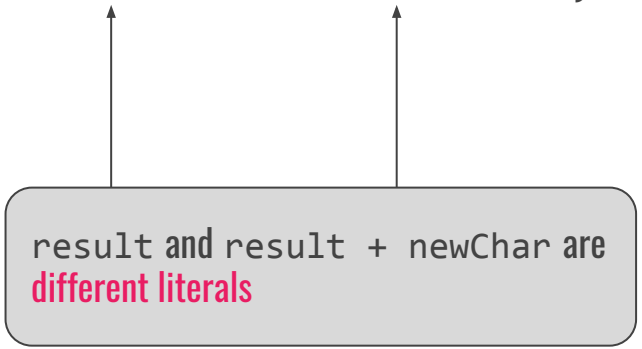
It is surrounded by double quotes

What can we do with a string?

`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

A common pattern for String problems

```
String str = “banter”;
String result = “”;                          // make a result string
for (int i = 0; i < str.length(); i++) {     // iterate through the original string
     char c = str.charAt(i);                 // get the i-th character
     char newChar = /* process c */;         // process the i-th character
     result = result + newChar;              // reassign the result string to a new
}                                            // literal
```



Turning stuff into Strings

```
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”
```

File Processing

```
try {
     Scanner sc = new Scanner(new File(filename));
     while (sc.hasNextLine()) {
          String line = sc.nextLine();
          println(“Just read: ” + line);
     }
     sc.close();
} catch (IOException ex) {
     throw new ErrorException(ex):
}
```

```
try {
     Scanner sc = new Scanner(new File(filename));
     while (sc.hasNextLine()) {
          String line = sc.nextLine();
          println("Just read: " + line);
     }
     sc.close();
} catch (IOException ex) {
     throw new ErrorException(ex):
}
```

Can only give you the next line in a file

```
try {
     Scanner sc = new Scanner(new File(filename));
     while (sc.hasNextLine()) {
          String line = sc.nextLine();
          println(“Just read: ” + line);
     }
     sc.close();
} catch (IOException ex) {
     throw new ErrorException(ex):
}
```

Try living dangerously

Life insurance

```
public void printFile() {
     try {
          Scanner sc = new Scanner(new File(“file.txt”));
          while (sc.hasNextLine()) {
               String line = sc.nextLine();
               println(“Just read: ” + line.toUpperCase());
          }
          sc.close();
     } catch (IOException ex) {
          throw new ErrorException(ex):
     }
}
```

file.txt

Space is limited
In a haiku, so it's hard
To finish what you

```
public void printFile() {
     try {
          Scanner sc = new Scanner(new File(“file.txt”));
          while (sc.hasNextLine()) {
               String line = sc.nextLine();
               println(“Just read: ” + line.toUpperCase());
          }
          sc.close();
     } catch (IOException ex) {
          throw new ErrorException(ex):
     }
}
```

file.txt

Space is limited
In a haiku, so it's hard
To finish what you

```
public void printFile() {
     try {
          Scanner sc = new Scanner(new File("file.txt"));
          while (sc.hasNextLine()) {
               String line = sc.nextLine();
               println("Just read: " + line.toUpperCase());
          }
          sc.close();
     } catch (IOException ex) {
          throw new ErrorException(ex):
     }
}
```

file.txt

Space is limited
In a haiku, so it's hard
To finish what you

```
public void printFile() {
     try {
          Scanner sc = new Scanner(new File(“file.txt”));
          while (sc.hasNextLine()) {
               String line = sc.nextLine();
               println(“Just read: ” + line.toUpperCase());
          }
          sc.close();
     } catch (IOException ex) {
          throw new ErrorException(ex):
     }
}
```

file.txt

Space is limited
In a haiku, so it's hard
To finish what you

```
public void printFile() {
     try {
          Scanner sc = new Scanner(new File(“file.txt”));
          while (sc.hasNextLine()) {
               String line = sc.nextLine();
               println(“Just read: ” + line.toUpperCase());
          }
          sc.close();
     } catch (IOException ex) {
          throw new ErrorException(ex):
     }
}
```

file.txt

Space is limited
In a haiku, so it's hard
To finish what you

```
just read: SPACE IS LIMITED
```

```
public void printFile() {
     try {
          Scanner sc = new Scanner(new File(“file.txt”));
          while (sc.hasNextLine()) {
               String line = sc.nextLine();
               println(“Just read: ” + line.toUpperCase());
          }
          sc.close();
     } catch (IOException ex) {
          throw new ErrorException(ex):
     }
}
```

file.txt

Space is limited
In a haiku, so it's hard
To finish what you

```
just read: SPACE IS LIMITED
```

```
public void printFile() {
     try {
          Scanner sc = new Scanner(new File(“file.txt”));
          while (sc.hasNextLine()) {
               String line = sc.nextLine();
               println(“Just read: ” + line.toUpperCase());
          }
          sc.close();
     } catch (IOException ex) {
          throw new ErrorException(ex):
     }
}
```

file.txt

Space is limited
In a haiku, so it's hard
To finish what you

```
just read: SPACE IS LIMITED
```

```
public void printFile() {
     try {
          Scanner sc = new Scanner(new File(“file.txt”));
          while (sc.hasNextLine()) {
               String line = sc.nextLine();
               println(“Just read: ” + line.toUpperCase());
          }
          sc.close();
     } catch (IOException ex) {
          throw new ErrorException(ex):
     }
}
```

file.txt

Space is limited
In a haiku, so it's hard
To finish what you

```
just read: SPACE IS LIMITED
just read: IN A HAIKU, SO IT’S HARD
```

```
public void printFile() {
     try {
          Scanner sc = new Scanner(new File(“file.txt”));
          while (sc.hasNextLine()) {
               String line = sc.nextLine();
               println(“Just read: ” + line.toUpperCase());
          }
          sc.close();
     } catch (IOException ex) {
          throw new ErrorException(ex):
     }
}
```

file.txt

Space is limited
In a haiku, so it's hard
To finish what you

```
just read: SPACE IS LIMITED
just read: IN A HAIKU, SO IT’S HARD
```

```
public void printFile() {
     try {
          Scanner sc = new Scanner(new File(“file.txt”));
          while (sc.hasNextLine()) {
               String line = sc.nextLine();
               println(“Just read: ” + line.toUpperCase());
          }
          sc.close();
     } catch (IOException ex) {
          throw new ErrorException(ex):
     }
}
```

file.txt

Space is limited
In a haiku, so it's hard
To finish what you

```
just read: SPACE IS LIMITED
just read: IN A HAIKU, SO IT’S HARD
```

```
public void printFile() {
     try {
          Scanner sc = new Scanner(new File(“file.txt”));
          while (sc.hasNextLine()) {
               String line = sc.nextLine();
               println(“Just read: ” + line.toUpperCase());
          }
          sc.close();
     } catch (IOException ex) {
          throw new ErrorException(ex):
     }
}
```

file.txt

Space is limited
In a haiku, so it's hard
To finish what you

```
just read: SPACE IS LIMITED
just read: IN A HAIKU, SO IT’S HARD
just read: TO FINISH WHAT YOU
```

```
public void printFile() {
     try {
          Scanner sc = new Scanner(new File(“file.txt”));
          while (sc.hasNextLine()) {
               String line = sc.nextLine();
               println(“Just read: ” + line.toUpperCase());
          }
          sc.close();
     } catch (IOException ex) {
          throw new ErrorException(ex):
     }
}
```

file.txt

Space is limited
In a haiku, so it's hard
To finish what you

```
just read: SPACE IS LIMITED
just read: IN A HAIKU, SO IT’S HARD
just read: TO FINISH WHAT YOU
```

```
public void printFile() {
     try {
          Scanner sc = new Scanner(new File(“file.txt”));
          while (sc.hasNextLine()) {
               String line = sc.nextLine();
               println(“Just read: ” + line.toUpperCase());
          }
          sc.close();
     } catch (IOException ex) {
          throw new ErrorException(ex):
     }
}
```

file.txt

Space is limited
In a haiku, so it's hard
To finish what you

```
just read: SPACE IS LIMITED
just read: IN A HAIKU, SO IT’S HARD
just read: TO FINISH WHAT YOU
```

```
public void printFile() {
    try {
        Scanner sc = new Scanner(new File(“file.txt”));
        while (sc.hasNextLine()) {
            String line = sc.nextLine();
            println(“Just read: ” + line.toUpperCase());
        }
        sc.close();
    } catch (IOException ex) {
        throw new ErrorException(ex):
    }
}
```

file.txt

Space is limited
In a haiku, so it's hard
To finish what you

```
just read: SPACE IS LIMITED
just read: IN A HAIKU, SO IT’S HARD
just read: TO FINISH WHAT YOU
```

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:

```
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.

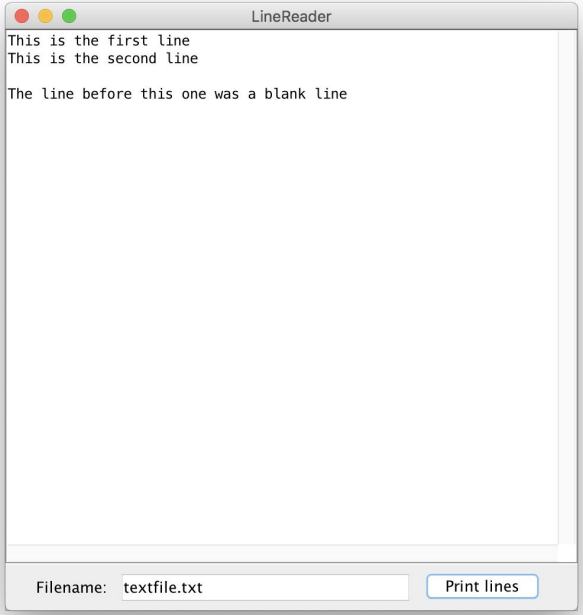
```
String maxName = “”;
int maxGuests = 0;

try {
     Scanner sc = new Scanner(new File(“guestList.txt”));
     while (sc.hasNextLine()) {
          String line = sc.nextLine();
          String[] parts = line.split(“ ”);
          String name = parts[0];
          int numGuests = Integer.parseInt(parts[1]);
          if (numGuests > maxGuests) {
               maxGuests = numGuests;
               maxName = name;
          }
     }
} catch (IOException Ex) {
     throw new ErrorException(Ex);
}
```

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



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




}
```

First, add the interactors in init()

```
private JTextField tf;


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

      tf = new JTextField(20);









}
```

`JTextFields` are always instance variables

```
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

```
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

```
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();
      }
      if (cmd.equals("Print lines")) {
            printFile()
      }
}
```

We call the printFile method defined in the last section

```
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();
      }
      if (cmd.equals("Print lines")) {
            printFile()
      }
}
```

Collections: `ArrayLists`, `HashMaps` and arrays

Arrays

Fixed size

Store objects or primitives

No methods, only `.length`

Ordered

Arrays

Fixed size

Store objects or primitives

No methods, only `.length`

Ordered

`ArrayLists`

Variable size

Store only objects

Methods

Ordered

Arrays

Fixed size

Store objects or primitives

No methods, only `.length`

Ordered

`ArrayLists`

Variable size

Store only objects

Methods

Ordered

`HashMaps`

Variable size

Store only objects

Methods

Key-Value Associations

Arrays

Fixed size

Store objects or primitives

No methods, only `.length`

Ordered

`ArrayLists`

Variable size

Store only objects

Methods

Ordered

`HashMaps`

Variable size

Store only objects

Methods

Key-Value Associations

Wrapper classes

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

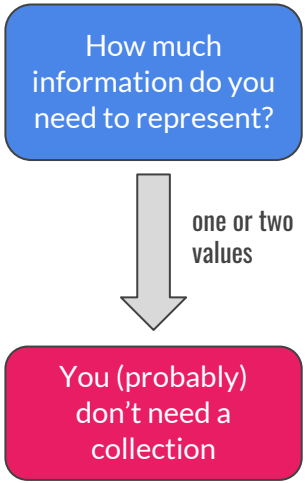
Use these instead

Disclaimer: We'll get to matrices in a sec! They're super important and worth understanding, but aren't usually a natural alternative to an Array, an `ArrayList`, or a `HashMap`

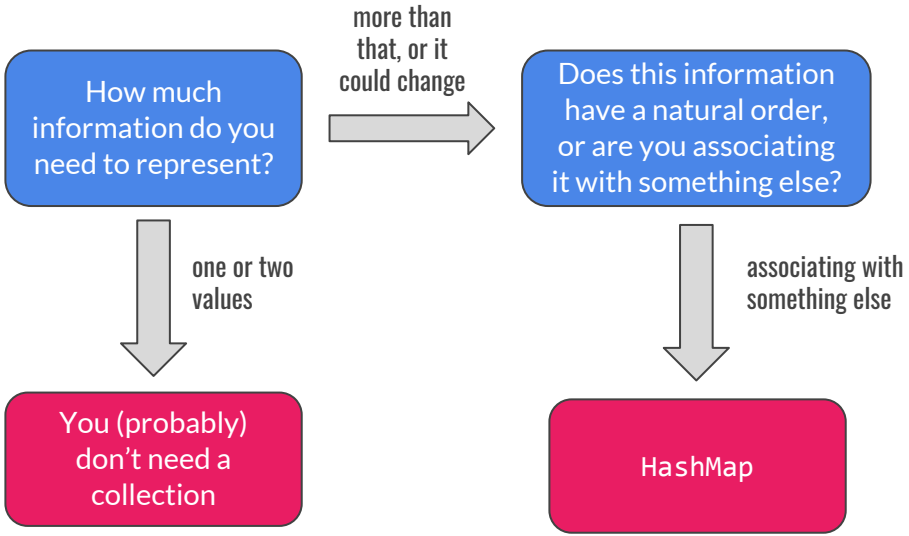
“What data structure should I use?”: A heuristic

How much information do you need to represent?

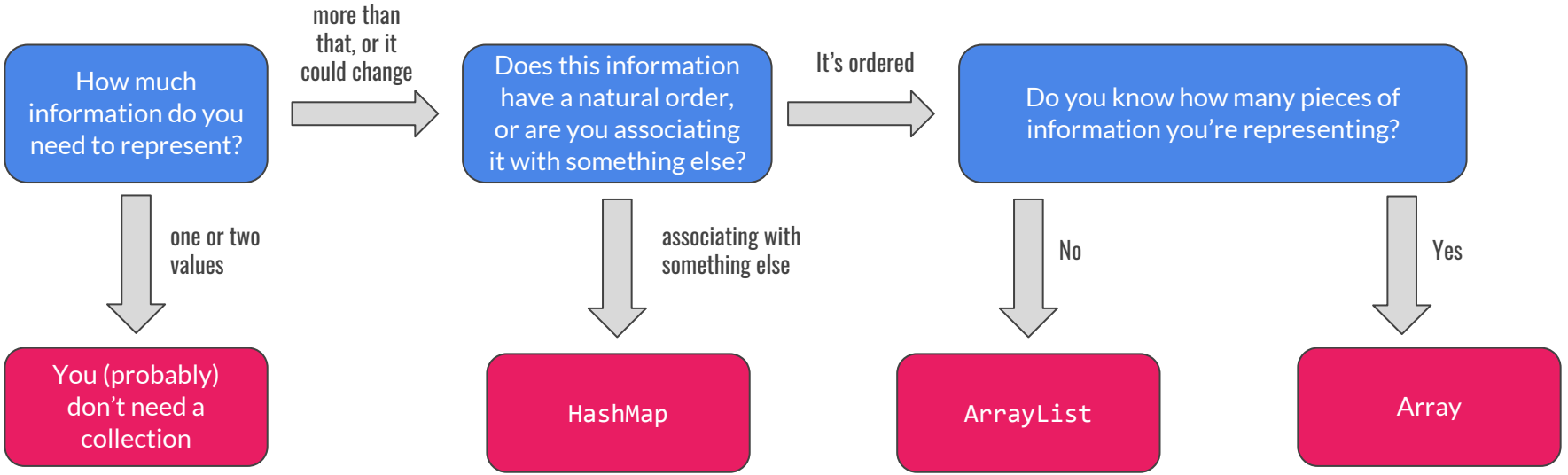
“What data structure should I use?”: A heuristic



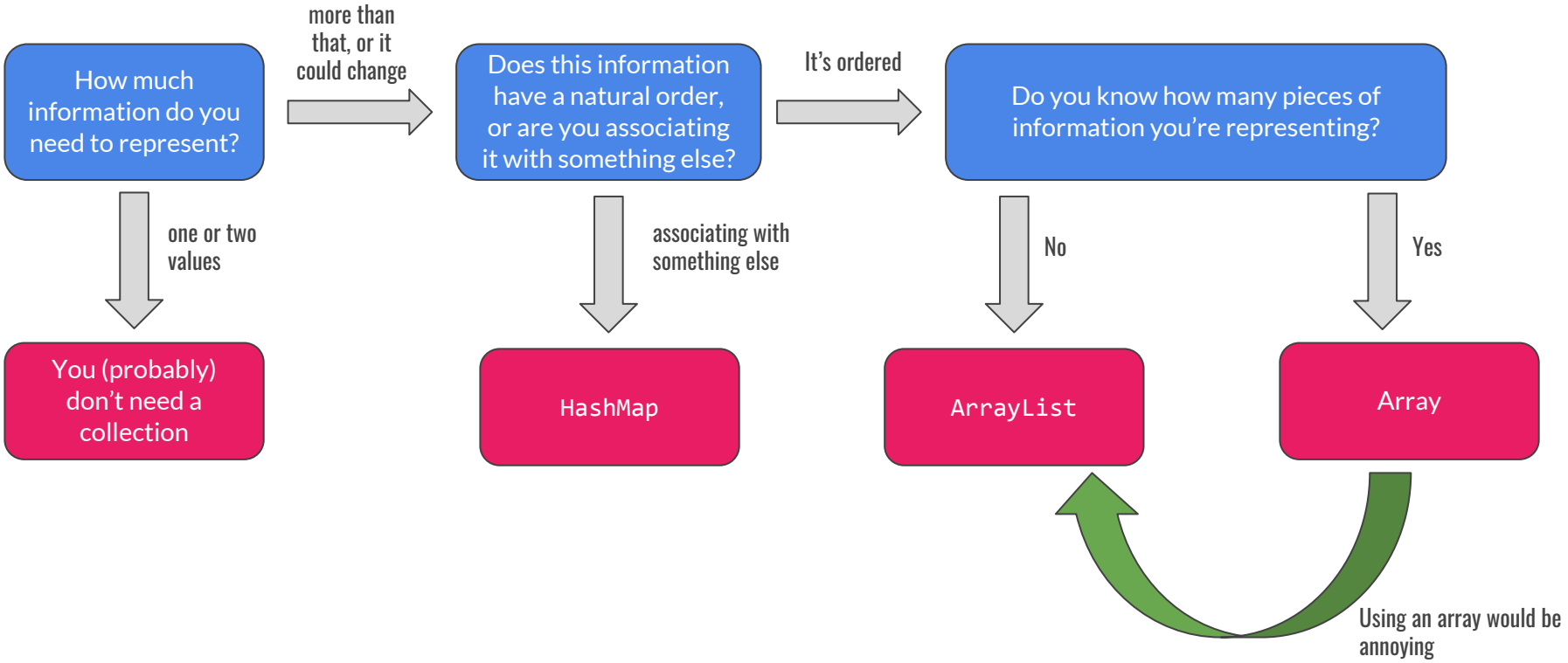
“What data structure should I use?”: A heuristic



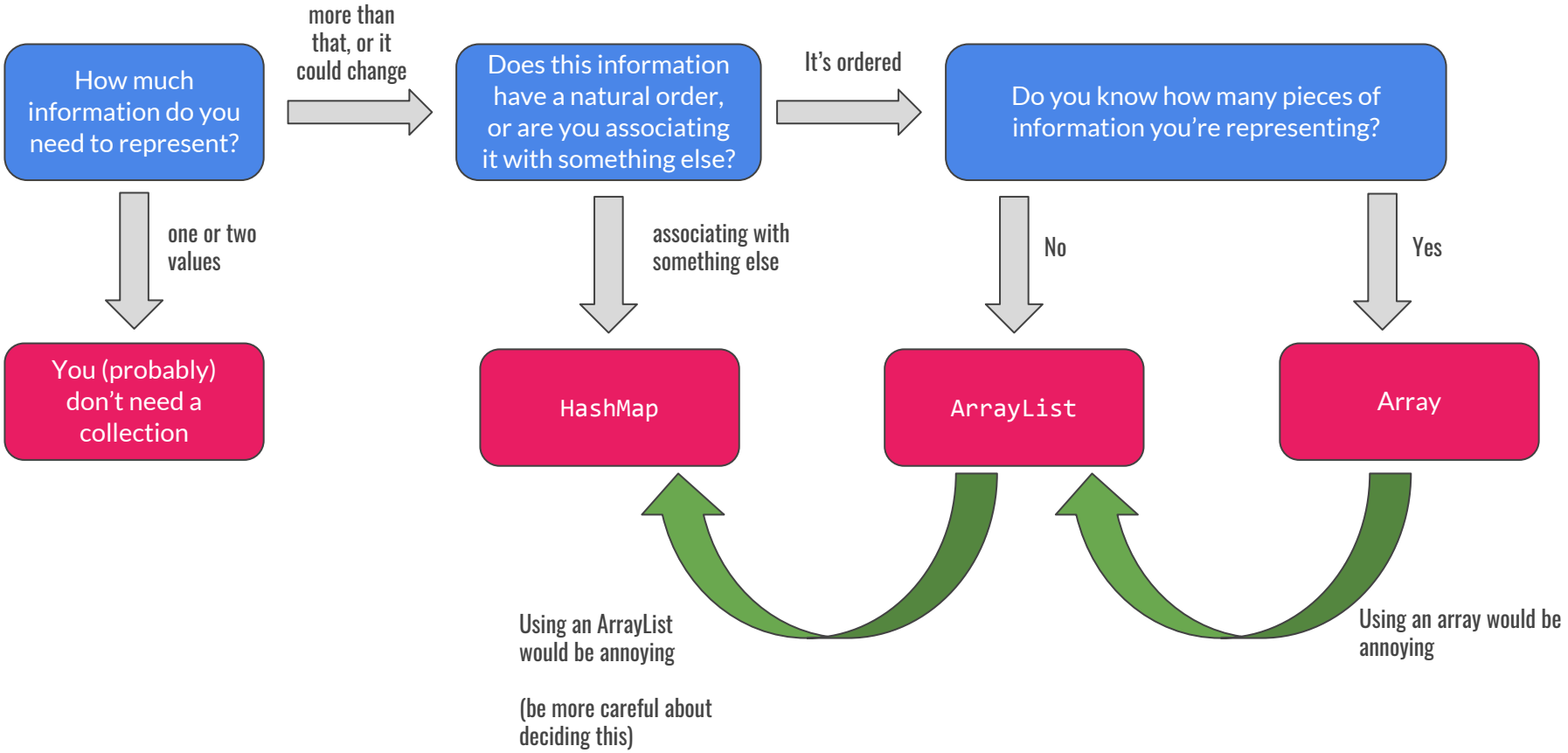
"What data structure should I use?": A heuristic



"What data structure should I use?": A heuristic



"What data structure should I use?": A heuristic



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”, “Mellany”, “Andrew”}
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
Mellany is in carriage 2, which has Mellany, Andrew
Andrew is in carriage 2, which has Mellany, Andrew
```

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”, “Mellany”, “Andrew”}
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
Mellany is in carriage 2, which has Mellany, Andrew
Andrew is in carriage 2, which has Mellany, Andrew
```

A problem: The Stanford Carriage Pact ☜(ﾟヮﾟ☜) (☞ﾟヮﾟ)☞

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”, “Mellany”, “Andrew”}
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
Mellany is in carriage 2, which has Mellany, Andrew
Andrew is in carriage 2, which has Mellany, Andrew
```

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>>`

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>>`

You could also use `String[]`, but the fact that the carriages are of different sizes feels a *little* annoying

```
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

```
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

```
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!

```
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);
      }
}
```