CS 106A Midterm Review Session

Brahm Capoor

Gameplan

Logistics

October 30th, 7-9 PM

Last names A-L: CEMEX Auditorium in the GSB

Last names M-Z: Hewlett 200 (where we have lecture)

Come a little early!

I’ll be holding extended office hours for midterm prep on Tuesday from 12 to 4 pm

BlueBook

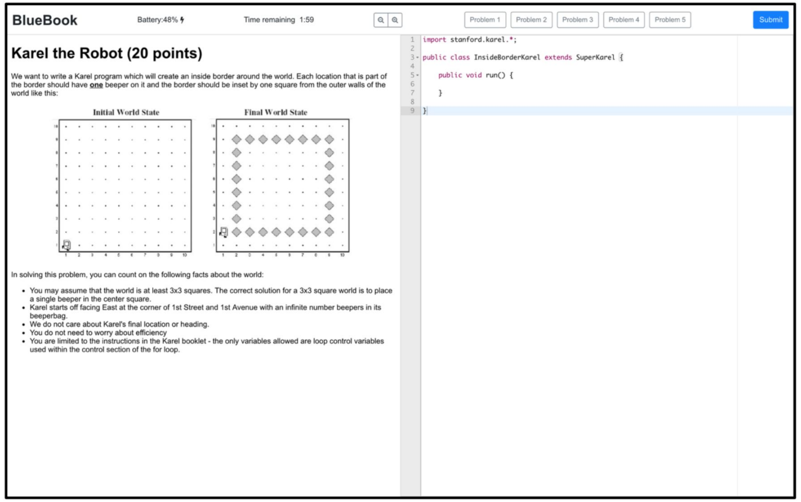
Download for Mac here

Download for Windows here

Handout here

Practice exam here (right click -> save link as)

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.

```java
import stanford.karel.*;

public class InsideBorderKarel extends SuperKarel {

    public void run() {

    }

}
```

Karel

Your general strategy for Karel problems

Figure out a general pattern of motion (strategy)

- What is the simplest and most general way Karel would move to solve this problem?

Figure out how to break up that motion (top-down decompose)

- What are the component parts of Karel’s motion?

Some common patterns of motion

Row-by-row, starting from the left

Column-by-column, starting from the bottom

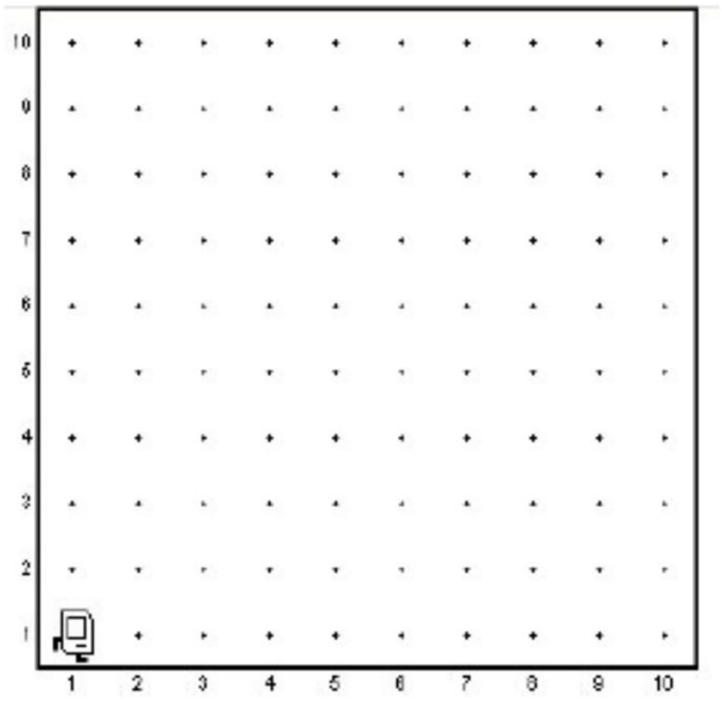
Follow the beepers

Follow the wall

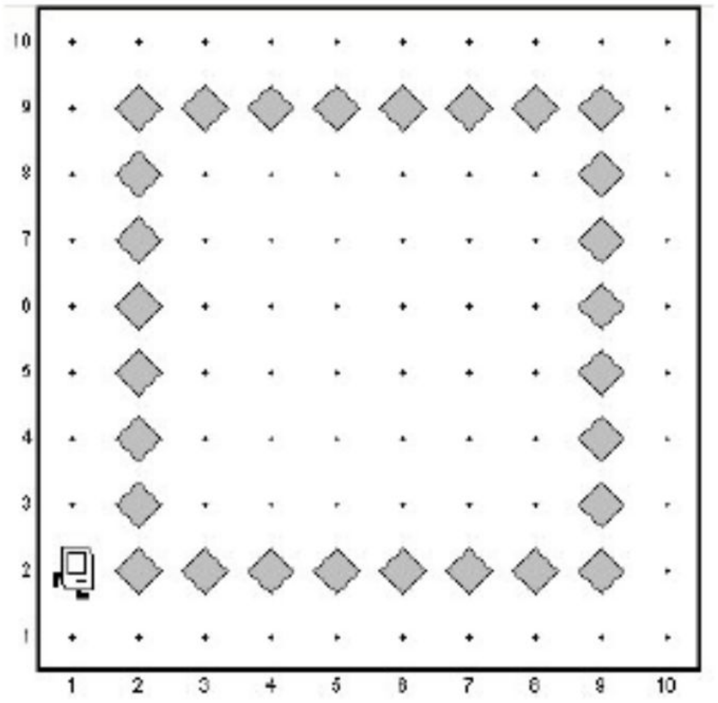
Diagonal (this is super rare)

Let’s do an example

Initial World State



Final World State



Our options

Row by row - kind of annoying, a different number of beepers per row

Column by column - kind of annoying, a different number of beepers per column

Follow the wall - doesn’t help here

Diagonal - ¯_(ツ)_/¯

Our options

Row by row - kind of annoying, a different number of beepers per row

Column by column - kind of annoying, a different number of beepers per column

Follow the wall - doesn’t help here

Diagonal - ¯_(ツ)_/¯

Follow the beepers - this could work!

Our strategy

Motion pattern: ‘Follow the beepers’

- Get to a starting position, and then lay down each edge

How to decompose this motion

- Getting to a starting position: `moveUpRow()`
- Lay down an edge: `handleBorder()`
- Move to the next edge: `nextPosition()`

Our strategy

Motion pattern: ‘Follow the beepers’

- Get to a starting position, and then lay down each edge

How to decompose this motion

- Getting to a starting position: `moveUpRow()`
- Lay down an edge: `handleBorder()`
- Move to the next edge: `nextPosition()`

```
public void run() {
	moveUpRow();
	for (int i = 0; i < 4; i++) {
		handleBorder();
		nextPosition();
	}
}
```

Our strategy

Motion pattern: ‘Follow the beepers’

- Get to a starting position, and then lay down each edge

How to decompose this motion

- Getting to a starting position: `moveUpRow()`
- Lay down an edge: `handleBorder()`
- Move to the next edge: `nextPosition()`

```
private void moveUpRow() {
      turnLeft();
      move();
      turnRight();
}

private void handleBorder() {
      move();
      while (frontIsClear()) {
            if (noBeepersPresent()) {
                  putBeeper();
            }
            move();
      }
}


private void nextPosition() {
      turnRight();
      move();
      turnRight();
      move();
      turnRight();
}
```

Some last things to remember

No non-Karel features! (Variables, parameters, return values, break statements etc)

Postconditions of a code block should match the preconditions of the next code block

- If one loop requires that the front is clear, the lines of code before it should guarantee that
- Applies to methods, loops, if statements and individual lines of code

Java

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 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

Things to remember about variables

The expressive hierarchy

```
boolean < char < int < double
```

Compare primitive variables using ==

```
if (x == 7) {...}
```

Conditional operators: && and | |

```
if (x == 7 && y == 6.3)
if (x == 7 || x == 6)
```

Avoid this:

```
if (x == 7 || 6)
```

Use constants!

```
private static final int MY_NUM = 10;
```

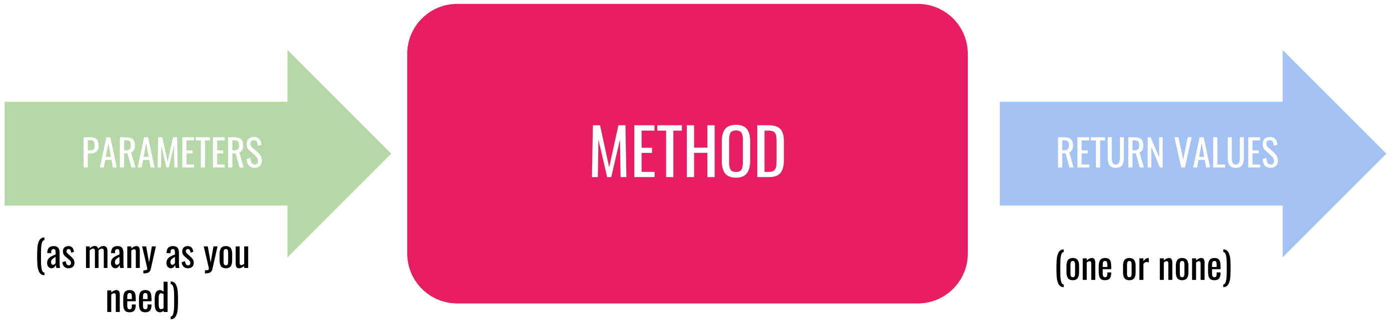
Methods

```
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



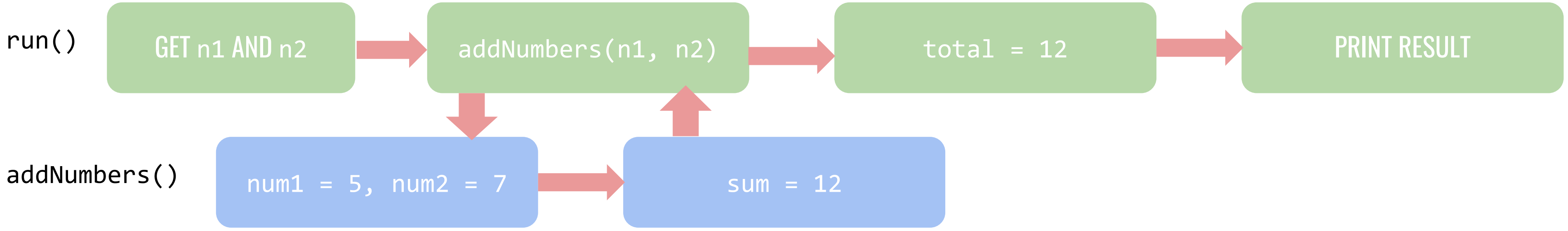
```
private returnType methodName(type parameter1, type parameter2,...)

private int returnsInt() {...}
private void drawsRect(int width, int length) {...} //void is no type
public boolean frontIsClear() {...} //look familiar?
```

Parameters and a return value are both optional!

Example: Methods and Parameters

```
public void run() {
     println("Choose 2 numbers!");
     int n1 = readInt("Enter n1"); //5
     int n2 = readInt("Enter n2"); //7

     int total = addNumbers(n1, n2);
     println ("The total is " + total);
}
```

```
                                  5          7
private int addNumbers(int num1, int num2) {
     int sum = num1 + num2;   //12
     return sum;
}
```



Variable scope

Variables live inside the block, or pair of braces, in which they're declared

Scope for i

Scope for y

```
for (int i = 0; i < 5; i++) {
    int y = i * 4;
}
i = 3; // Error!
y = 2; // Error!

... // in some code far, far away
int y = 0;
for (int i = 0; i < 5; i++) {
    y = i * 4;
}
y = 2;
```

Scope for y

Returning in different places

```
private int multipleReturns(int x) {

      if (x == 5) {
            return 0;
      }

      return 1; // this only happens if x != 5
      return 5; // never gets to this line
}


// note: every path through the method ends
with a single return statement

// note: a function ends immediately after it
returns
```

A trace problem

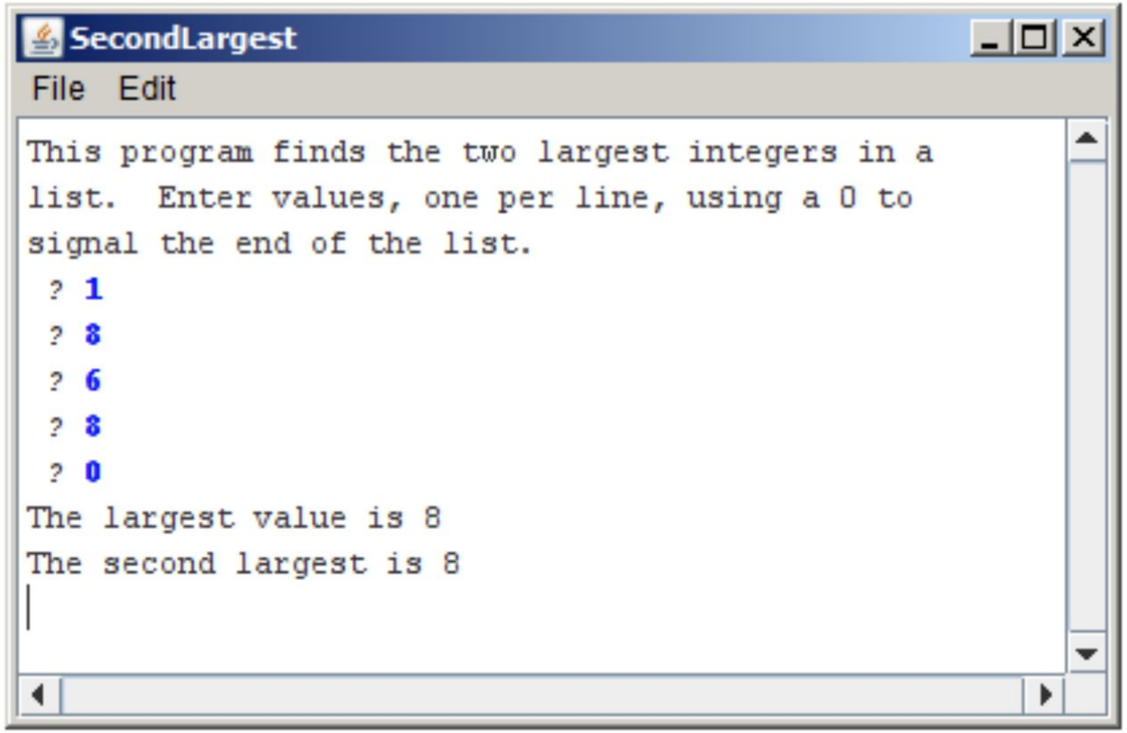
```
public void run() {
     int num1 = 2;
     int num2 = 13;
     println("The 1st number is: " + Mystery(num1, 6));
     println("The 2nd number is: " + Mystery(num2 % 5, 1 + num1 * 2));
}

private int Mystery(int num1, int num2) {
     num1 = Unknown(num1, num2);
     num2 = Unknown(num2, num1);
     return(num2);
}

private int Unknown(int num1, int num2) {
     int num3 = num1 + num2;
     num2 += num3 * 2;
     return num2;
}
```

Our strategy: draw stack frames and trace through each line

A trace problem

```
public void run() {
     int num1 = 2;
     int num2 = 13;
     println("The 1st number is: " + Mystery(num1, 6));
     println("The 2nd number is: " + Mystery(num2 % 5, 1 + num1 * 2));
}

private int Mystery(int num1, int num2) {
     num1 = Unknown(num1, num2);
     num2 = Unknown(num2, num1);
     return(num2);
}

private int Unknown(int num1, int num2) {
     int num3 = num1 + num2;
     num2 += num3 * 2;
     return num2;
}
```

Another problem



```
This program finds the two largest integers in a
list.  Enter values, one per line, using a 0 to
signal the end of the list.
 ? 1
 ? 8
 ? 6
 ? 8
 ? 0
The largest value is 8
The second largest is 8
```

Questions I would ask myself about this problem

What information do I need to store? Where does it need to be available?

What structures lend themselves best to the repeating nature of this problem?

How should I treat the numbers that the user enters?

How I’d answer them

What information do I need to store? Where does it need to be available?

> *It feels like I need to keep track of the largest and second largest outside the loop*

What structures lend themselves best to the repeating nature of this problem?

> *A while loop, because I don’t know how many numbers the user will enter*

How should I treat the numbers that the user enters?

> *I should compare them to my current largest numbers and update them accordingly*

```java
public void run() {
     println("This program finds the two largest integers in a");
     println("list. Enter values, one per line, using a " + SENTINEL + " to");
     println("signal the end of the list.");
















}
```

```
public void run() {
     println("This program finds the two largest integers in a");
     println("list. Enter values, one per line, using a " + SENTINEL + " to");
     println("signal the end of the list.");

     int largest = -1;
     int secondLargest = -1;











     println("The largest value is " + largest);
     println("The second largest is " + secondLargest);
}
```

```java
public void run() {
     println("This program finds the two largest integers in a");
     println("list. Enter values, one per line, using a " + SENTINEL + " to");
     println("signal the end of the list.");

     int largest = -1;
     int secondLargest = -1;
     while (true) {









     }
     println("The largest value is " + largest);
     println("The second largest is " + secondLargest);
}
```

```
public void run() {
    println("This program finds the two largest integers in a");
    println("list. Enter values, one per line, using a " + SENTINEL + " to");
    println("signal the end of the list.");

    int largest = -1;
    int secondLargest = -1;
    while (true) {
        int input = readInt(" ? ");
        if (input == SENTINEL) break;







    }
    println("The largest value is " + largest);
    println("The second largest is " + secondLargest);
}
```

```java
public void run() {
    println("This program finds the two largest integers in a");
    println("list. Enter values, one per line, using a " + SENTINEL + " to");
    println("signal the end of the list.");

    int largest = -1;
    int secondLargest = -1;
    while (true) {
        int input = readInt(" ? ");
        if (input == SENTINEL) break;
        if (input > largest) {
            secondLargest = largest;
            largest = input;
        }


    }
    println("The largest value is " + largest);
    println("The second largest is " + secondLargest);
}
```

```
public void run() {
    println("This program finds the two largest integers in a");
    println("list. Enter values, one per line, using a " + SENTINEL + " to");
    println("signal the end of the list.");

    int largest = -1;
    int secondLargest = -1;
    while (true) {
        int input = readInt(" ? ");
        if (input == SENTINEL) break;
        if (input > largest) {
            secondLargest = largest;
            largest = input;
        } else if (input > secondLargest) {
            secondLargest = input;
        }
    }
    println("The largest value is " + largest);
    println("The second largest is " + secondLargest);
}
```

Graphics & Animation

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!

Animation

```
while(executing condition) {
     // update graphics
     obj.move(dx, dy);
     pause(PAUSE_TIME_MILLISEC);
}
```

Classes & Interfaces

Programming involves **things** which have **properties** and **behaviour**

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`

$$\underbrace{\texttt{public class <ClassName>}}_{}\ \underbrace{\texttt{extends <SuperClass>}}_{}$$

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

    // sick code here

}
```

Student.java

```java
public class Student {

   // sick code here

}
```

Stanford.java

Creating objects of type Student

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

Instance variables

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

```
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

```

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 */
}
```

Now we can make students!

```
public class Student {

    public Student(String name, int id, String email,
                   int numUnits, boolean isInternational) {...}

}
```

```
public void run() {

    Student s1 = new Student(“Brahm”, 31415926, “brahm@stanford.edu”,
                             180, true);
}
```

Getters and Setters

```
public class Student {

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

    public int getUnits() {
        return numUnits;
    }

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

    private int numUnits;

}
```

```
public void run() {

    Student s1 = new Student(42);


    println(“Curr:” + s1.getUnits());



    s1.setUnits(60);




}
```

Getters and Setters: some notes

```
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:

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

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

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

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

Interfaces

A list of methods representing non-unique characteristics of a particular class

GOvals are GFillable, GMoveable and GScalable

GLines are GMoveable and GScalable

GLabels are GMoveable

To implement an interface, we have to define all these methods in our own class

Let different classes tell Java they have the same behaviour, but allow for different implementations

Memory

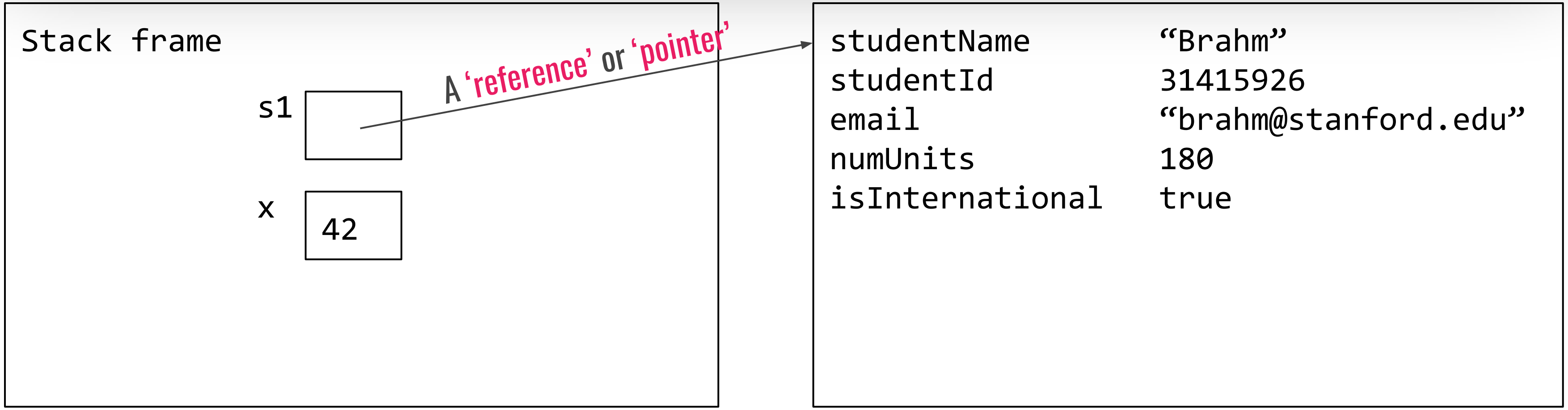
Passing parameters

```java
public void run() {
     int x = 7;
     doSomething(x);
     println(x); // prints 7
}

private void doSomething(int n) {
     n *= 2;
}
```

```java
public void run() {
     Student s1 = new Student(42);
     doSomething(s1);
     println(s1.getUnits()); // prints 84
}

private void doSomething(Student s) {
     s.setUnits(s.getUnits() * 2);
}
```

Under the hood

```
Student s1 = new Student("Brahm", 31415926, "brahm@stanford.edu", 180, true);
int x = 42;
```



studentName	"Brahm"
studentId	31415926
email	"brahm@stanford.edu"
numUnits	180
isInternational	true

Going a little deeper

There are two main parts of memory: the stack and the heap

The stack stores local variables, and references to objects

The heap stores objects themselves

== compares whatever’s in the stack

Going even deeper

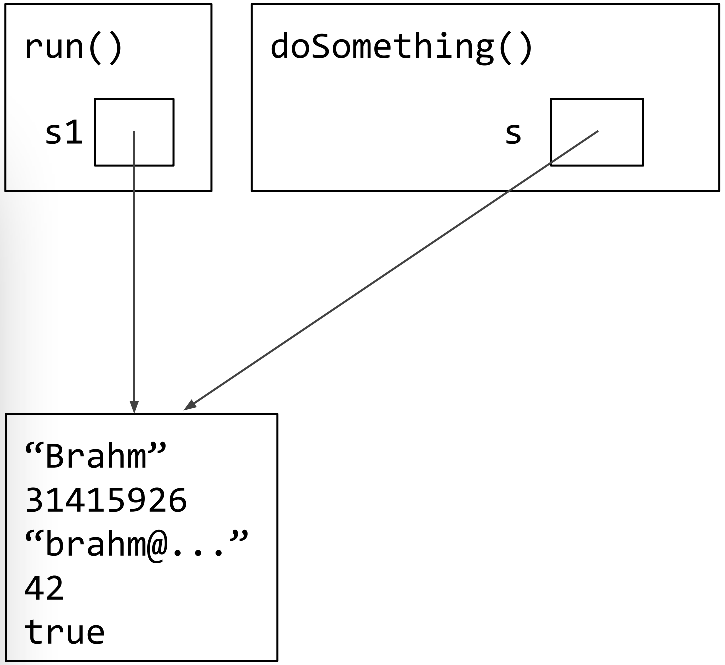
When we pass a parameter, we pass a copy of whatever’s on the stack

For a primitive, that’s a copy of a value

For an object, that’s a copy of a reference

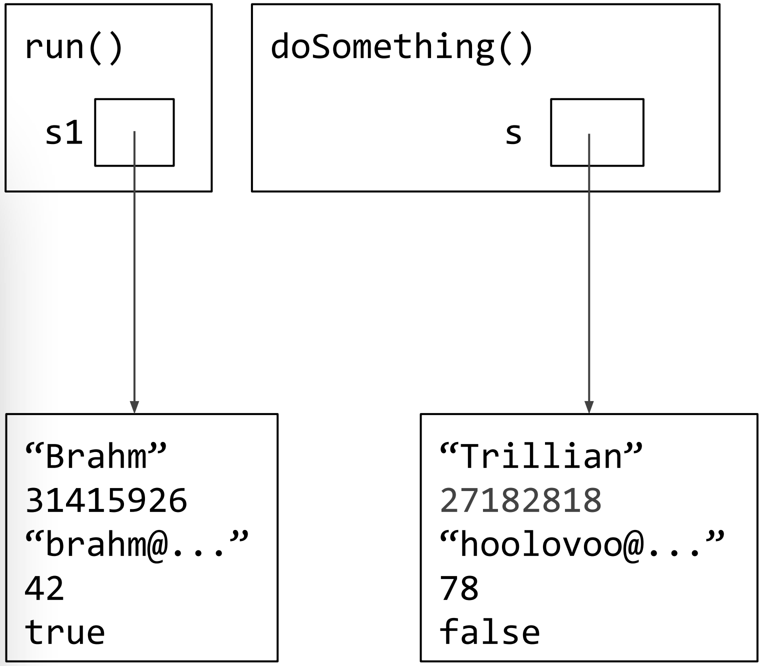
What does that mean?

```
public void run() {
    Student s1 = new Student(...);
➡   doSomething(s1);
    println(s1.getUnits());
}

private void doSomething(Student s) {
    s = new Student(...);
}
```



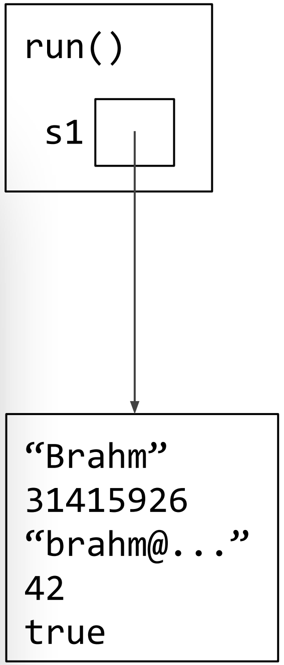
What does that mean?

```
public void run() {
     Student s1 = new Student(...);
     doSomething(s1);
     println(s1.getUnits());
}

private void doSomething(Student s) {
     s = new Student(...);
}
```



What does that mean?

```
public void run() {
     Student s1 = new Student(...);
     doSomething(s1);
➡    println(s1.getUnits());
}

private void doSomething(Student s) {
     s = new Student(...);
}
```



Event Driven Programming

Why is it necessary?

We tell our computer what to do, and when to do it

We don’t know when a user will click their mouse or type something

We need to specify the behaviour of our program if something happens rather than saying when it will happen

This programmed behaviour is driven by events out of the control of the program

Two parts to Event Driven programming

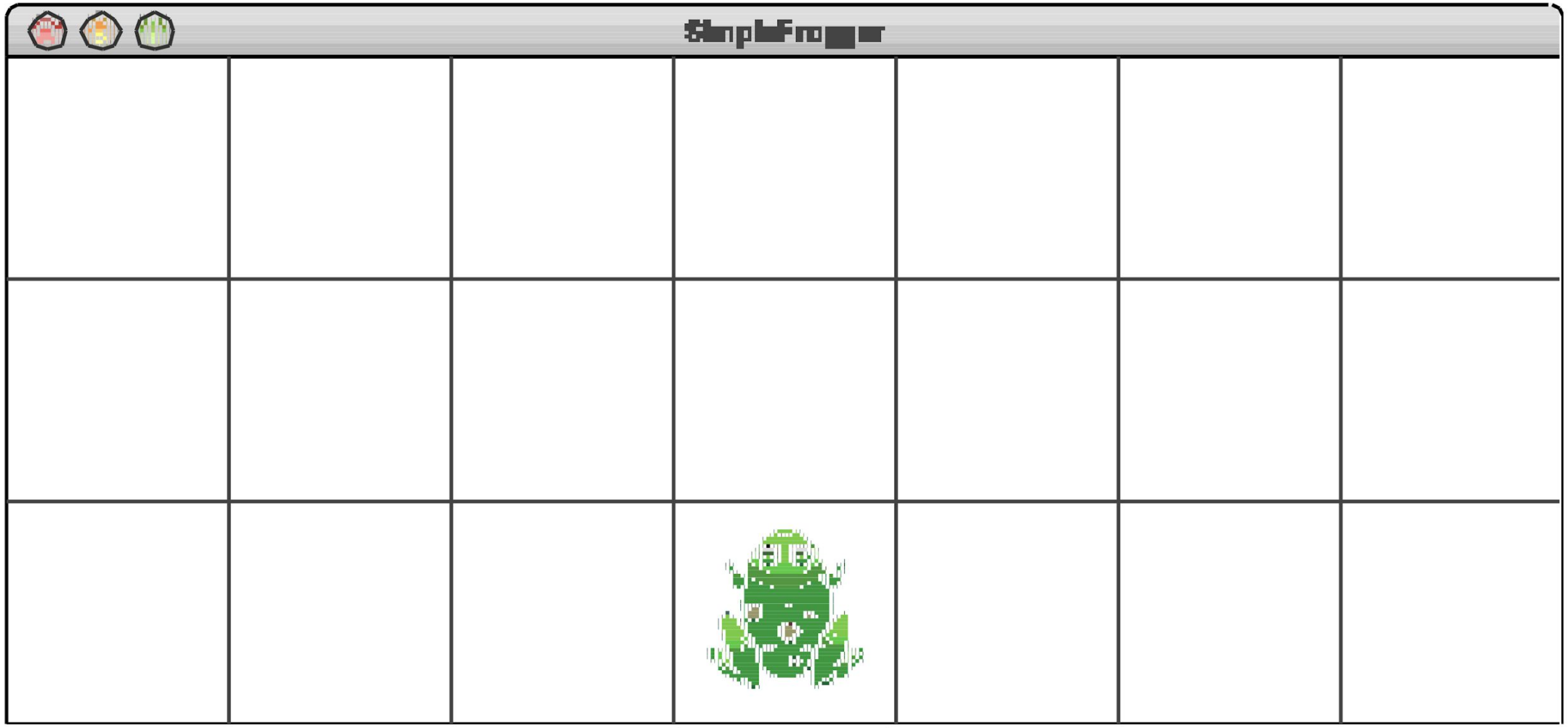
1) Subscribe to notifications about user events

```
addMouseListeners();
```

2) Specify behaviour when an event occurs

```
public void mouseClicked(MouseEvent e) {
    double clickX = e.getX();
    double clickY = e.getY();
    // process click
}
```

A good problem to think about



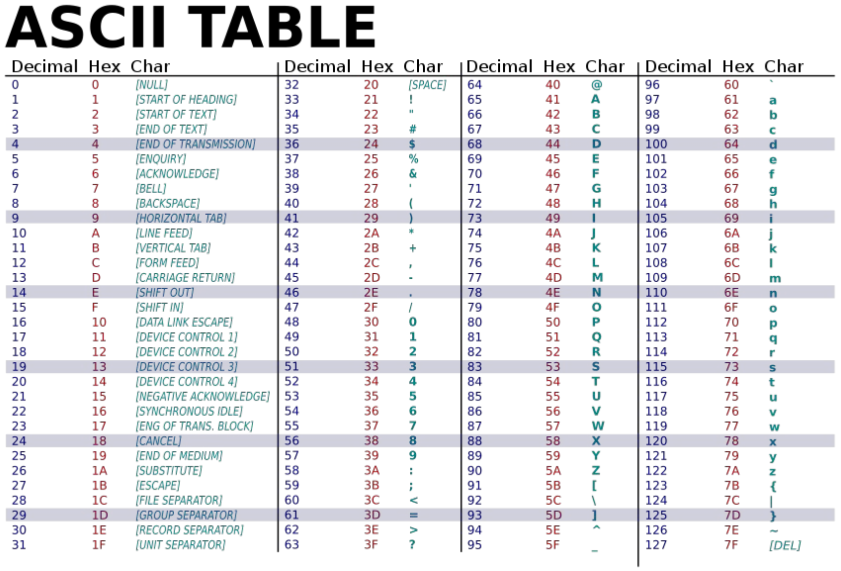
Characters & Strings

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)

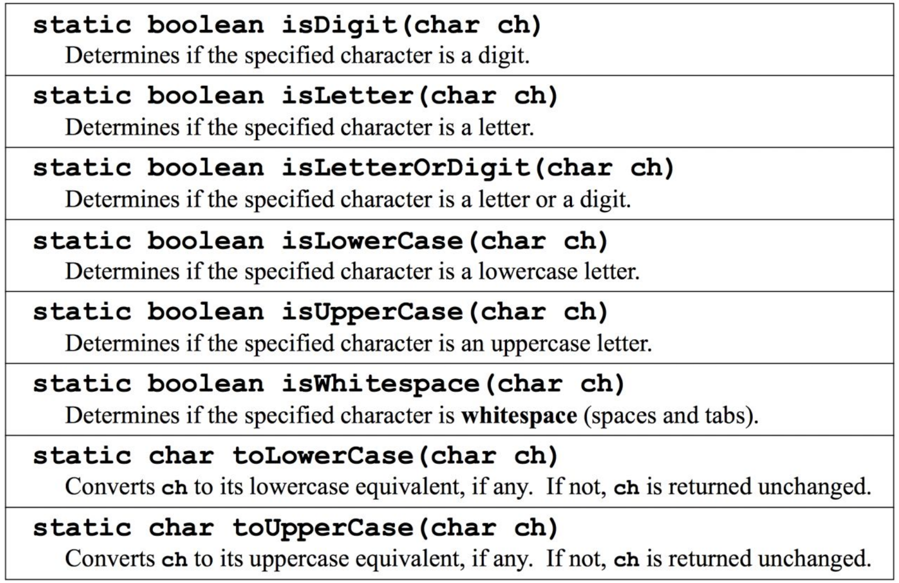
```
char upperA = ‘A’;

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

int numLetters = ‘z’ - ‘a’ + 1;
```



ASCII TABLE

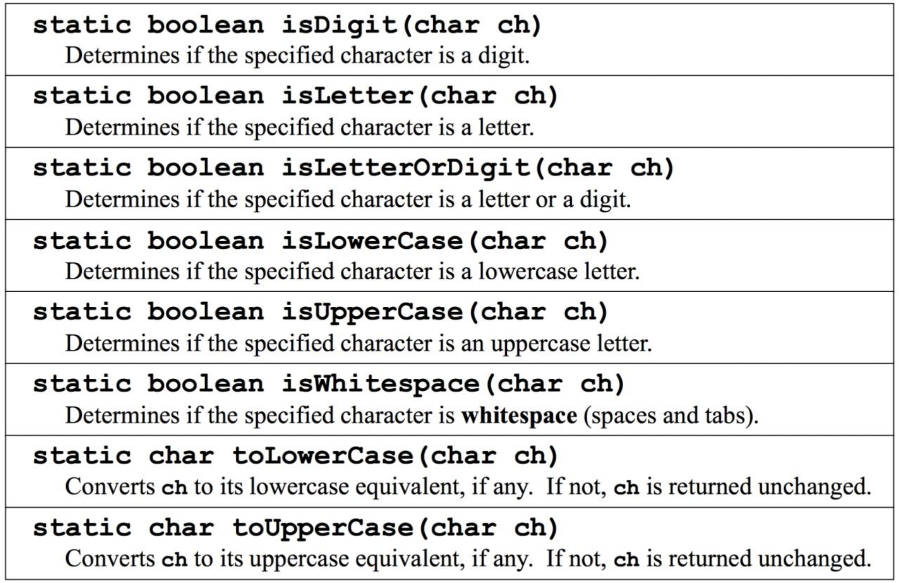
Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	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.

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

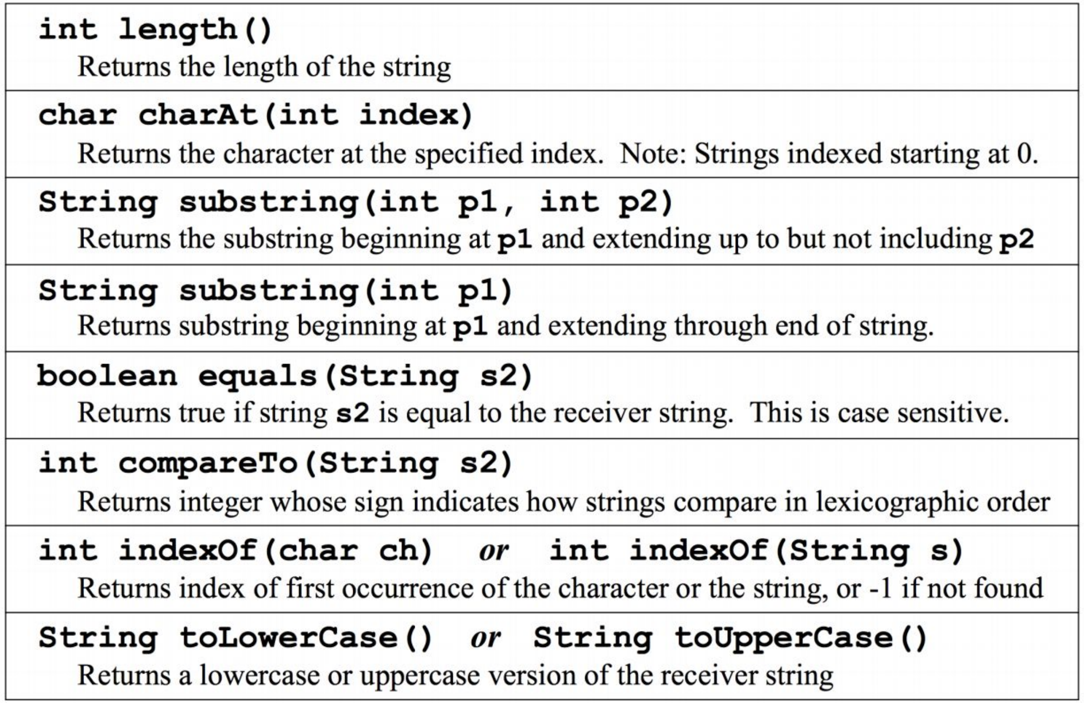
What’s a String?

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

It consists of a series of `char`s, 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

Strings are 0-indexed

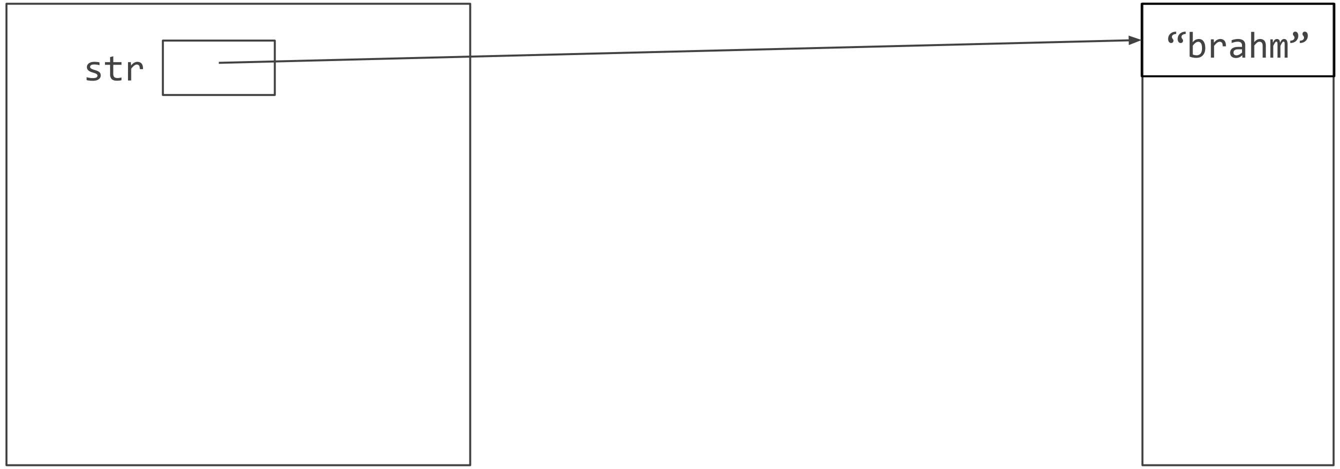
```
“banter”
 0 1 2 3 4 5
```

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”
```

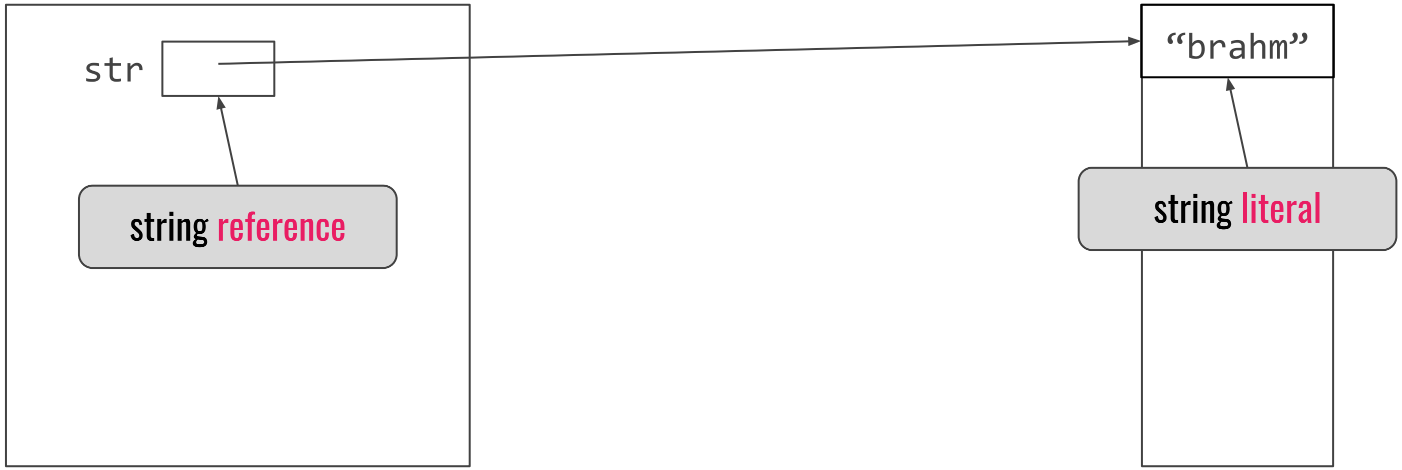
Strings are objects

```
String str = “brahm”;
```



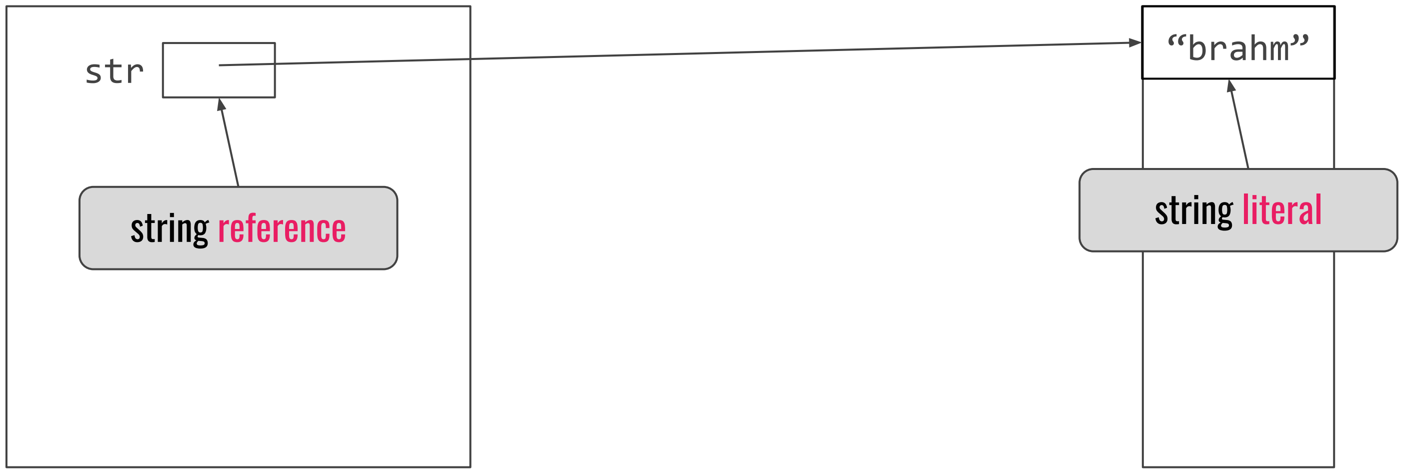
Strings are objects

```
String str = “brahm”;
```



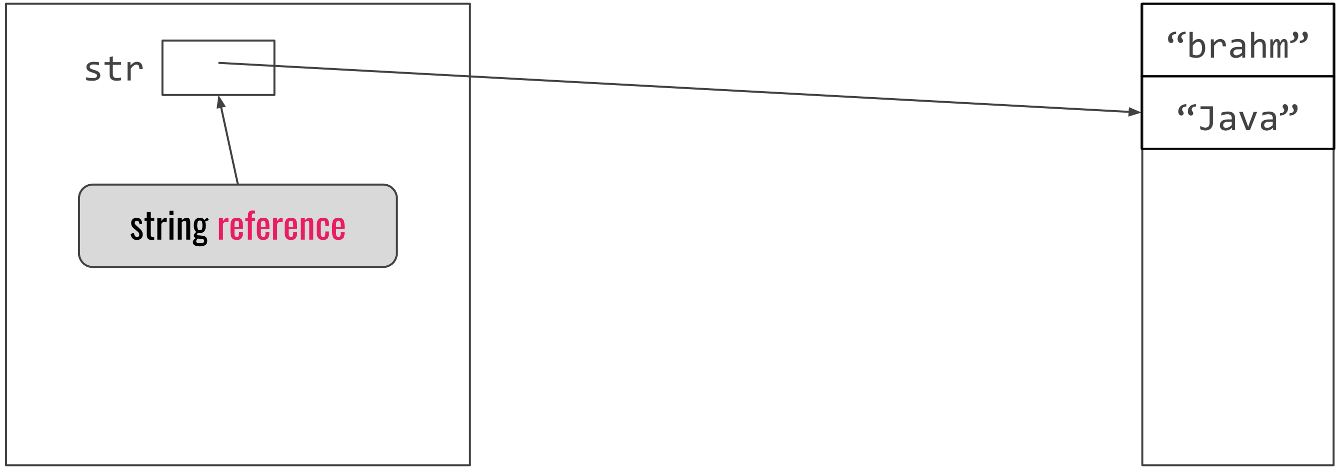
An important nuance: string literals are immutable

```
String str = “brahm”;
```



...but references aren't!

```
String str = "Java";
```



This leads to 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
```

`result` and `result + newChar` are different literals

Why are Strings immutable?

¯_(ツ)_/¯

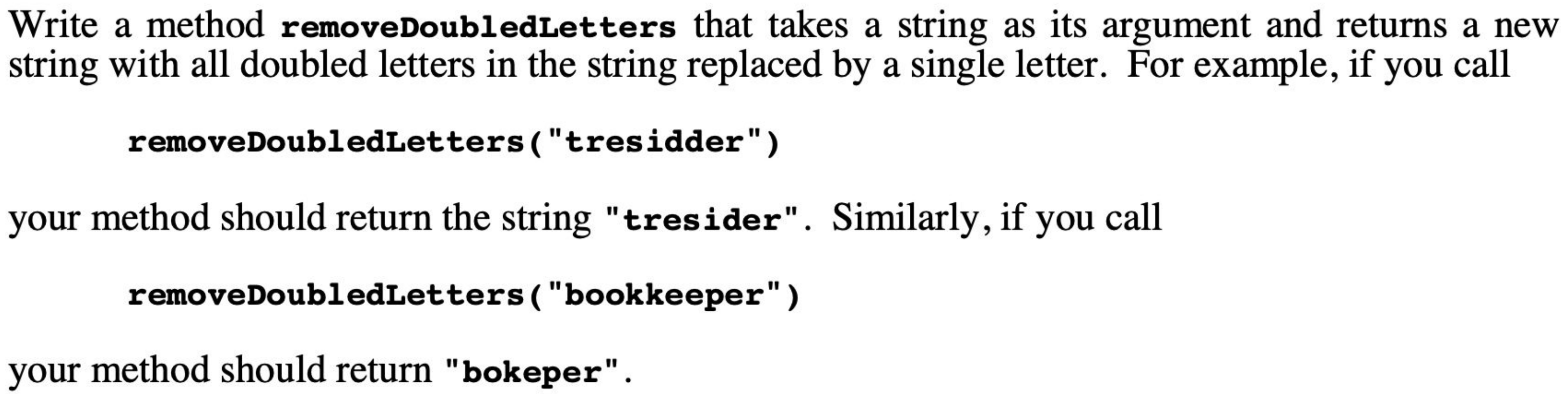
There’s actually a cool reason! Come and chat about it in office hours!

String Tokenizers

Key idea: Strings can be viewed as collections of whitespace-separated tokens

```
private void printTokens(String str){
     StringTokenizer t = new StringTokenizer(str);
     while (t.hasMoreTokens()){
          println("Next token: " + t.nextToken());
     }

}
```

A final problem



Write a method **`removeDoubledLetters`** that takes a string as its argument and returns a new string with all doubled letters in the string replaced by a single letter. For example, if you call

```
removeDoubledLetters("tresidder")
```

your method should return the string **`"tresider"`**. Similarly, if you call

```
removeDoubledLetters("bookkeeper")
```

your method should return **`"bokeper"`**.

Questions I’d ask myself

What do I do with each character?

Questions I’d ask myself

What do I do with each character?

If it isn’t the same as the last character, I add it to the result string

Questions I'd ask myself

What do I do with each character?

If it isn't the same as the last character, I add it to the result string

How do I get the last character?

Questions I’d ask myself

What do I do with each character?

If it isn’t the same as the last character, I add it to the result string

How do I get the last character?

I go to the index before my current one

Questions I’d ask myself

What do I do with each character?

> *If it isn’t the same as the last character, I add it to the result string*

How do I get the last character?

> *I go to the index before my current one*

Is there anything else I’d need to think about?

Questions I’d ask myself

What do I do with each character?

If it isn’t the same as the last character, I add it to the result string

How do I get the last character?

I go to the index before my current one

Is there anything else I’d need to think about?

The character at index 0 doesn’t have a character before it but needs to go into the string

The solution

```
private String removeDoubledLetters(String str) {




}
```

The solution

```
private String removeDoubledLetters(String str) {
     String result = "";
     for (int i = 0; i < str.length(); i++) {
          char ch = str.charAt(i);



     }
     return result;
}
```

The solution

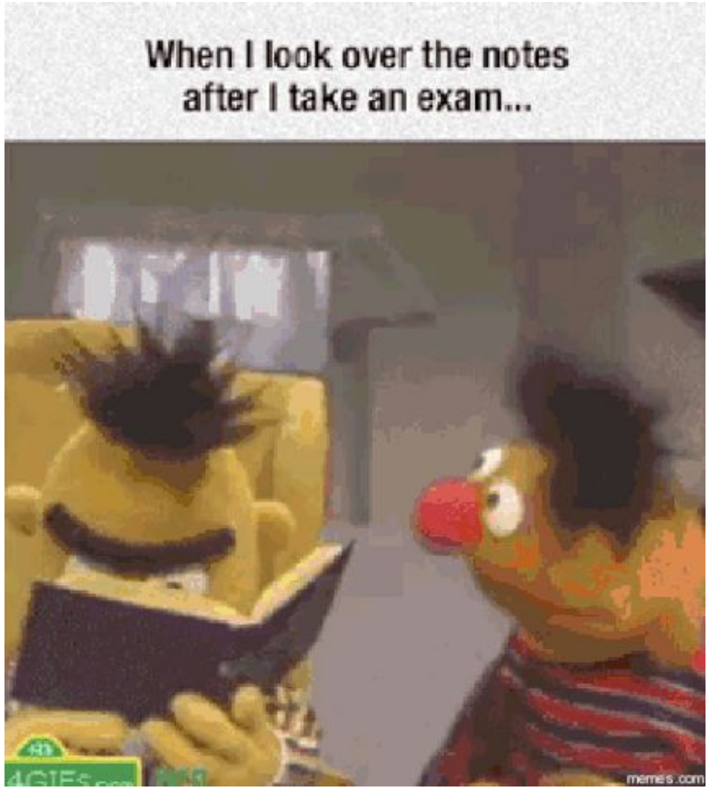
```java
private String removeDoubledLetters(String str) {
     String result = "";
     for (int i = 0; i < str.length(); i++) {
          char ch = str.charAt(i);
          if (ch != str.charAt(i - 1)) {
               result += ch;
          }
     }
     return result;
}
```

The solution

```java
private String removeDoubledLetters(String str) {
     String result = "";
     for (int i = 0; i < str.length(); i++) {
          char ch = str.charAt(i);
          if (i == 0 || ch != str.charAt(i - 1)) {
               result += ch;
          }
     }
     return result;
}
```

Exam Strategies

My main advice: understand, don't memorize



Decompose as you write your code

Try to attempt every problem, even if you’re not sure how to finish it off.

If you’re not sure about something, ask questions!

Try not to rely too much on your notes and books

Compile a quick reference sheet

Don’t panic!

Good luck!