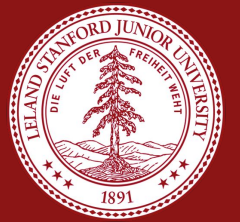
CS106A Midterm Review

CS106A, Summer 2019

Aditya Chander && Ryan Cao
Slides mostly from Sarai Gould && Laura Cruz-Albrecht



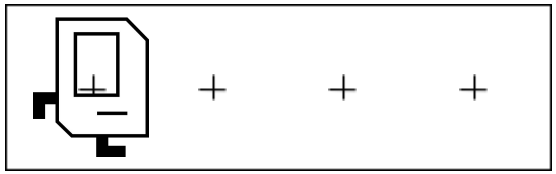
Midterm Logistics

- When and where?
  - Monday 22 July, 7-9pm (unless otherwise arranged with the lecturers)
  - Bishop Auditorium
- Bluebook
  - Download from CS106A website
  - Try it out before the exam (practice midterms available)
- What to bring
  - Laptop and charger
  - 2 double-sided 8.5" x 11" sheets of physical notes **(no notes on your laptop)**
  - Your two-step authentication device (probably your phone)
  - Make sure you download the encrypted midterm onto Bluebook before the exam!!!

Topics

- Karel
  - Control flow
  - Decomposition
  - Methods
- Introductory Java
  - Variables, expressions
  - Booleans, control flow
  - Methods, parameters, scope
- Graphics
  - Shapes
  - Animation and randomness
  - Mouse events
  - Instance variables
- Computer memory
  - Passing by value vs. reference
  - Objects vs. primitives
- Text processing
  - Characters and strings
  - Useful methods
  - File reading

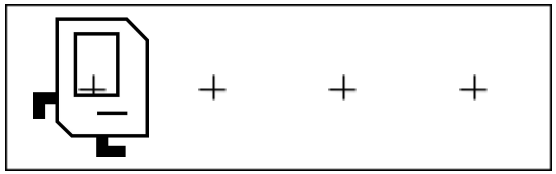
Review: A Method



A **method** is a set of new instructions we've created!

```
/* Comment describing method */
private void nameOfMethod(){

     // command 1
     // command 2

}
```

Review: For Loops

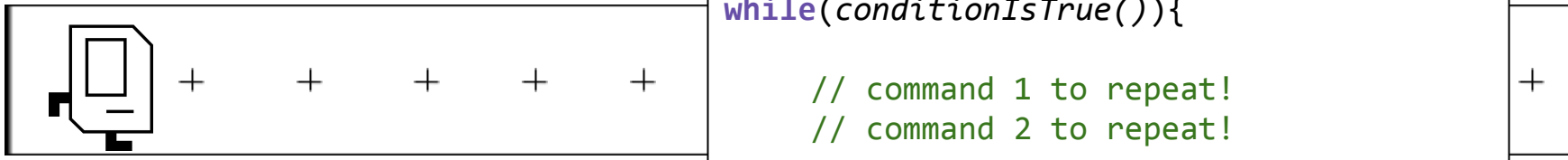


For Loops repeat something a *specific number of times!*

```
for(int i = 0; i < num_times; i++){

    // command 1 to repeat!
    // command 2 to repeat!

}

// code out here is NOT repeated!
```

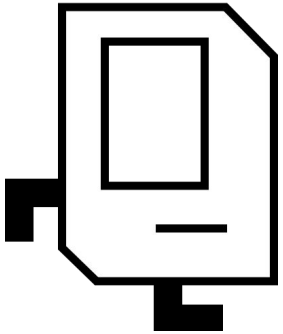
Review: While Loops

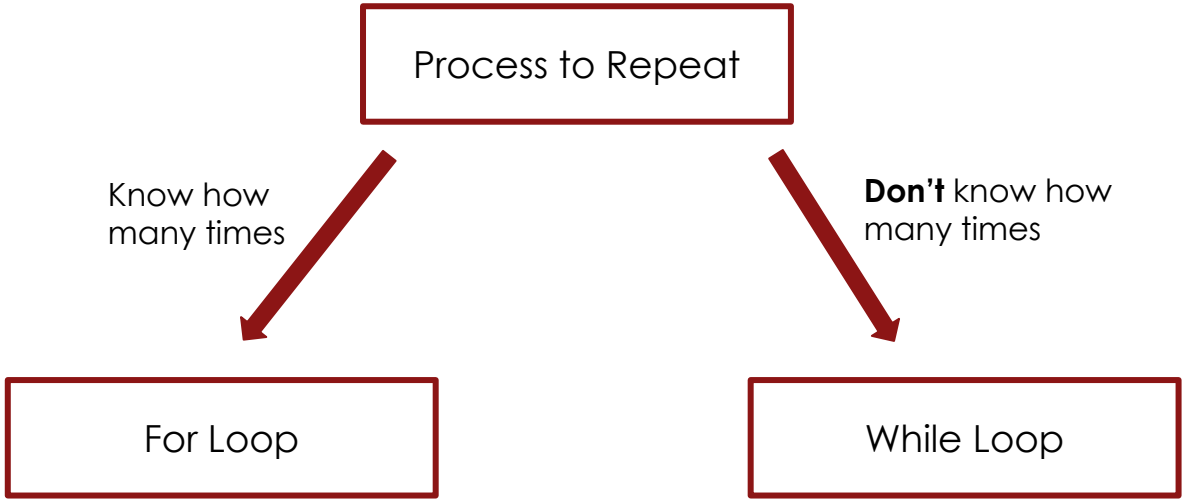
`While Loops` repeat *until a condition is no longer met!*



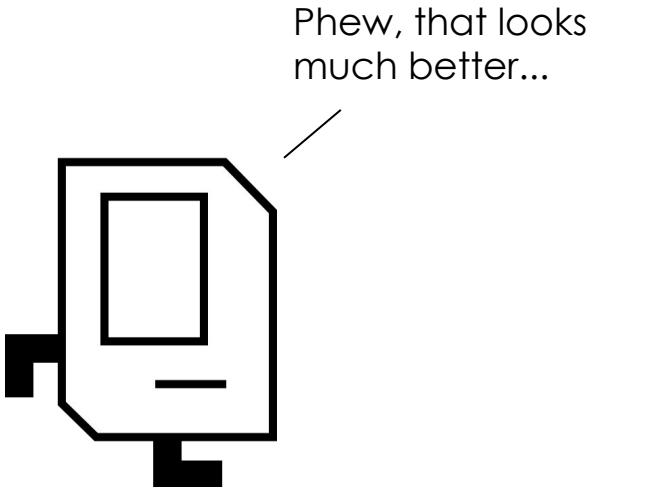
```
while(conditionIsTrue()){

    // command 1 to repeat!
    // command 2 to repeat!

}

// the condition isn't true anymore,
// so our code exited the while loop.
// code out here is NOT repeated!
```

While Loop or For Loop?





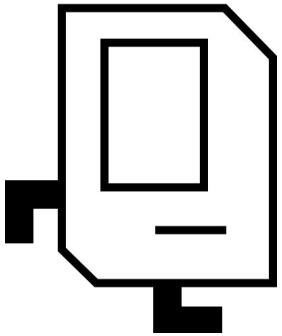
Review: If Statements



`If statements` check a condition *once*!

```
if(conditionIsTrue()){

     // command 1!
     // command 2!

}

// we have left the if statement.
// code out here happens no matter what!
```

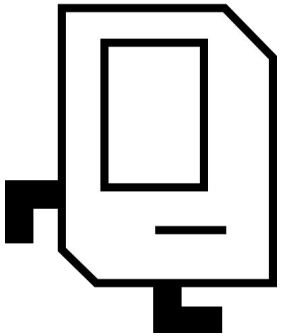
Review: If-Else



The **`else`** code only runs if the condition is *not* true!

```
if(conditionIsTrue()){

     // command 1 if conditionIsTrue()!
     // command 2 if conditionIsTrue()!

}  else {

     // command 1 if conditionIsFalse()!
     // command 2 if conditionIsFalse()!

}
// code out here happens no matter what!
```

Review: If-Else if-Else



The **`else if`** code allows you to put extra conditions.

```
if(conditionIsTrue()){
     …

}  else if(otherConditionIsTrue()) {
     …

}  else {
     …

}
// code out here happens no matter what!
```

What Conditions Can Karel Check?

Test	*Opposite*	*What it checks*
`frontIsClear()`	`frontIsBlocked()`	Is there a wall in front of Karel?
`leftIsClear()`	`leftIsBlocked()`	Is there a wall to Karel's left?
`rightIsClear()`	`rightIsBlocked()`	Is there a wall to Karel's right?
`beepersPresent()`	`noBeepersPresent()`	Are there beepers on this corner?
`beepersInBag()`	`noBeepersInBag()`	Any there beepers in Karel's bag?
`facingNorth()`	`notFacingNorth()`	Is Karel facing north?
`facingEast()`	`notFacingEast()`	Is Karel facing east?
`facingSouth()`	`notFacingSouth()`	Is Karel facing south?
`facingWest()`	`notFacingWest()`	Is Karel facing west?

Combining Conditions: && and | |

|| means "or". **One or both conditions** can be true!

```
if(condAIsTrue() || condBIsTrue()){

    // This code will run if one or
    // both of the conditions is true!

}
```

&& means "and". **Both conditions** must be true!

```
if(condAIsTrue() && condBIsTrue()){

    // This code will only run if both
    // of the conditions are true!

}
```

What is the boolean value of this expression? Let a = true; b = false; c = true;

!a || c && b

Combining Conditions: && and | |

|| means "or". **One or both conditions** can be true!

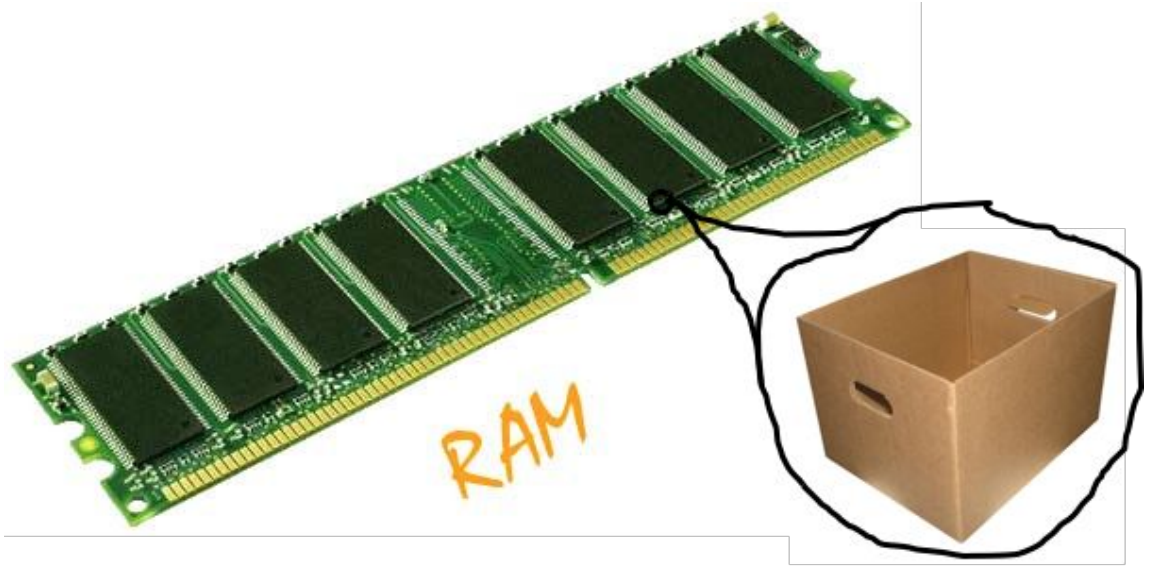
```
if(condAIsTrue() || condBIsTrue()){

    // This code will run if one or
    // both of the conditions is true!

}
```

&& means "and". **Both conditions** must be true!

```
if(condAIsTrue() && condBIsTrue()){

    // This code will only run if both
    // of the conditions are true!

}
```

What is the boolean value of this expression? Let a = true; b = false; c = true;
!a || c && b --> false || false --> false

Variables = Boxes



* my computer has space for about 2 billion boxes

Variable Types

```
// an integer number
int num = 5;

// a real (decimal) number
double fraction = 0.2;

// true or false
boolean isSummer = true;
```

Variable Types

```
// a single character
char letter = 'c';

// a "string" of text
String phrase = "Hi!";
String alsoAString = "5"; // not an int!
```

Create, Modify, Use

```
// Create a variable of type int, called coffees
// with the value 2
int coffees = 2;

// Modify coffees to be 1 greater (did someone
// just drink another coffee...)
coffees = coffees + 1;

// Use the value in coffees (output it)
println("I had " + coffees + " today.");
```

Expressions

- You can combine literals or variables together into ***expressions*** using *binary operators*:

+	Addition	*	Multiplication
-	Subtraction	/	Division
		%	Remainder

Type Interactions

`int` and `int` results in an `int`	`1 / 2`	→ `0`
`int` and `double` results in a `double`	`1 / 2.0`	→ `0.5`
`double` and `double` results in a `double`	`4.4 * 0.5`	→ `2.2`
`String` and `int` results in a `String`	`"abc" + 3`	→ `"abc3"`

* operations return the most expressive type

`String` > `double` > `int` > `char` > `boolean`

Practice with Type Interactions!

Try your hand at these --

```
> (5 + 3 / 4) / 10.0
> "CS" + 1 + (0 + 6) + "A"
> (char)('d' + 4)
```

Review: Shorthand Operators

Shorthand:

```
// +, -, /, *, % any value
variable += value;
x -= 3;
y /= 5;
z *= someValue;
k %= 10;

// add or subtract exactly 1
x++;
y--;
```

Equivalent Longer Version:

```
variable = variable + value;
x = x - 3;
y = y / 5;
z = z * someValue;
k = k % 10;


x = x + 1;
y = y - 1;
```

Review: Constants

```
public class ReceiptForFive extends ConsoleProgram {
    private static final double TAX_RATE = 0.08;
    private static final double TIP_RATE = 0.2;
    public void run() {
        double subtotal1 = readDouble("Meal cost? $");
        double tax1 = subtotal * TAX_RATE;
        double tip1 = subtotal * TIP_RATE;
        double total1 = subtotal1 + tax1 + tip1;
        println("Total for person 1: $" + total1);
        ...
        double subtotal5 = readDouble("Meal cost? $");
        double tax5 = subtotal * TAX_RATE;
        double tip5 = subtotal * TIP_RATE;
        double total5 = subtotal1 + tax1 + tip1;
        println("Total for person 5: $" + total5);
    }
}
```

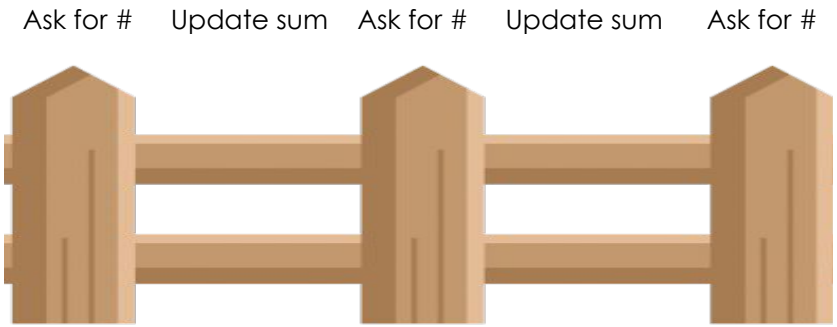
much better

Review: Relational Operators

Operator	Meaning	Example	Value
==	equals	`1 + 1 == 2`	`true`
!=	does not equal	`3.2 != 2.5`	`true`
<	less than	`10 < 5`	`false`
>	greater than	`10 > 5`	`true`
<=	less than or equal to	`126 <= 100`	`false`
>=	greater than or equal to	`5.0 >= 5.0`	`true`

* all have equal precedence

Review: Sentinel Loops




```
// fencepost problem!
// ask for number - post
// add number to sum - fence

int sum = 0;
int num = readInt("Enter a number: ");
while (num != -1) {
    sum += num;
    num = readInt("Enter a number: ");
}
println("Sum is " + sum);
```

Review: Using the For Loop Variable

```
// Adds up the numbers 1 through 42
int sum = 0;
for (int i = 1; i <= 42; i++) {
    sum += i;
}
println("Sum is " + sum);
```

Methods

A **method** is a set of new instructions we've created!

Example of a comment explaining Pre and Post conditions.

```
/* Method description.
* Pre: What we assume is true beforehand.
* Post: What we promise is true afterwards.
*/
private void nameOfMethod(){

     // commands go in here!

}
```

Review: Parameters and Return Statements

```
/* We will do x, y, and z actions and return methodType data
* Pre: What we assume is true beforehand.
* Post: What we promise is true afterwards.
*/
visibility methodType methodName(paramType1 paramName1, paramType2 paramName2){

     commands;

     return dataThatIsMethodType;
}
```

Review: Making Sandwiches

```
/* We will make Java a sandwich.
* Pre: n/a
* Post: We will have a completed sandwich.
*/
private Food makeASandwich(Food bread, Food veg, Food protein, Food spread){
     Food sandwich = bread + veg + protein + spread + bread;

     return sandwich;
}
```

Review: Return Statements

```
public void run(){
     double average = calculateAverage(5, 10);
     println("The average is: " + average);
}

private double calculateAverage(double num1, double num2){
     println("I can also do actions!");
     println("I also HAVE to return a double because that's the method type!");

     double sum = num1 + num2;
     return sum / 2; // This will equal 7.5
}
```

7.5

calculateAverage(5, 10);

ParameterReturnExample [completed]

```
I can also do actions!
I also HAVE to return a double because that's the method type!
The average is 7.5
```

Review: Return Statements

```
public void run(){
     double average = 7.5;
     println("The average is: " + average);
}

private double calculateAverage(double num1, double num2){
     println("I can also do actions!");
     println("I also HAVE to return a double because that's the method type!");

     double sum = num1 + num2;
     return sum / 2; // This will equal 7.5
}
```

7.5

average

ParameterReturnExample [completed]

```
I can also do actions!
I also HAVE to return a double because that's the method type!
The average is 7.5
```

Review: Scope

Scope is the idea that variables only exist inside a certain block of code.

In Java, a variable is **born when it is declared.**

A variable **terminates when it hits the ending bracket** of the code block in which it was declared.

```
public void run(){
     int newVariable = 0; // I am born!
     ...
     ...
     ...
} // I hope I haven't bored you. Goodbye.
```

A Variable Love Story: Scope

The variable only exists from its declaration to the end of its current code block.

```
while (conditionAIsTrue){

     if (conditionBIsTrue){
          ...
          int newVariable = 0; // I am born!
          ...

     } // I'm bored with it all. Goodbye.
     println(newVariable);
}
```

A Variable Love Story: Scope

It doesn't exist before it's declared, and it doesn't exist outside of its current code block!

```
while (conditionAIsTrue){

     if (conditionBIsTrue){
          ...
          int newVariable = 0; // I am born!
          ...

     } // I'm bored with it all. Goodbye.
     println(newVariable);
}
```

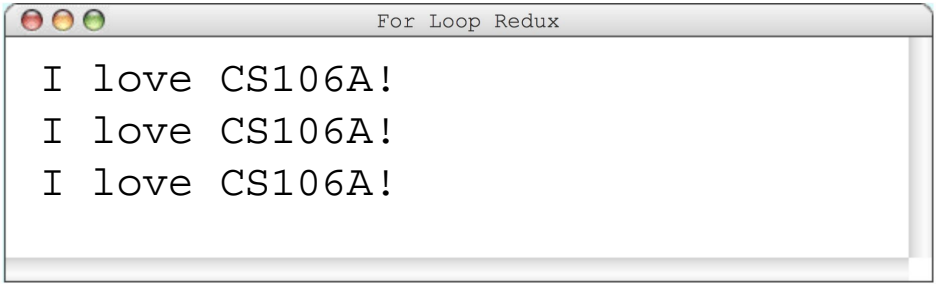
Review: For Loop

This code is run once, just before the for loop starts

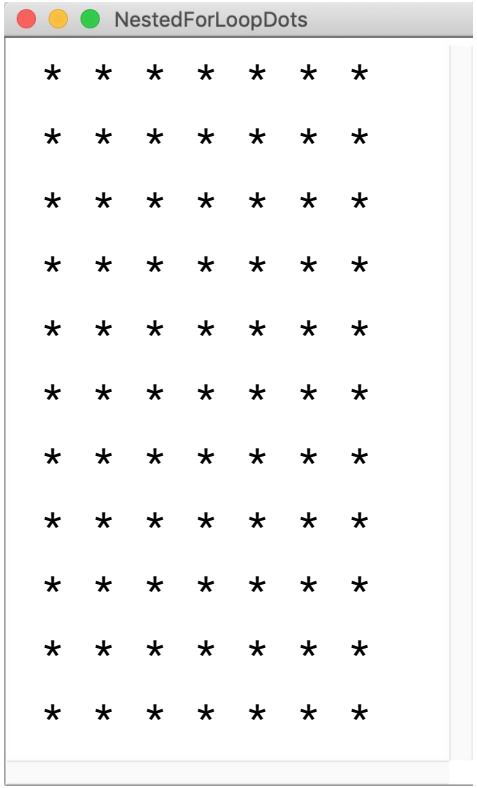
Repeats the loop if this condition passes

This code is run each time the code gets to the end of the "body"

```
for (int i = 0; i < 3; i++) {
   println("I love CS106A!");
}
```



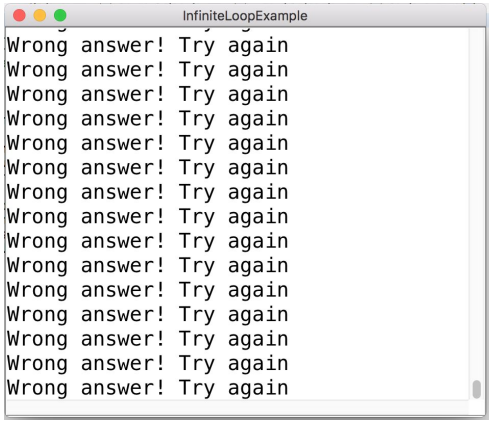
Review: Nested For Loops




```
for (int i = 0; i < NUM_ROWS; i++) {
    for (int j = 0; j < NUM_COLS; j++) {
        print("*");
    }
    println();
}
```

Review: Infinite Loops

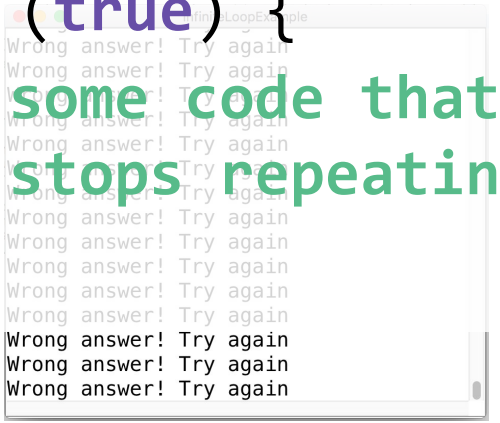
```
int answer = readInt("What is 1 + 1?");

while (answer != 2){
     println("Wrong answer! Try again");
}
```



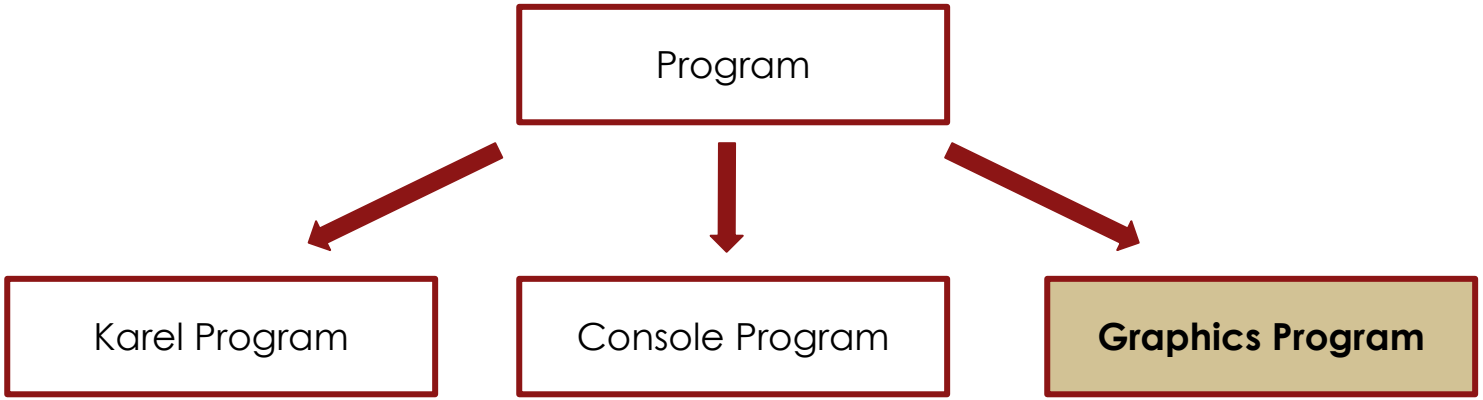
Review: Infinite Loops

```
int answer = readInt("What is 1 + 1?");

while (answer != 2){
     println("Wrong answer! Try again");
}
```

```
while (true) {
    // some code that never
    // stops repeating ...
}
```

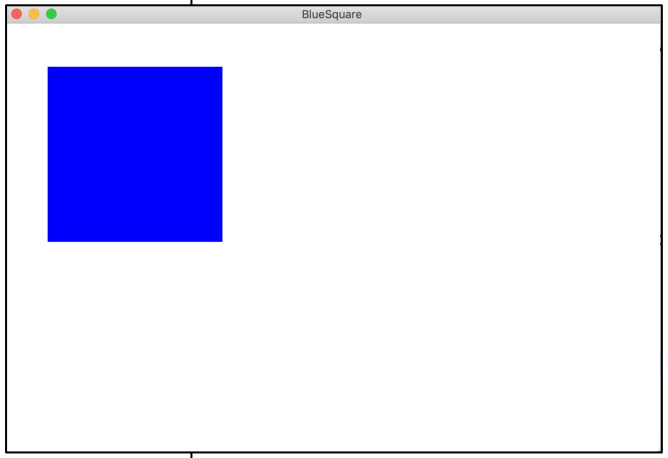


Review: Intro to Graphics Programs



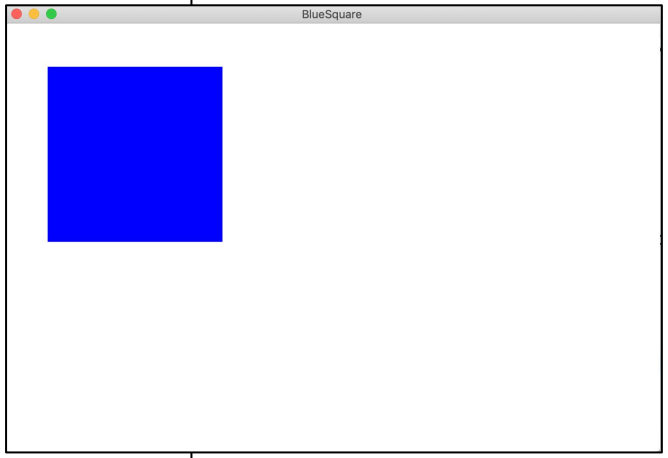
Review: Our First Graphics Program

```
import acm.program.*;
import acm.graphics.*; // Stanford graphical objects
import java.awt.*;     // Java graphical objects

public class BlueSquare extends GraphicsProgram {
     public void run(){
          drawBlueSquare();
     }

     private void drawBlueSquare(){
          GRect rect = new GRect(200, 200);
          rect.setFilled(true);
          rect.setColor(Color.BLUE);
          add(rect, 50, 50);
     }
}
```

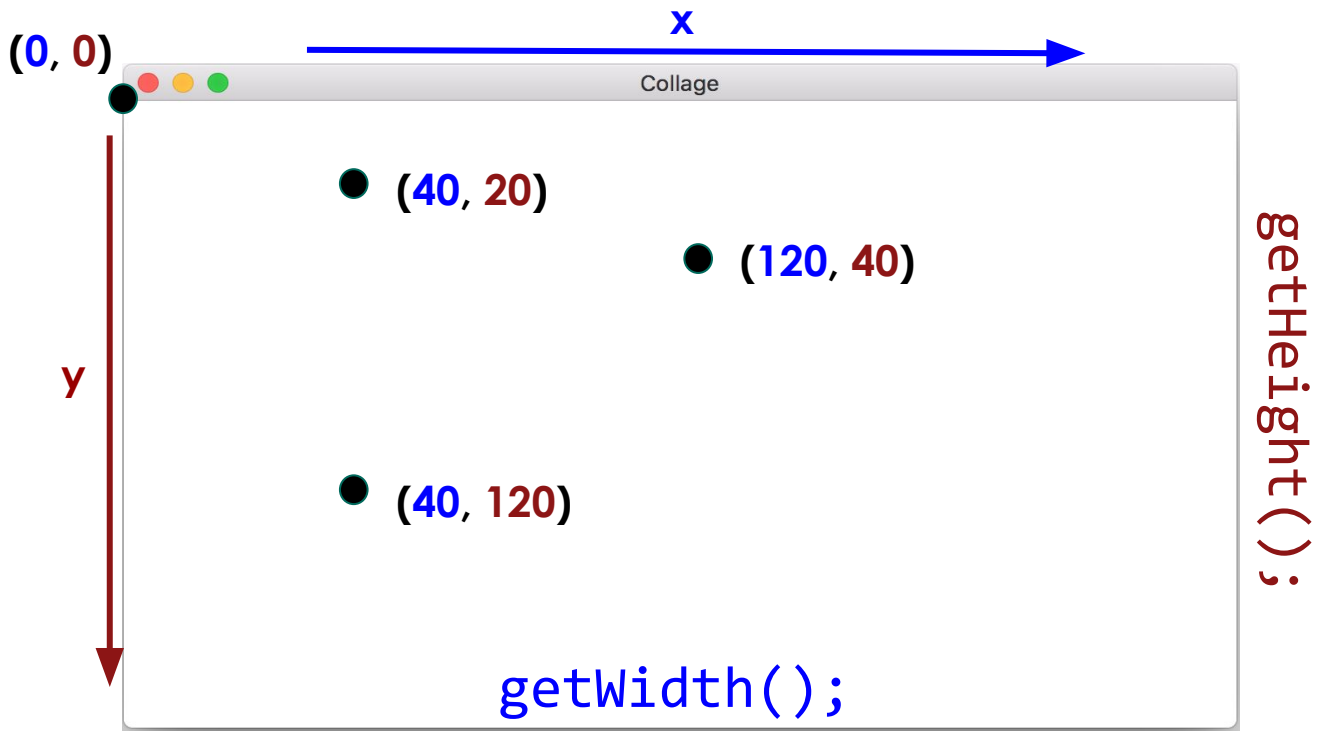


Review: Our First Graphics Program

```
import acm.program.*;
import acm.graphics.*; // Stanford graphical objects
import java.awt.*;     // Java graphical objects

public class BlueSquare extends GraphicsProgram {
     public void run(){
          drawBlueSquare();
     }

     private void drawBlueSquare(){
          GRect rect = new GRect(200, 200);
          rect.setFilled(true);
          rect.setColor(Color.BLUE);
          add(rect, 50, 50);
     }
}
```



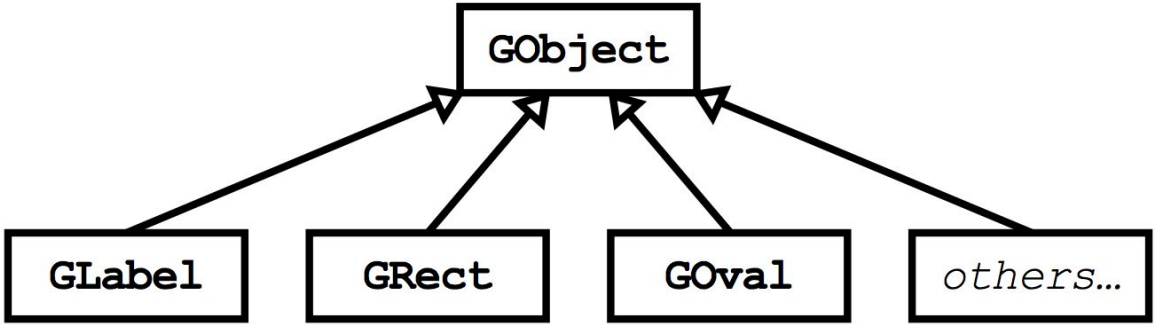
Review: The Graphics Canvas



* Note: The y coordinate gets *bigger* as we go *down*. This is opposite of how it acts in most math classes!

Review: Working with Graphics

We make graphics programs by *creating* ***graphics objects*** and *manipulating their properties*.



Review: Working with Graphics

- We can ***create*** graphics objects using the `new` keyword:

```
GRect rect = new GRect(100, 200);
```

- We can ***manipulate*** graphics objects by calling methods on those objects:

```
rect.setColor(Color.BLUE);
```

who? what? what specifically?

Review: Animation Loop

```
public void run() {
    // setup
    Make variables. Add graphics to canvas.

    while (condition) {
        // update world
        Update graphics.

        // pause
        pause(milliseconds);
    }
}
```

Review: Animation Loop

```
public void run() {
    // setup
    GRect square = makeSquare();

    while (true) {
        // update world
        square.move(1, 0);

        // pause
        pause(PAUSE_TIME);
    }
}
```

Review: RandomGenerator

```
// this variable can generate random values
RandomGenerator rgen = RandomGenerator.getInstance();

// make a random number between 1 and 6 inclusive
int diceRoll = rgen.nextInt(1, 6);

// also: nextDouble, nextBoolean, nextColor, etc
```

Review: Accessing the Canvas

- It is possible to determine what, if anything, is at the canvas at a particular point.
- The method:

```
GObject getElementAt(double x, double y);
```

  returns which object is at the given location on the canvas.

- The return type is `GObject`, since we don't know what specific type (`GRect`, `GOval`, etc.) is really there.
- If no object is present, the special value `null` is returned.

Review: Null

`null` is a special variable value that objects can have that means "nothing". Primitives cannot be null.

If a method returns an object, it can return `null` to signify "nothing". (just say `return null;`)

```
// may be a GObject, or null if nothing at (x, y)
GObject maybeAnObject = getElementAt(x, y);
```

Objects have the value `null` before being initialized.

```
GOval circle;   // initially null
```

Review: Null

You can check if something is `null` using `==` and `!=`

```
// may be a GObject, or null if nothing at (x, y)
GObject maybeAnObject = getElementAt(x, y);
if (maybeAnObject != null) {
    // do something with maybeAnObject
} else {
    // null - nothing at that location
}
```

Review: Null

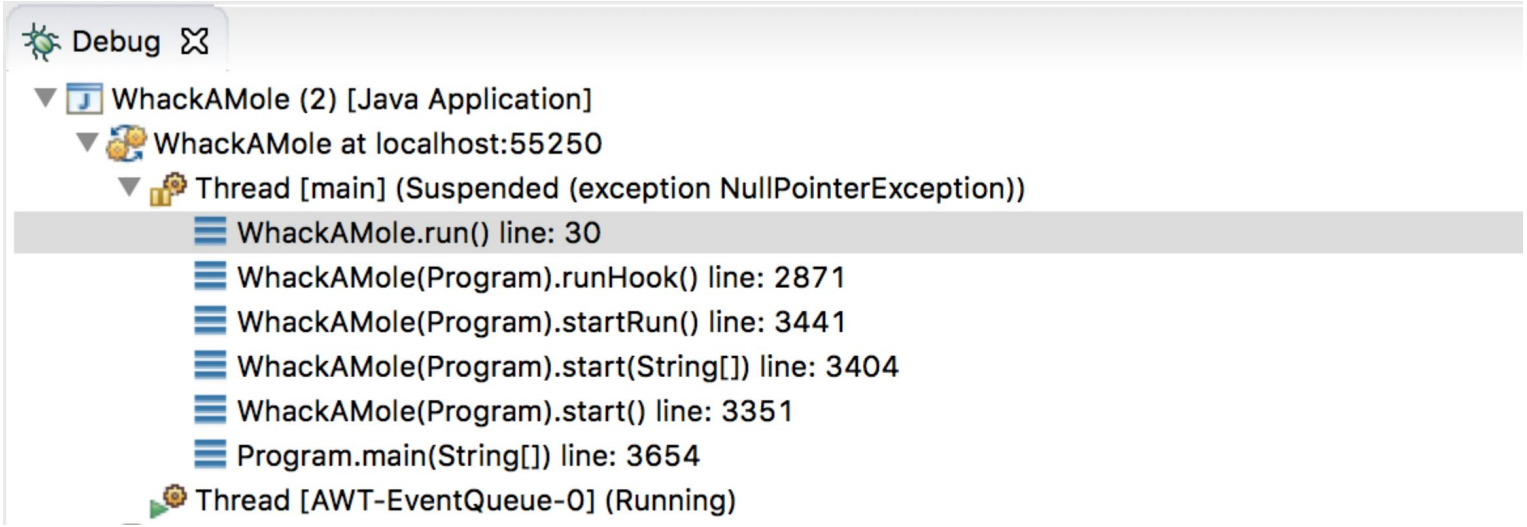
Calling methods on an object that is `null` will crash your program!

```
// may be a GObject, or null if nothing at (x, y)
GObject maybeAnObject = getElementAt(x, y);
if (maybeAnObject != null) {
    int x = maybeAnObject.getX();  // OK
} else {
    int x = maybeAnObject.getX();  // CRASH!
}
```

Review: Null

Calling methods on an object that is `null` will crash your program!

⇒ Throws a `NullPointerException`



Review: Events

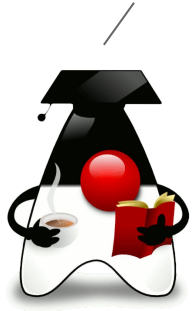
- An ***event*** is some external stimulus that your program can respond to.

- Common events include:
  - Mouse motion / clicking.
  - Keyboard buttons pressed.
  - Timers expiring.
  - Network data available.

Review: Events

```
public void run() {
    // Java runs this when program launches
}

public void mouseClicked(MouseEvent event) {
    // Java runs this when mouse is clicked
}

public void mouseMoved(MouseEvent event) {
    // Java runs this when mouse is moved
}
```

To **respond** to events, your program must write methods to **handle** those events.



Review: Types of Mouse Events

- There are many different types of mouse events!
- Each takes the form:

```
public void eventMethodName(MouseEvent e) { ...
```

Method	Description
mouseMoved	mouse cursor moves
mouseDragged	mouse cursor moves while button is held down
mousePressed	mouse button is pressed down
mouseReleased	mouse button is lifted up
mouseClicked	mouse button is pressed and then released
mouseEntered	mouse cursor enters your program's window
mouseExited	mouse cursor leaves your program's window

Review: MouseEvent Objects

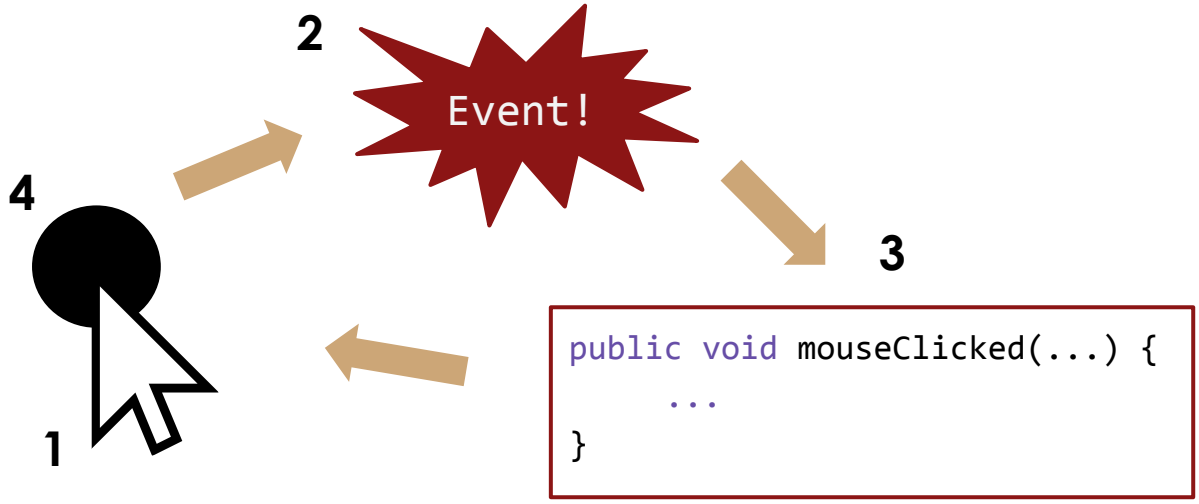
```
public void mouseClicked(MouseEvent e) { ...
```

- A `MouseEvent` contains information about the event that just occurred:

Method	Description
`e.getX()`	the *x*-coordinate of mouse cursor in the window
`e.getY()`	the *y*-coordinate of mouse cursor in the window

Review: Events

1. User performs some action, like moving / clicking the mouse.
2. This causes an event to occur!
3. Java executes a particular method to handle the event.
4. That method's code updates the screen appearance in some way



Review: Instance Variables

1. Variables exist until their inner-most control block ends.
2. If a variable is *defined outside all methods*, its inner-most control block is the entire program!
3. We call these variables ***instance variables***.

```
private type name;     // declared outside any method!
```

```
private GRect square = null;

public void run() {
    square = new GRect(...);
    GRect localSquare = new GRect(...);
}
```

Review: Instance Variables + Events

Often you need instance variables to pass information between the run method and the mouse event methods.

```
/* Instance variable for the square to be tracked */
private GRect square = null;

public void run() {
     square = makeSquare();
     addSquareToCenter();
}

public void mouseMoved(MouseEvent e) {
     double x = e.getX() - SQUARE_SIZE / 2.0;
     double y = e.getY() - SQUARE_SIZE / 2.0;
     square.setLocation(x, y);
}
```

Review: The Importance of Style

- It is considered extremely poor style to use instance variables unnecessarily:

Do not use instance variables where local variables, parameters, and return values suffice.

- Use *local variables* for temporary information.
- Use *parameters* to communicate data into a method.
- Use *return values* to communicate data out of a method.

Pass by Reference vs Value

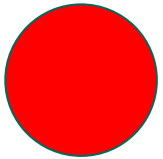
Pass by Reference

Objects are passed by **reference.**

A few examples:

`GRect`

`GImage`



`GOval`

Pass by Value

Primitives are passed by **value**.

A few examples:

`int`

`char`

`double`

`boolean`

Pass by Reference vs Value

What does this mean?

When you pass something by **reference**, you are passing an address to that thing (e.g. giving your phone number -- you wouldn't hand your actual phone to them!)

If something is passed by ***reference***, it ***can*** be altered simply by passing it into a method.

If something is passed by ***value***, it ***cannot*** be altered simply by passing it into a method.

Review: Memory

Primitives

passed by *value*

stored on the ***stack***

Objects

passed by *reference*

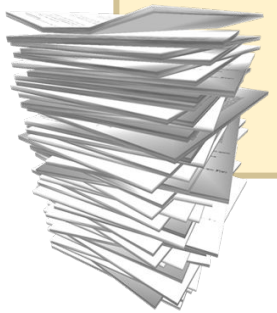
stored on the ***heap***,
referred to from the ***stack***

Review: Stack vs. Heap

The **Stack** is more temporary memory. Things on the Stack are added and removed as methods are called and returned.

The **Heap** is more permanent memory. Things on the Heap don't disappear as methods are called or returned.

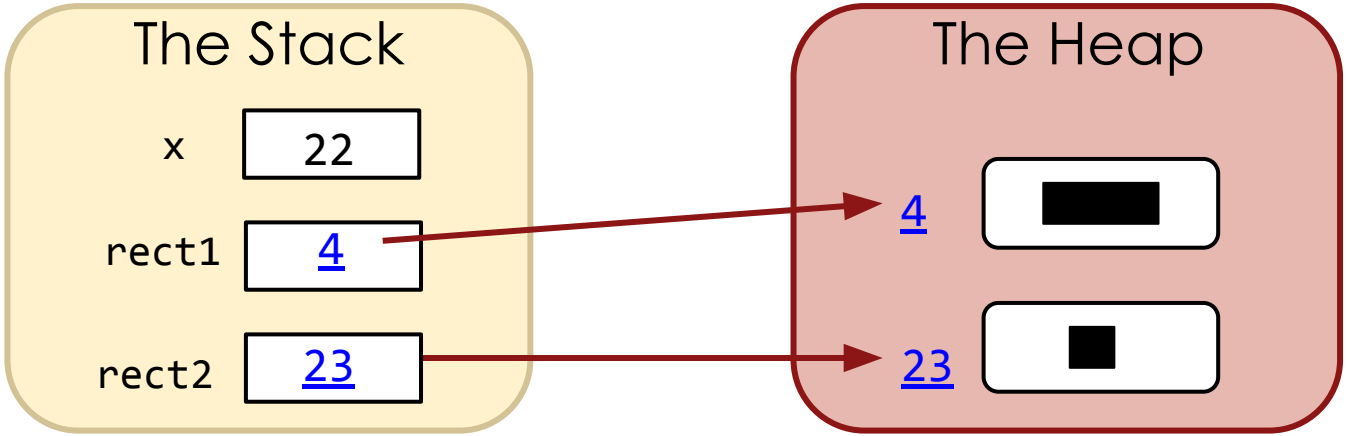
The Stack



The Heap

Review: Stack vs. Heap

```
int x = 22;
GRect rect1 = new GRect(30, 10);
GRect rect2 = new GRect(10, 10);
```



== compares values in the Stack

`.equals()` compares objects on the Heap

Review: Are They Equal? #1

```
public void run(){

     GRect rect1 = new GRect(100, 100); // rect1 = location 4
     GRect rect2 = new GRect(100, 100); // rect2 = location 5

     // is the memory address of rect1 equal to the memory address of rect2?
     if(rect1 == rect2){
          println("These rectangles are equal!");
     } else {
          println("Actually, these rectangles are not equal.");
     }
}
```

Review: Are They Equal? #1

```
public void run(){

     GRect rect1 = new GRect(100, 100); // rect1 = location 4
     GRect rect2 = new GRect(100, 100); // rect2 = location 5

     // is the memory address of rect1 equal to the memory address of rect2?
     if(rect1 == rect2){
          println("These rectangles are equal!");
     } else {
          println("Actually, these rectangles are not equal.");
     }
}
```

We cannot create two different GRects in the same location in memory. The two GRects have different addresses, and are thus not ==.

Review: Are They Equal? #2

```
public void run(){

    GRect rect3 = new GRect(100, 100);

    if(rect3 == rect3){
        println("This rectangle is equal to itself!");
    } else {
        println("Actually, this rectangle is not equal to itself.");
    }
}
```

Review: Are They Equal? #2

```
public void run(){

    GRect rect3 = new GRect(100, 100);

    if(rect3 == rect3){
        println("This rectangle is equal to itself!");
    } else {
        println("Actually, this rectangle is not equal to itself.");
    }
}
```

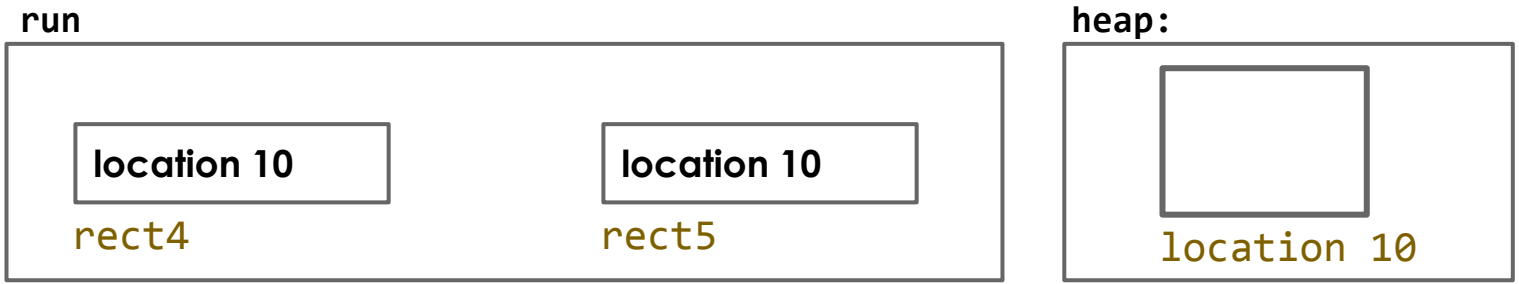
Remember: This will only evaluate to true if it's the exact same rectangle!

Review: Are They Equal? #3

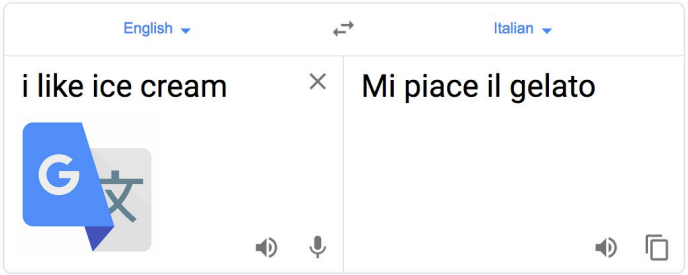
```
public void run(){

      GRect rect4 = new GRect(100, 100); // rect4 = location 10
      GRect rect5 = rect4;               // rect5 also stores location 10

      // is the memory address of rect4 equal to the memory address of rect4?
      if(rect4 == rect5){
            println("These rectangles are equal!");
      } else {
            println("Actually, these rectangles are not equal.");
      }
}
```

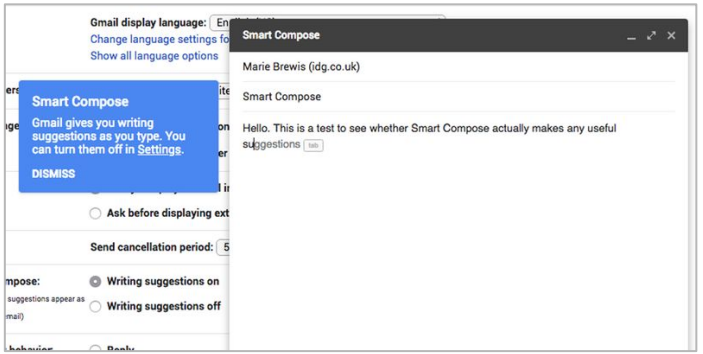
Review: Are They Equal? #3

```
public void run(){

      GRect rect4 = new GRect(100, 100); // rect4 = location 10
      GRect rect5 = rect4;               // rect5 also stores location 10

      // is the memory address of rect4 equal to the memory address of rect4?
      if(rect4 == rect5){
            println("These rectangles are equal!");
      } else {
            println("Actually, these rectangles are not equal.");
      }
}
```



Review: Text Processing





Review: Characters

A **char** is a variable type that represents a single character or "glyph".

Note that **char** is a primitive data type!

Review: Char

Under the hood, Java represents each **char** as an *integer*. This integer is its "ASCII" value.

Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char
32	[space]	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	[backspace]

Review: Char

Under the hood, Java represents each **char** as an *integer*. This integer is its "ASCII" value.

```
char uppercaseA = 'A';     // Actually 65
char lowercaseA = 'a';     // Actually 97
char zeroDigit = '0';      // Actually 48
```

Review: Char

Under the hood, Java represents each **char** as an *integer*. This integer is its "ASCII" value.

```
char uppercaseA = 'A';     // Actually 65
char lowercaseA = 'a';     // Actually 97
char zeroDigit = '0';      // Actually 48
```

- Uppercase letters (`'A' -> 'Z'`) are sequentially numbered
- Lowercase letters (`'a' -> 'z'`) are sequentially numbered
- Digits (`'0' -> '9'`) are sequentially numbered

Review: Char Math

We can take advantage of Java representing each `char` as an *integer* (its "ASCII" value).

```
boolean areEqual = 'A' == 'A';      // true
boolean earlierLetter = 'f' < 'c'; // false
char uppercaseB = 'A' + 1;          // 'B'
int diff = 'c' - 'a';               // 2

int alphabetSize = 'z' - 'a' + 1;
// or
int alphabetSize = 'Z' - 'A' + 1;
```

Review: Type-Casting

If we want to force Java to treat an expression as a particular type, we can also *cast it* to that type.

```
'A' + 1              // evaluates to 66 (int)
(char)('A' + 1)      // evaluates to 'B' (char)

1 / 2                // evaluates to 0 (int)
(double)1 / 2        // evaluates to 0.5 (double)
1 / (double)2        // evaluates to 0.5 (double)
```

Review: Char Loops

```
// prints the characters a to z
for (char ch = 'a'; ch <= 'z'; ch++) {
    println(ch);
}
```

Review: Character Methods

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

Remember: these return a new **`char`**, they cannot modify an existing **`char`**.

Review: Strings

Text is stored using the variable type `String`.
A `String` is a sequence of characters!

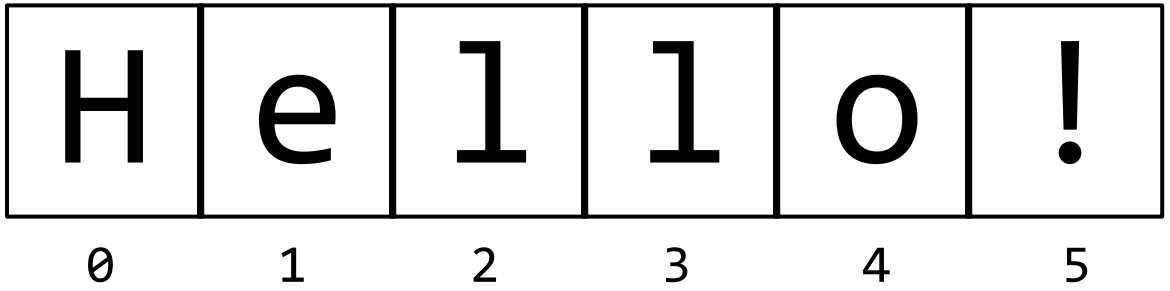
Strings are objects!

Double quotes!

```
String text = "Hello!";
```

Review: Strings

- Each character is assigned an index, going from 0 to length-1.
- There is a `char` at each index.




```
int strLen = text.length();             // 6
char last = text.charAt(strLen - 1);    // '!'
```

Review: Strings vs. Chars

Remember: `chars` and length-1 `Strings` are different!

```
char ch = 'A'   DIFFERENT FROM   String str = "A"
```

```
'A' + 1   // evaluates to 66 (int)
"A" + 1   // evaluates to "A1" (String)
```

Review: Creating Strings

```
String str = "Hello, world!";
String empty = "";

// Read in text from the user
String name = readLine("What is your name? ");

// String concatenation (using “+”)
String message = name + " is " + 2 + " cool.";
```

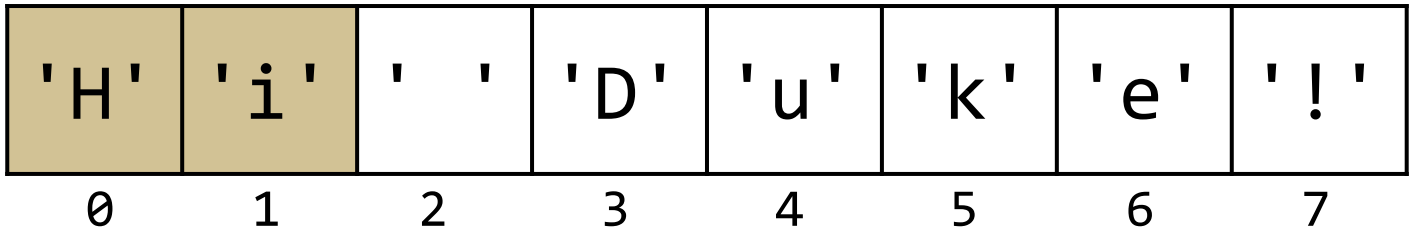
Review: From Chars to Strings

```
char c1 = 'a';
char c2 = 'b';

// How do we concatenate these characters?

String str = c1 + c2;          // ERROR: this is an int!

String str = "" + c1 + c2;     // ✔
```

Review: Substrings

A ***substring*** is a subset of a string.

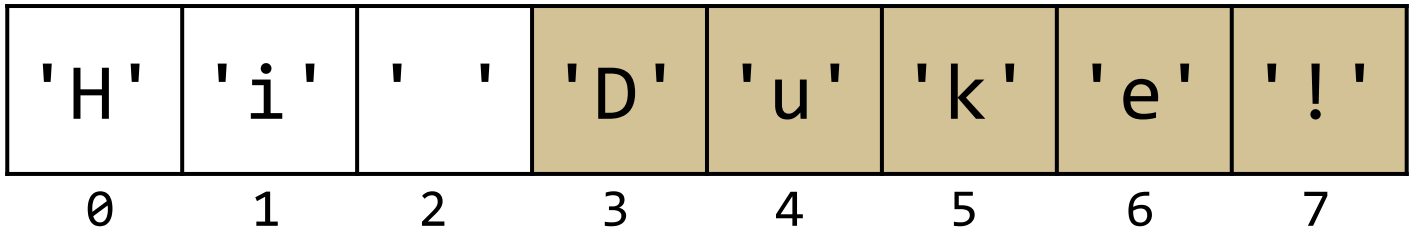
```
String str = “Hi Duke!”;
String hi = str.substring(0, 2);
```



Review: Substrings

A ***substring*** is a subset of a string.

```
String str = “Hi Duke!”;
String dukeExclm = str.substring(3);  // to end
```



Review: Useful String Methods

`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

* remember, called using **dot notation**: *`myString`*`.length()`

Review: Strings are Immutable

- Java strings are **immutable**: once you create a String, its contents cannot be changed.
- To change a String, you must create a *new* String containing the value you want (e.g. using String methods).

```
String typo = "Hello, warld!”;

typo.charAt(8) = ‘o’;   // Error! Will not run.

String corrected = typo.substring(0, 8) +
                   ‘o’ + typo.substring(9);
```

Review: Comparing Strings

Method	Description
`s1.equals(s2)`	whether two strings contain the same characters
`s1.equalsIgnoreCase(s2)`	whether two strings contain the same characters, ignoring upper vs. lower case
`s1.startsWith(s2)`	whether **s1** contains **s2**'s characters at start
`s1.endsWith(s2)`	whether **s1** contains **s2**'s characters at end
`s1.contains(s2)`	whether **s2** is found within **s1**

Always use `.equals()` instead of `==` and `!=`
to compare the characters in a string

Review: Looping over Strings

A common String programming pattern is looping over a String and operating on each character.

```
for (int i = 0; i < str.length(); i++) {
    char ch = str.charAt(i);
    // do something with ch here
}
```

Review: Looping over Strings

Another common String programming pattern is **building up a new string** by adding characters to it over time.

```
// Creates a new String in all caps
String str = "Hello!";
String newStr = "";
for (int i = 0; i < str.length(); i++) {
    char ch = str.charAt(i);
    newStr += Character.toUpperCase(ch);
}
println(newStr);  // HELLO!
```

Revew: File Reading

```
Are you there Diary? It's me, Duke.

I had a really tough day yesterday. Between stack
overflows and integer division, it felt like I got
nothing done! Hopefully today will be a better day!
```

We read and printed each line in Duke's Diary!

```
try {
      Scanner input = new Scanner(new File("Dukes Diary.txt"));
      while(input.hasNextLine()){
            String line = input.nextLine();
            println(line);
      }
      input.close();
} catch (IOException ex){
      println("The error is: " + ex);
}
```

One Word at a Time

```
123456 password qwerty iloveCS106A letmein
```

Reading through a file one word at a time

```
try {
      Scanner input = new Scanner(new File("Dukes Passwords.txt"));
      // Will print all passwords in the file!
      while(input.hasNext()){
            String password = input.next();
            println(password);
      }
      input.close();
} catch (IOException ex){
      println("The error is: " + ex);
}
```

Different Scanner Methods

Method	Description
`sc.nextLine()`	reads and returns a one-*line* `String` from the file
`sc.next()`	reads and returns a one-word `String` from the file
`sc.nextInt()`	reads and returns an `int` from the file
`sc.nextDouble()`	reads and returns a `double` from the file
`sc.hasNextLine()`	returns `true` if there are any more lines
`sc.hasNext()`	returns `true` if there are any more tokens
`sc.hasNextInt()`	returns `true` if there is a next token and it's an `int`
`sc.hasNextDouble()`	returns `true` if there is a next token and it's a `double`
`sc.close();`	should be called when done reading the file

Choosing Files

What if we want to pick the file we open?

```
// Gets filename from user
String filename = promptUserForFile("What file would you like to open?");

try {
      Scanner input = new Scanner(new File(filename));
      while(input.hasNextLine()){
            String line = input.nextLine();
            println(line);
      }
      input.close();
} catch (IOException ex){
      println("The error is: " + ex);
}
```

Note: we changed this to the variable `filename`.

How do I review for the midterm?

> Content review is fantastic, but you've got to apply your knowledge!

> Start early and systematically. Do old section problems! Under time conditions! Section problems are the best practice you can get for exams.

> Do the practice midterm! Don't look at the answer key for anything unless you've given it a good go first :)

> If you need even more practice problems, head over to https://www.codestepbystep.com/ and log in with your SUID

Questions

Any questions? :D