TODO: record lecture

Week03

Today is...
✨ No-Laptop Monday! ✨

- HW03 preview
- functions
- classes
- HW03 behind the scenes
- references

WARMUP
What is a bullet hell?

BONUS
Who wrote the game Everyday Shooter? What is she working on now?

BONUS
What does Touhou 7: Perfect Cherry Blossom's Extra Stage look like? Who wrote this?

233

HW03 preview

239

functions

240

anatomy of a function

241

anatomy of a function (1/2)

```
ReturnType functionName(ArgumentOneType argumentOne, ...) {
    ...
}
```

- a **function** is a lil chunk of code you can call from elsewhere
  - a function takes any number of **arguments**
    - `... foo(int arg) { ... } // function foo takes an int`
  - a function with a non-void **return type** must **return** a value of that type
    - `int bar(...) { ... } // bar returns an int`
    - `void baz(...) { ... } // baz doesn't return anything`

242

anatomy of a function (2/2)

```
void drawLine(
        double a_x,
        double a_y,
        double b_x,
        double b_y,
        Color color) {
    ...
}
```

243

return

244

return (1/2)

- a **return** statement stops execution of a function and returns the program to where the function was called
  - some return statements return a value
    - `return 123;`
  - others do not
    - `return;`
      - ✨ this can be used to stop running a void-returning function in the middle

245

return (2/2)

- a function with a non-void **return type** must **return** a value of that type, regardless of the path taken through the function

```
Error: missing return statement

static boolean isEven(int n) {
    if (n % 2  == 0) {
        return true;
    }
}
```

246

return (2/2)

- a function with a non-void **return type** must **return** a value of that type, regardless of the path taken through the function

```
static boolean isEven(int n) {
    if (n % 2  == 0) {
        return true;
    }
    return false;
}
```

247

return (3/2)

- a function with a non-void **return type** must **return** a value of that type, regardless of the path taken through the function

```
static boolean isEven(int n) {
    return (n % 2  == 0);
}
```

248

return (3/2)

- a function with a non-void **return type** must **return** a value of that type, regardless of the path taken through the function

249

void

void

- **void** is a special return type meaning a function does not return a value
  - void functions often modify (the objects referenced by) their arguments
    - `static void inPlaceReverse(int[] array) { ... }`
      - `// no need to return a reference to array`
        `// (user of the function already has one)`
  - in Java, the **main method** is a void function (it doesn't return anything)
    - `public static void main(String[] arguments) { ... }`

the call stack

the call stack

- functions can call other functions
  - the resulting "stack" of function calls is called the **call stack**

```
class Main {

    static void snap() {
        crackle();
    }

    static void crackle() {
        pop();
    }

    static void pop() {
        return;
    }

    public static void main(String[] arguments) {
        snap();
    }
}
```

Stack | Threads

Method	Line
Main.pop	12
Main.crackle	8
Main.snap	4
Main.main	16
sun.reflect.NativeMethodAccessorImpl.invoke0	-1
sun.reflect.NativeMethodAccessorImpl.invoke	62
sun.reflect.DelegatingMethodAccessorImpl.invoke	43
java.lang.reflect.Method.invoke	498
edu.rice.cs.drjava.model.compiler.JavacCompiler.runCommand	259
sun.reflect.NativeMethodAccessorImpl.invoke0	-1
sun.reflect.NativeMethodAccessorImpl.invoke	62
sun.reflect.DelegatingMethodAccessorImpl.invoke	43
java.lang.reflect.Method.invoke	498
edu.rice.cs.dynamicjava.symbol.JavaClass$JavaMethod.evalu...	362
edu.rice.cs.dynamicjava.interpreter.ExpressionEvaluator.han...	92
edu.rice.cs.dynamicjava.interpreter.ExpressionEvaluator.visit	84
koala.dynamicjava.tree.StaticMethodCall.acceptVisitor	121
edu.rice.cs.dynamicjava.interpreter.ExpressionEvaluator.value	38
edu.rice.cs.dynamicjava.interpreter.ExpressionEvaluator.value	37
edu.rice.cs.dynamicjava.interpreter.StatementEvaluator.visit	106
edu.rice.cs.dynamicjava.interpreter.StatementEvaluator.visit	29
koala.dynamicjava.tree.ExpressionStatement.acceptVisitor	101

```
sun.reflect.NativeMethodAccessorImpl.invoke0
sun.reflect.NativeMethodAccessorImpl.invoke
sun.reflect.DelegatingMethodAccessorImpl.invoke
java.lang.reflect.Method.invoke
edu.rice.cs.drjava.model.compiler.JavacCompiler.runCommand
sun.reflect.NativeMethodAccessorImpl.invoke0
sun.reflect.NativeMethodAccessorImpl.invoke
sun.reflect.DelegatingMethodAccessorImpl.invoke
java.lang.reflect.Method.invoke
edu.rice.cs.dynamicjava.symbol.JavaClass$JavaMethod.evalu...
edu.rice.cs.dynamicjava.interpreter.ExpressionEvaluator.han...
edu.rice.cs.dynamicjava.interpreter.ExpressionEvaluator.visit
koala.dynamicjava.tree.StaticMethodCall.acceptVisitor
edu.rice.cs.dynamicjava.interpreter.ExpressionEvaluator.value
edu.rice.cs.dynamicjava.interpreter.ExpressionEvaluator.value
edu.rice.cs.dynamicjava.interpreter.StatementEvaluator.visit
edu.rice.cs.dynamicjava.interpreter.StatementEvaluator.visit
koala.dynamicjava.tree.ExpressionStatement.acceptVisitor
```


[let's see what Eclipse does]

recursion

258

recursion (1/2)

- a **recursive function** is a function that calls itself
  - each call must make progress towards a **base case**
    (when the function finally returns without calling itself)
    - ✨ when in doubt, try something like zero for your base case

```
class Main extends Cow {
    static int digitSum(int n) {
        if (n == 0) {
            return 0;
        }
        return digitSum(n / 10) + (n % 10);
    }

    public static void main(String[] arguments) {
        PRINT(digitSum(256)); // 13
    }
}
```

259

```
static int digitSum(int n) {
    if (n == 0) {
        return 0;
    }
    return digitSum(n / 10) + (n % 10);
}
```

return 0;
return digitSum(0) + 2;
return digitSum(2) + 5;
return digitSum(25) + 6;
int a = digitSum(256);

260

```
static int digitSum(int n) {
    if (n == 0) {
        return 0;
    }
    return digitSum(n / 10) + (n % 10);
}
```

return 0;
return digitSum(0) + 2;
return digitSum(2) + 5;
return digitSum(25) + 6;
int a = digitSum(256);

261

```
static int digitSum(int n) {
    if (n == 0) {
        return 0;
    }
    return digitSum(n / 10) + (n % 10);
}
```

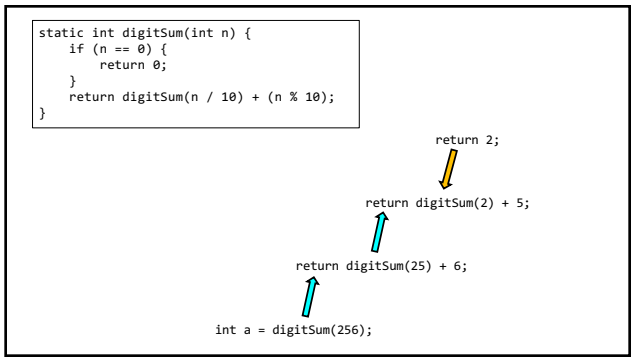
return 0 + 2;
return digitSum(2) + 5;
return digitSum(25) + 6;
int a = digitSum(256);

262

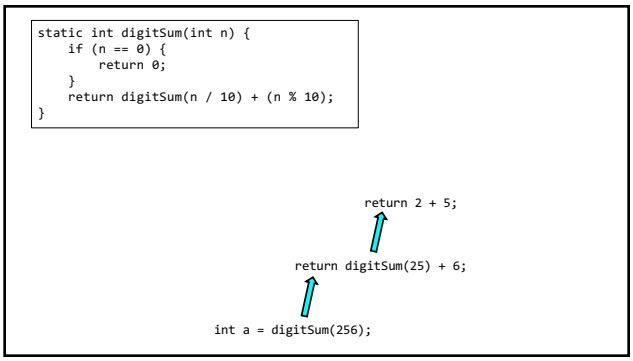
```
static int digitSum(int n) {
    if (n == 0) {
        return 0;
    }
    return digitSum(n / 10) + (n % 10);
}
```

return 2;
return digitSum(2) + 5;
return digitSum(25) + 6;
int a = digitSum(256);

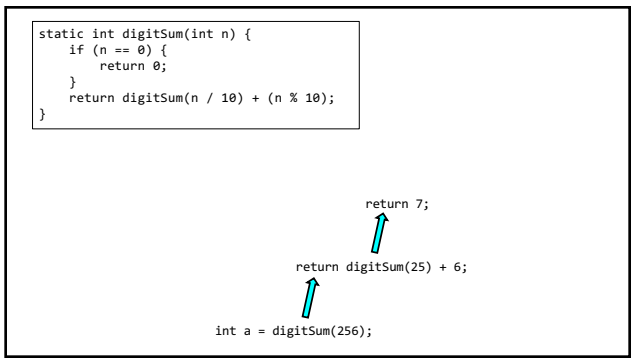
263



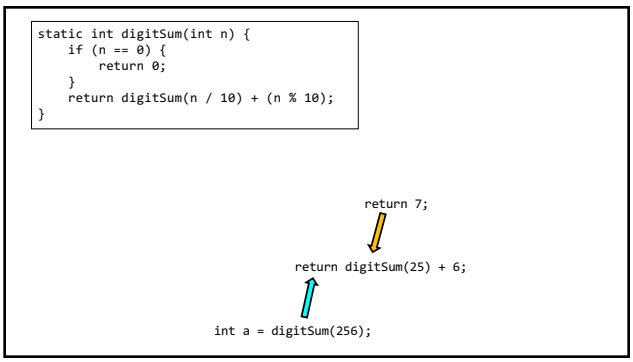
264



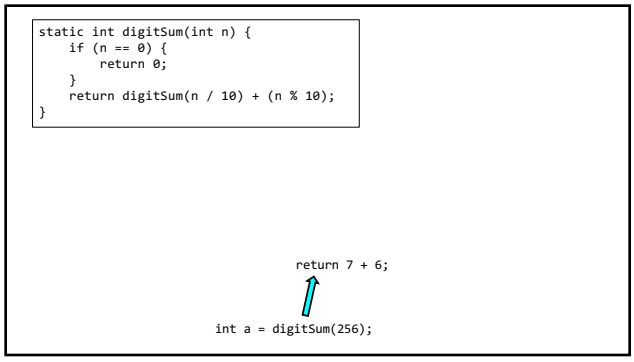
265



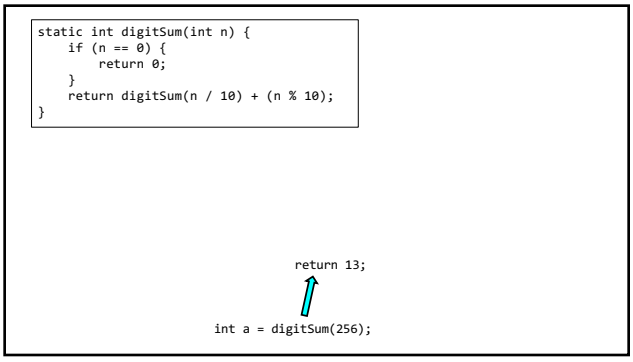
266



267



268



269

```
static int digitSum(int n) {
    if (n == 0) {
        return 0;
    }
    return digitSum(n / 10) + (n % 10);
}
```

return 13;

int a = digitSum(256);

270

```
static int digitSum(int n) {
    if (n == 0) {
        return 0;
    }
    return digitSum(n / 10) + (n % 10);
}
```

int a = 13;

271

fin.

272

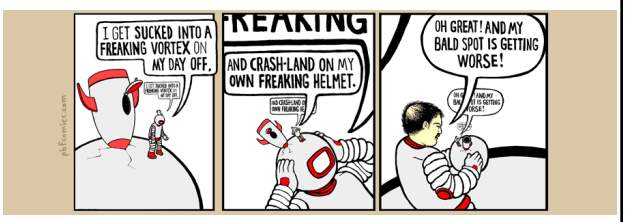
recursion (2/2)

- a **stack overflow** error happens when functions call too many functions without returning; these errors are usually caused by broken recursive code

```
class Main {
    static void stackOverflow() {
        stackOverflow();
    }

    public static void main(String[] arguments) {
        stackOverflow();
    }
}
```

```
java.lang.StackOverflowError
        at Main.stackOverflow(Tmp.java:3)
        at Main.stackOverflow(Tmp.java:3)
        at Main.stackOverflow(Tmp.java:3)
        at Main.stackOverflow(Tmp.java:3)
        at Main.stackOverflow(Tmp.java:3)
        at Main.stackOverflow(Tmp.java:3)
        at Main.stackOverflow(Tmp.java:3)
        at Main.stackOverflow(Tmp.java:3)
        at Main.stackOverflow(Tmp.java:3)
        at Main.stackOverflow(Tmp.java:3)
        at Main.stackOverflow(Tmp.java:3)
        at Main.stackOverflow(Tmp.java:3)
        at Main.stackOverflow(Tmp.java:3)
        at Main.stackOverflow(Tmp.java:3)
        at Main.stackOverflow(Tmp.java:3)
        at Main.stackOverflow(Tmp.java:3)
        at Main.stackOverflow(Tmp.java:3)
```

273



274

classes

275

anatomy of a class

276

anatomy of a class (1/2)

```
class ClassName {
    VariableOneType variableOne;
    ...

    FunctionOneReturnType functionOneName(...) { ... }
    ...
}
```

- a **class** lets you bundle together data and functions
  - a class may have any number of **variables** (fields)
    - `int foo; // objects of this class have an int called foo`
  - a class may have any number of **functions** (methods)
    - `int bar() { ... } // objects of class have function bar`

277

anatomy of a class (2/2)

```
class Thing {
    // instance variables
    double x;
    double y;
    Color color;
    double radius;

    // instance methods
    void draw() { ... }
    ...
}
```

278

dot

279

dot

- the **dot** operator is used to access an object's variables and functions

```
Thing thing = new Thing();
thing.x = 3.0;
thing.y = 4.0;
thing.draw();
```

280

terminology

281

class vs. object (instance of a class)

- a **class** is NOT the same thing as an **object**
  - a class is "a blueprint for making objects"
  - we can make an **instance of a class** (an **object**) using the **new** keyword
    - this is called "instantiating the class"
    - `Thing thing = new Thing();`

282

[off the record note on OOP (Object Oriented Programming) terminology]

283

new and constructors

284

new

- the **new** keyword create a new instance of a class and calls its appropriate **constructor**
  - `int[] array = new int[5]; // { 0, 0, 0, 0, 0 }`
  - `Color color = new Color(1.0, 0.0, 0.0); // (1.0, 0.0, 0.0)`
  - 🤓 **you don't need `new` to create a new string**
    - `String string = "strings are their own thing";`
  - 🤓 **you don't need `new` to create a new array when using {} syntax**
    - `int[] array = { 1, 2, 3 };`
  - 🤓 **`new` doesn't actually return the *object* it created; it returns *a reference to the object***

285

constructors (1/2)

- a **constructor** is called when an object is created
  - if the class does not have a constructor, then the **default constructor** must be called, which takes no arguments and sets all variables to zero
    - `Color color = new Color(); // (0.0, 0.0, 0.0)`

286

constructors (2/2)

- a (non-default) **constructor** is never necessary, but is often convenient

```
Color color = new Color(1.0, 1.0, 1.0); // (r=1.0, g=1.0, b=1.0)
```

```
Color color = new Color(); // (0.0, 0.0, 0.0)
color.r = 1.0;             // (1.0, 0.0, 0.0)
color.g = 1.0;             // (1.0, 1.0, 0.0)
color.b = 1.0;             // (1.0, 1.0, 1.0)
```

287

this

in Python, `this` is `self`

288

this

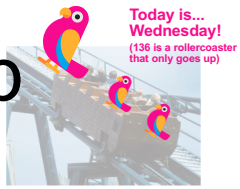
- **this** is a reference to the instance of the class whose function we're inside of
  - ✨ especially useful inside a constructor

```
class Color {
    ...

    void shade() {
        this.r /= 2;
        this.g /= 2;
        this.b /= 2;
    }

    Color(double r, double g, double b) { // constructor
        this.r = r;
        this.g = g;
        this.b = b;
    }
```

289

TODO: record lecture

Week03b



- ~~HW03 preview~~
- ~~functions~~
- ~~class~~es (not quite done)
- memory FUNdaMENTALs
- the stack and the heap
- passing arguments to functions

WARMUP
what is 42 in hex?

290

static

(wrapping up last lecture)

291

static variables and static methods

292

instance variables vs static variables

- an **instance variable** is part of an instance of a class
- a **static variable** (class variable) is part of the class itself
  - there is only one, period. (it's a global variable that lives "on the class")

```
class Thing {
    int type;
    static int TYPE_BULLET = 1;
    ...
}

// Thing thing = new Thing();
// thing.type = Thing.TYPE_BULLET;
```

293

instance methods vs static methods

- an **instance method** must be called on an instance (object) of a class
- a **static method** (class method) can be called on the class itself
  - ✅ there is no `this` in a static method

```
class Thing {
    void draw() { ... }; // (non-static method)
    static boolean collisionCheck(Thing a, Thing b) { ... }
    ...
}
// Thing a, b;
// a.draw();
// if (Thing.collisionCheck(...)) { ... }
```

294

and now... today's lecture!

295

☠️

this is the hardest lecture in 136

296

but learning this stuff is very worth

🥲 👍

297

but learning this stuff is very worth

🥲 👍

note: you will likely need to review this lecture a few times

298

but learning this stuff is very worth

🥲 👍

note: you will likely need to review this lecture a few times

(i am sure you will all do this)

299

but learning this stuff is very worth
🥹 👍

note: you will likely need to review this lecture a few times

(i am sure you will all do this 🤯)

300

memory FUNdaMENTALs

302

the two kinds of variables in Java

303

a variable in Java is either...

a primitive
or
a reference to an Object

304

primitives (review)

305

primitive types

- in this class, "a variable being a **primitive**" means that the variable is a `boolean`, `char`, `double`, or `int`

- primitive types are simple
- primitive types are small
- primitive types are NOT Objects
  - we will talk about Objects later
  - **examples of Objects:** `String`, `MyCoolClass`, `int[]` (array of ints)

306

`boolean`, `char`, `double`, `int`

- a **`boolean`** stores a truth value
  - `true`, `false`
- a **`char`** stores a character
  - `'\0'`, `'a'`, `'Z'`, `'!'`, `'\n'`
- a **`double`** stores a real number
  - `0.0`, `-0.5`, `3.1415926`, `Double.NEGATIVE_INFINITY`
- an **`int`** stores an integer
  - `0`, `-1`, `4`

307

primitives

- some examples of primitives
  - `int a;     // a is an int`
  - `boolean b; // b is a boolean`
  - `char c;    // c is a char`

308

references to Objects

309

references (1/2)

- we interact with Object's through **references**
  - `String string; // string is a reference to a String object`
  - `Color color;   // color is a reference to a Color object`
  - `int[] array;   // array is a reference to an int array`

310

references (2/2)

- a **reference** is a memory address ("where the object lives in memory")
  - a **memory address** is an **integer**
  - a memory address is often written in **hexadecimal** (**hex, base-16**, 0...9A...F)
- `Thing a = new Thing();`
- `//    ^ refers to a Thing object at memory address 0x70f806418`

311

null

312

null (1/2)

- a **null** reference refers to nothing
- the actual memory address referred to by null is zero (0x00... in hex)

```
Thing b = null;
//    ^ refers to nothing (memory address 0x000000000)
```

313

null (2/2)

```
Thing[] pool = new Thing[7];
//    ^ refers to a Thing[] object at memory address 0x70f805b68
//
// NOTE: the Thing array referred to by pool has 7 entries,
// all zero-initialized (null; memory address 0x000000000)

pool[0] = new Thing();
// pool[0] now refers to a Thing object at memory address
//                                          0x70f8079c0
```

314

the stack and the heap

(where stuff lives)

315

overview

316

let's get more specific than saying "variables live in memory"

we divide memory into two parts: **the stack** and **the heap**

317

"the stack"

local variable primitives & **references** to Objects live here
variables ***undefined*** (?) by default (will NOT compile if used)

"the heap"

the actual **Objects** (including arrays and Strings) live here
Objects are *cleared to 0* by default

318

primitives live on the stack

319

"the stack"

local variable primitives & **references** to Objects live here
variables ***undefined*** (?) by default (will NOT compile if used)

"the heap"

the actual **Objects** (including arrays and Strings) live here
Objects are *cleared to 0* by default

320

```
int a;
int b;
a = 3;
b = a;
a = 7;
```

321

a
?

```
int a;
int b;
a = 3;
b = a;
a = 7;
```

322

a
?
b
?

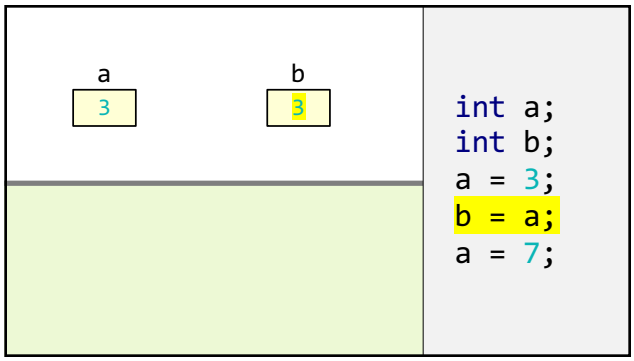
```
int a;
int b;
a = 3;
b = a;
a = 7;
```

323

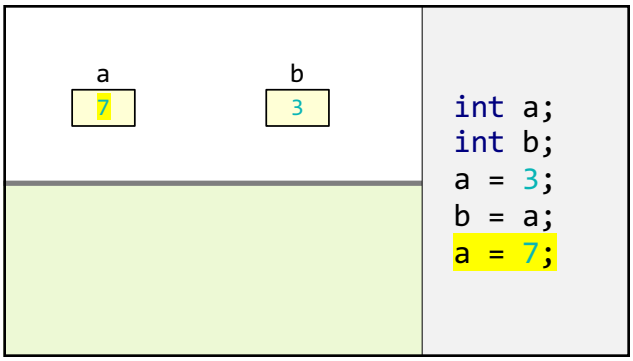
a
3
b
?

```
int a;
int b;
a = 3;
b = a;
a = 7;
```

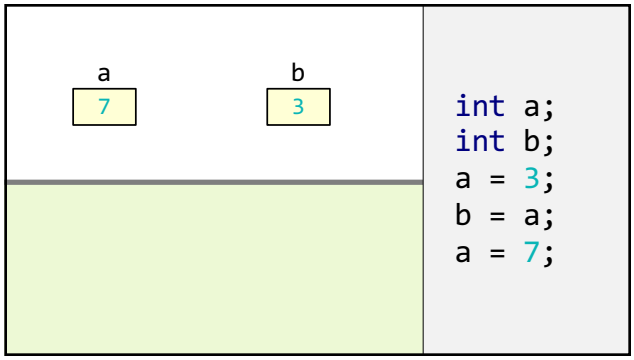
324



325



326



327

Objects live on the heap

(but *references to objects* live on the stack)

328

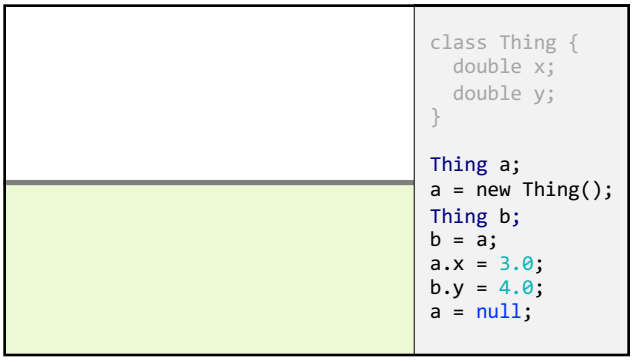
"the stack"

local variable primitives & **references** to Objects live here
variables ***undefined*** (?) by default (will NOT compile if used)

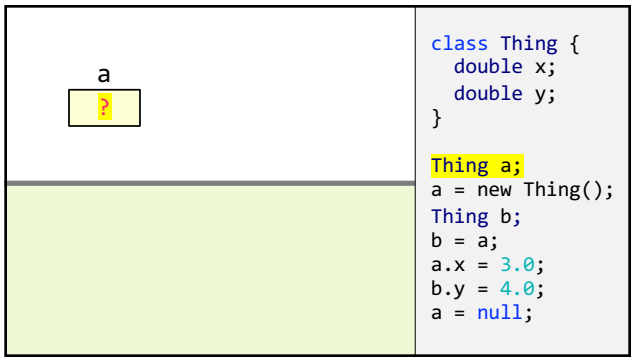
"the heap"

the actual **Objects** (including arrays and Strings) live here
Objects are *cleared to 0* by default

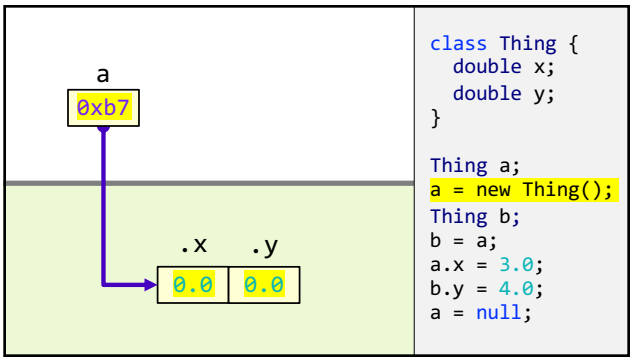
329



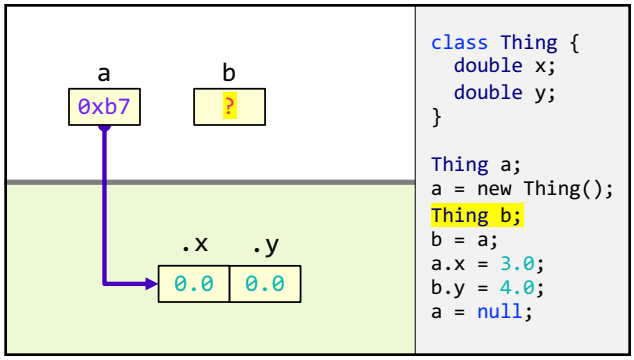
330



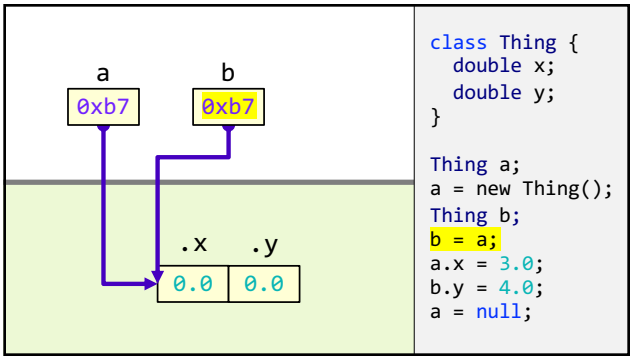
331



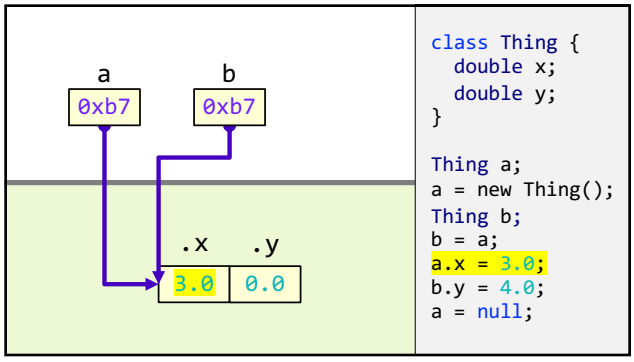
332



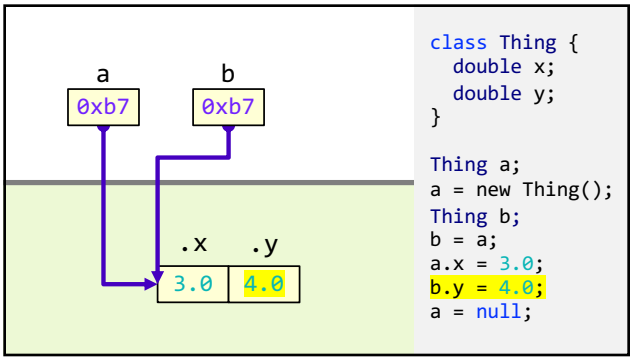
333



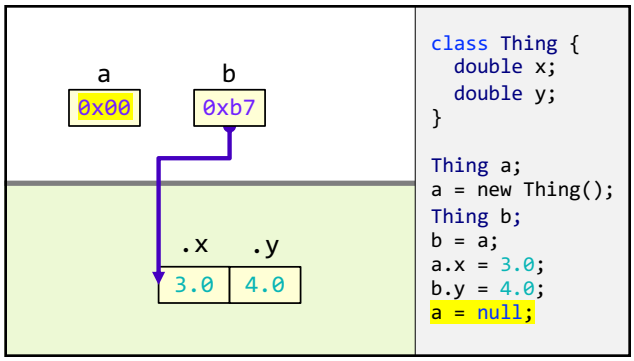
334



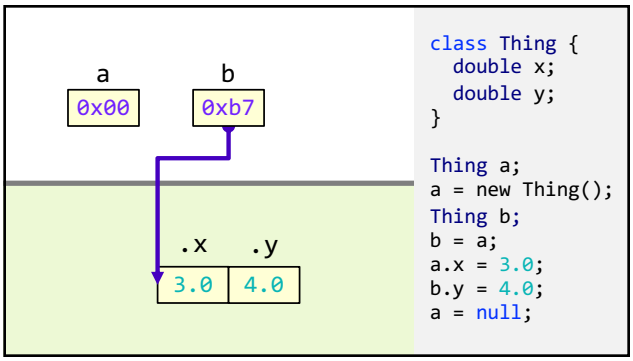
335



336



337



338

stack variables disappear when they leave scope

Objects are **garbage collected** when nothing refers to them anymore

339

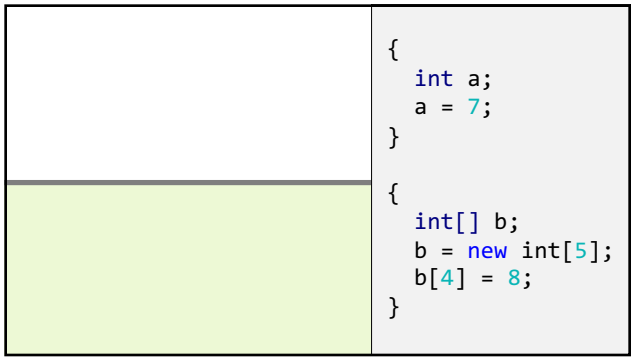
"the stack"

local variable primitives & **references** to Objects live here
variables ***undefined*** (?) by default (will NOT compile if used)

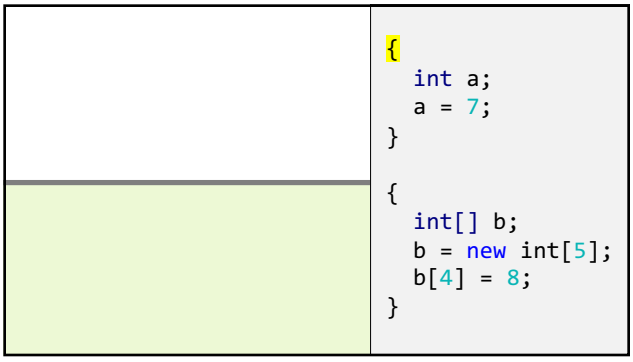
"the heap"

the actual **Objects** (including arrays and Strings) live here
Objects are *cleared to 0* by default

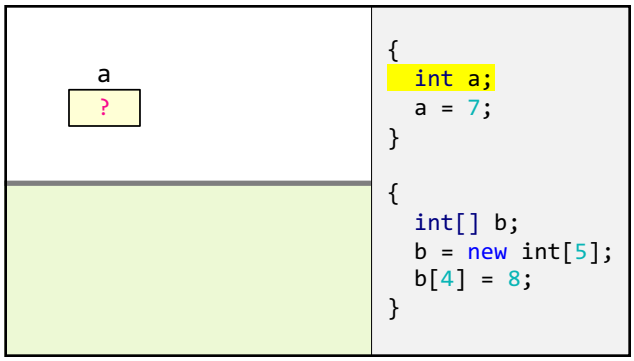
340



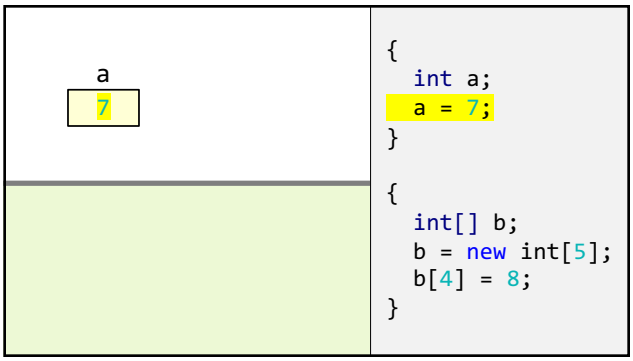
341



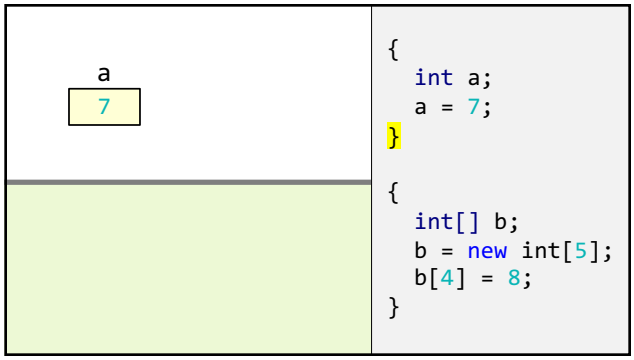
342



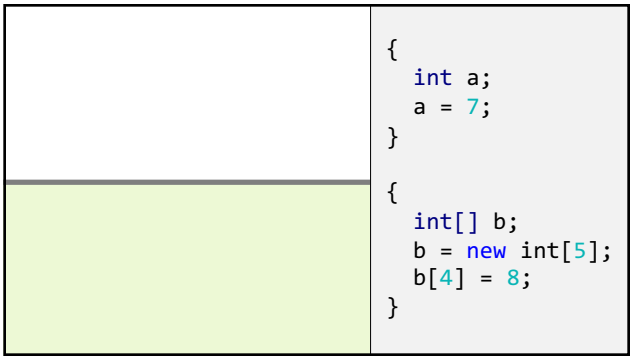
343



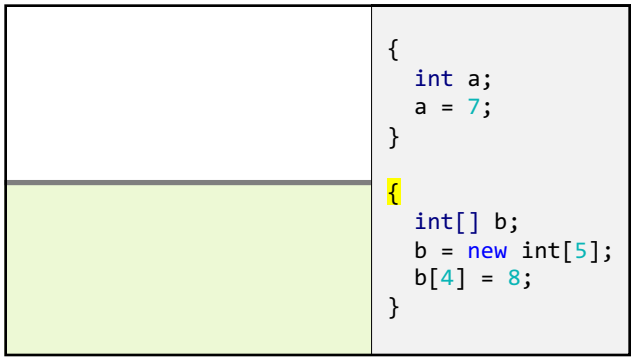
344



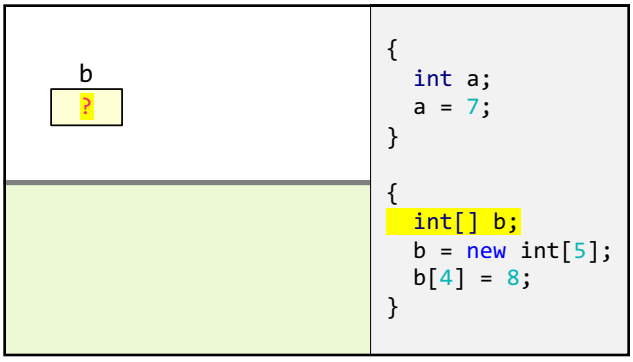
345



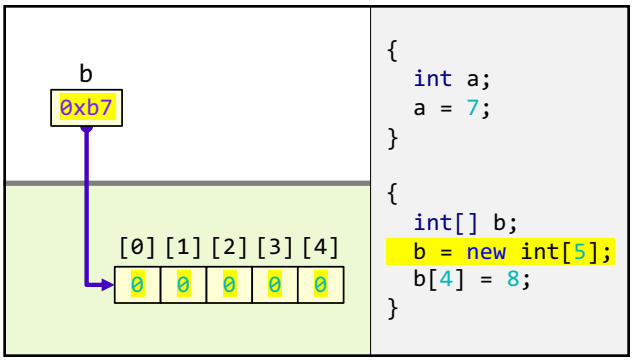
346



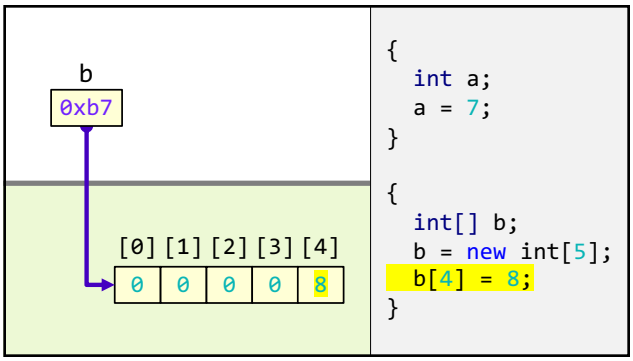
347



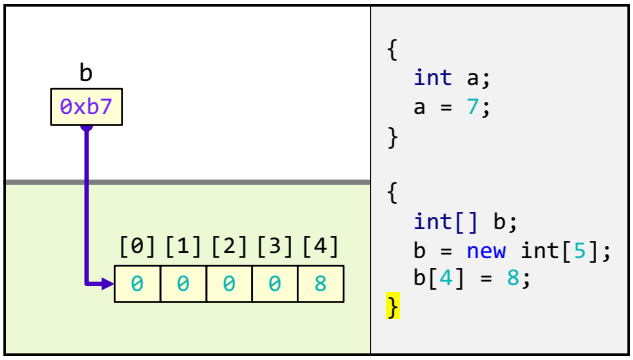
348



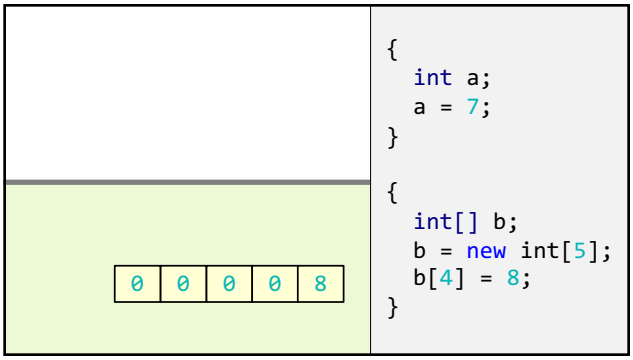
349



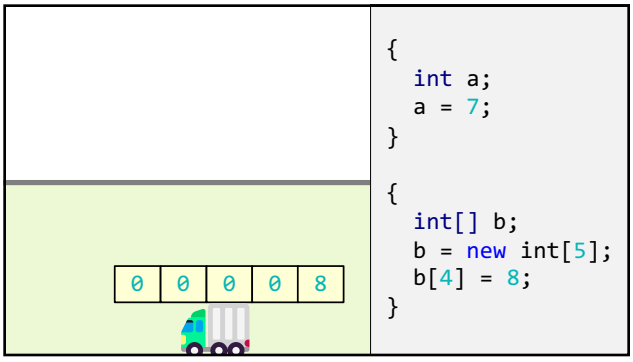
350



351



352



353

354

garbage collector

- the **garbage collector** is like a trash truck that drives around in the heap; when it notices an object that your program no longer has any references to, it frees up that memory for future use
  - C does NOT have a garbage collector; in C you free heap-allocated memory yourself by calling `free(...)`

355

[review all examples at least one more time]

356

passing arguments to functions

357

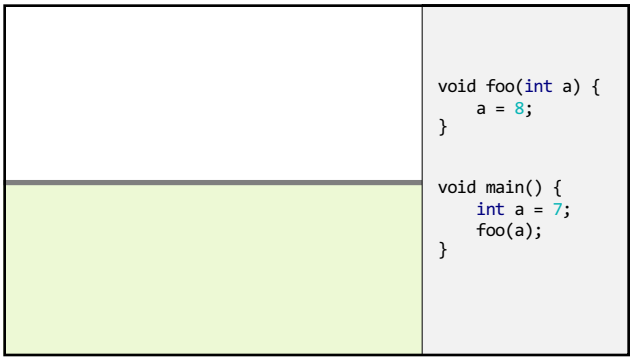
arguments to functions
are **passed by value**

(a copy of the) value of the primitive
or
(a copy of the) value of the reference

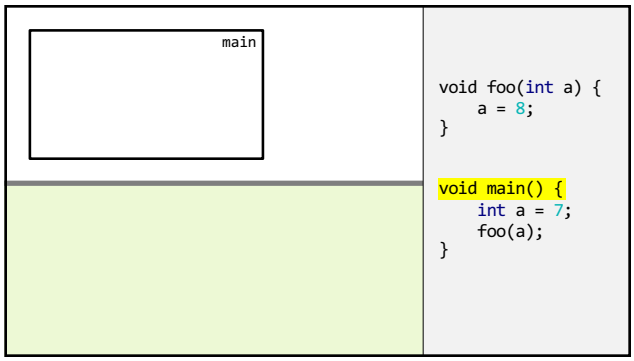
358

primitives are passed by value

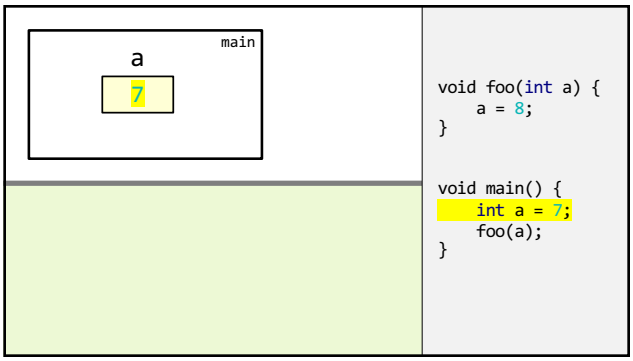
359



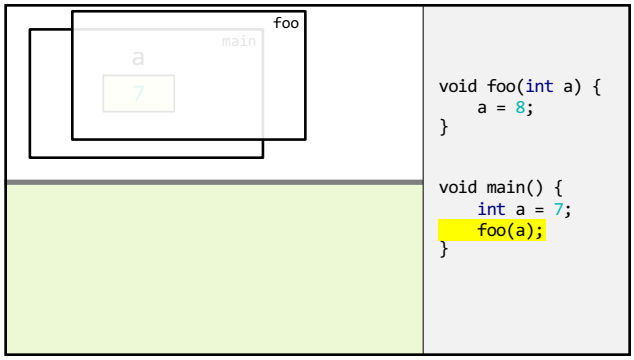
360



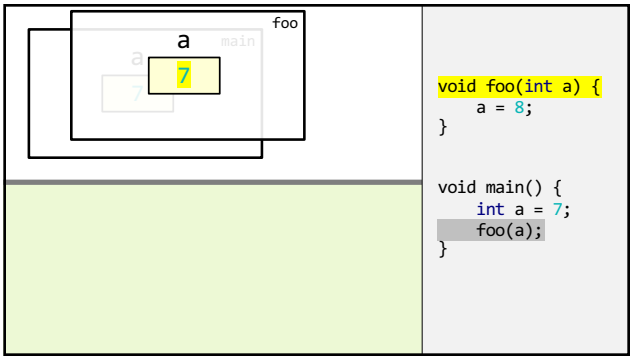
361



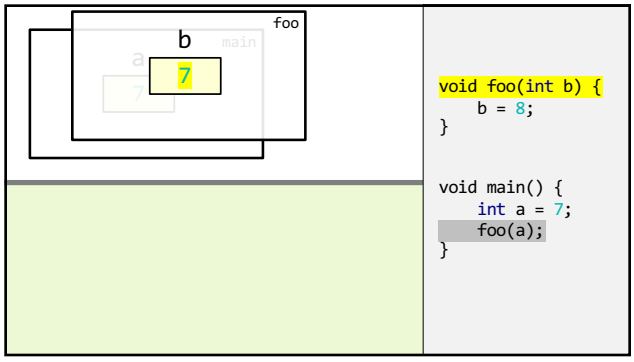
362



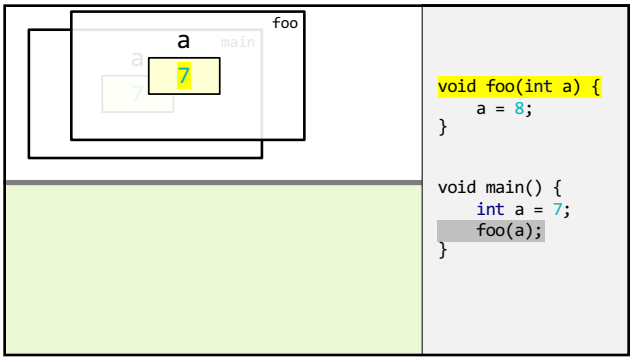
363



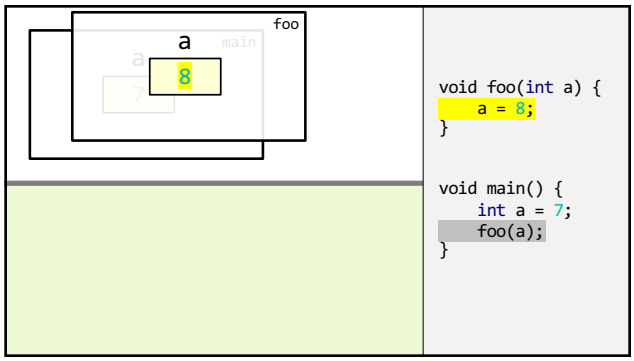
364



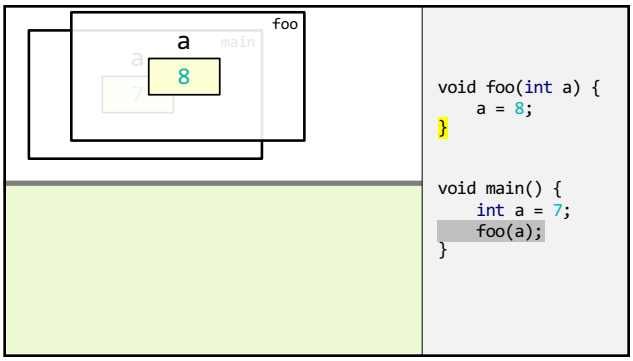
365



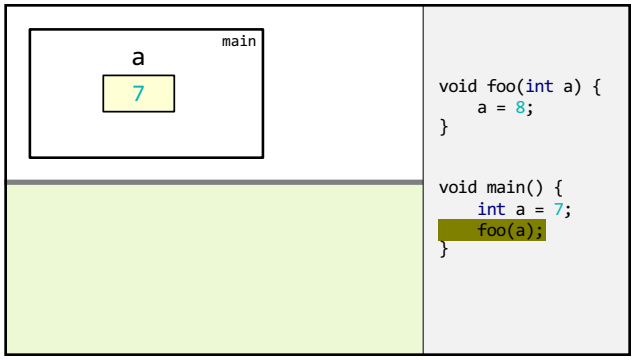
366



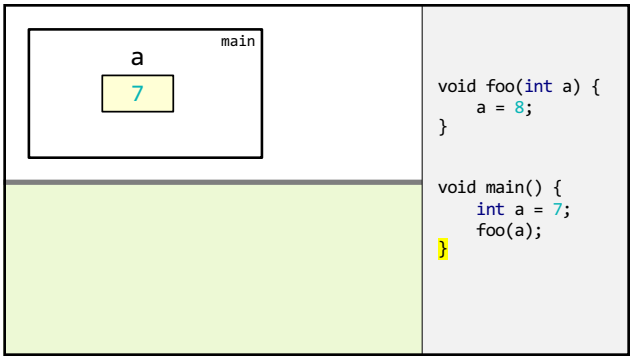
367



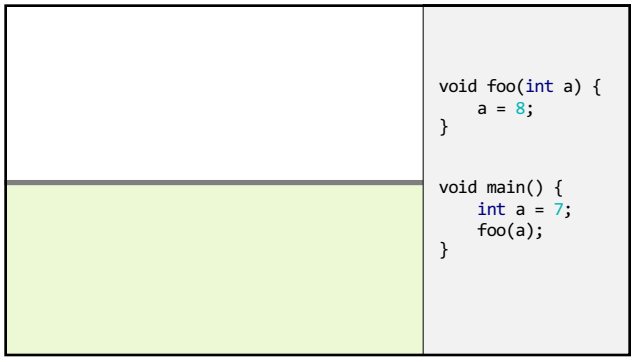
368



369



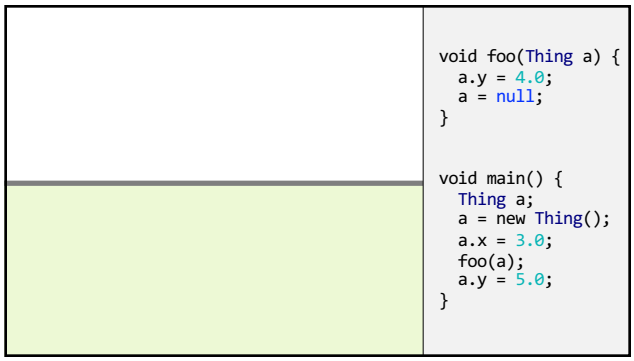
370



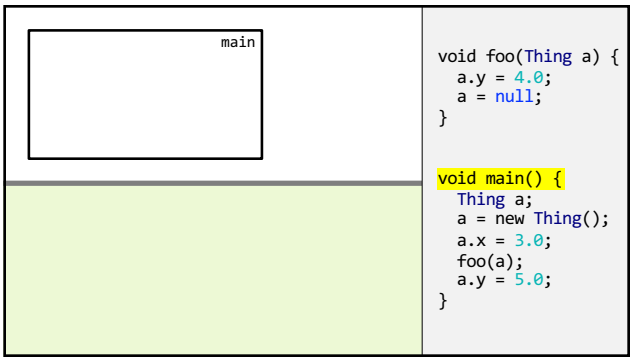
371

references are also passed by value

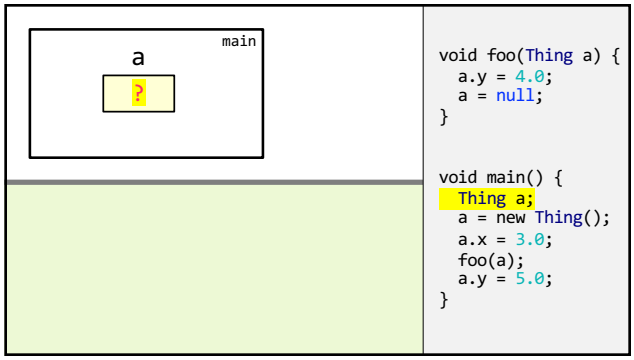
372



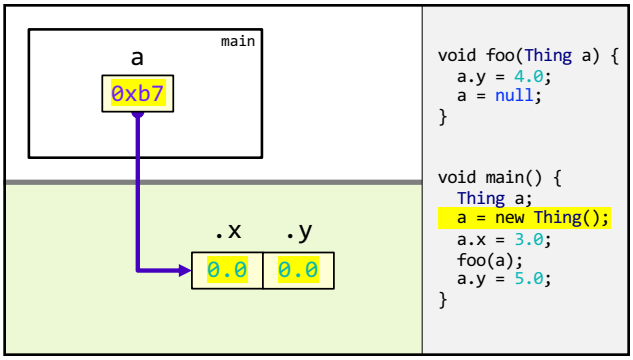
373



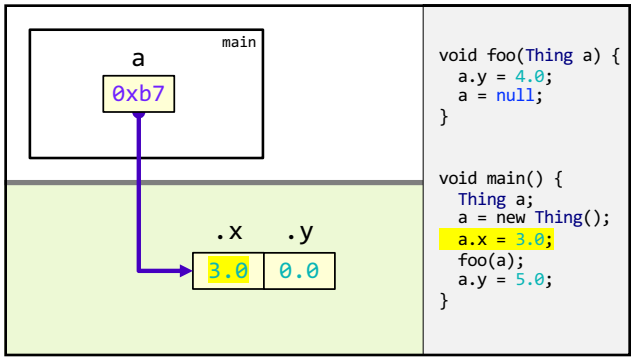
374



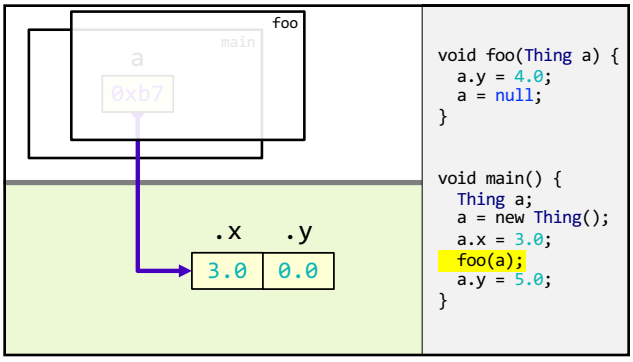
375



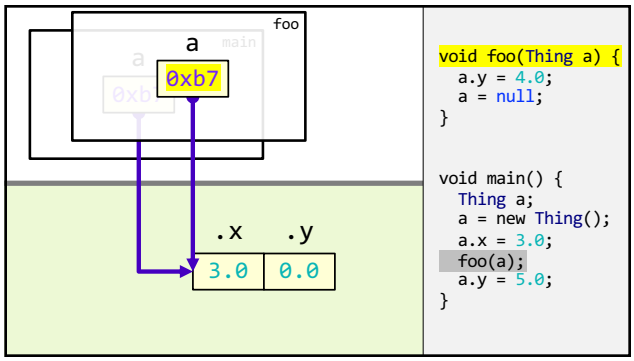
376



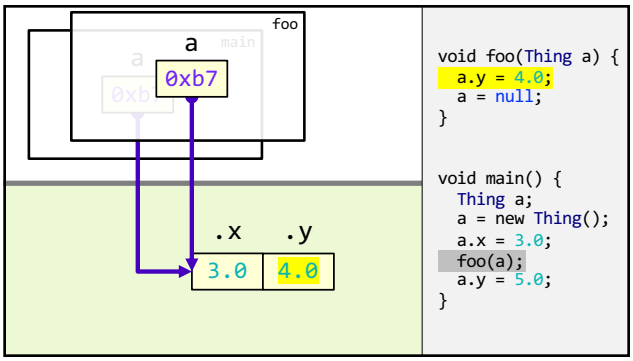
377



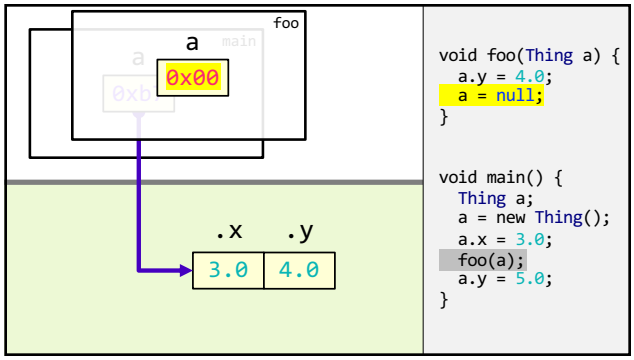
378



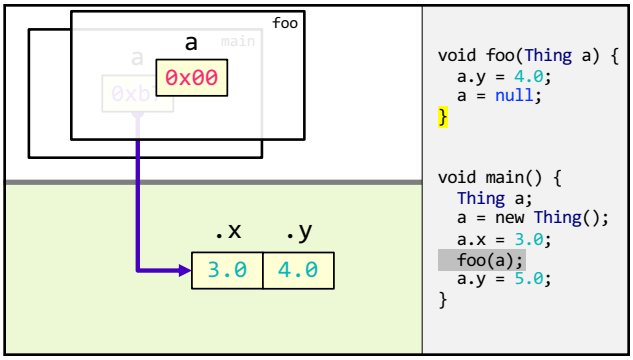
379



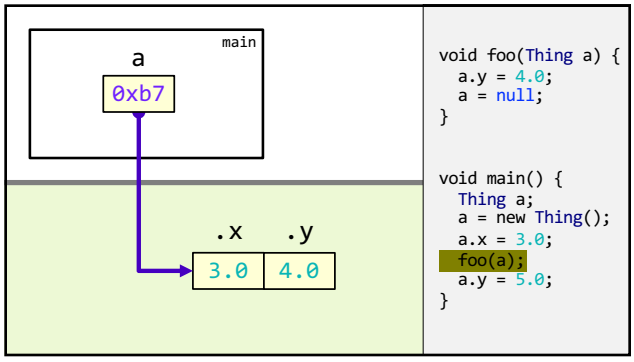
380



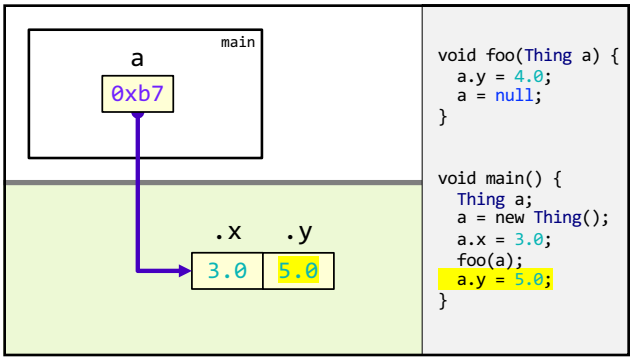
381



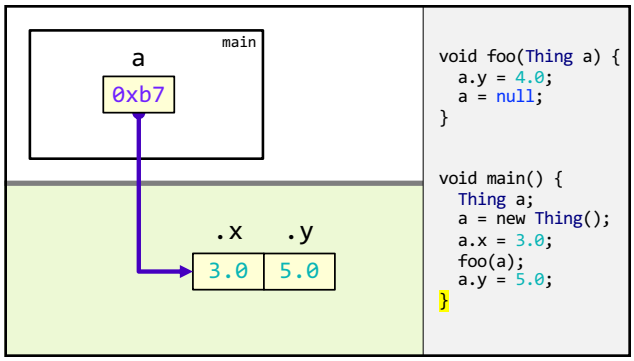
382



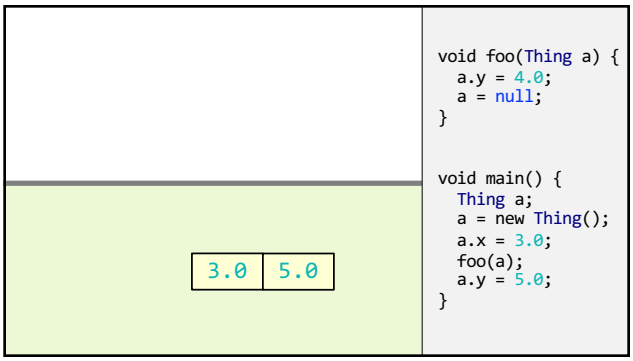
383



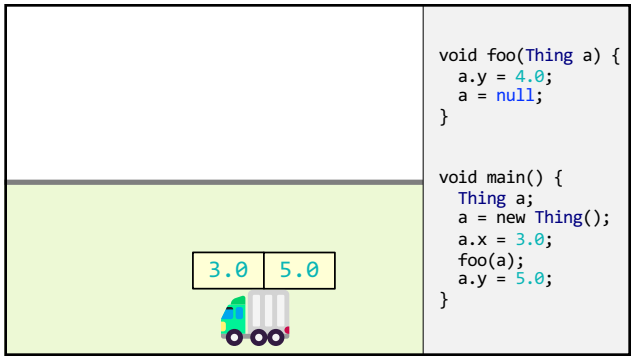
384



385



386



387

Week03c

no Jim office hours today
(i have to catch a train)

- String
- HW03 questions
- review

Today is Fun Friday!

WARMUP
what is a string?

TODO: record lecture

388

389

String (structure)

8 languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

String is a long flexible structure made from fibers twisted together into a single strand, or from multiple such strands which are in turn twisted together. String is used to tie, bind, or hang other objects. It is also used as a material to make things, such as textiles, and in arts and crafts. String is a simple tool, and its use by humans is known to have been developed tens of thousands of years ago.[1] In Mesoamerica, for example, string was invented some 20,000 to 30,000 years ago, and was made by twisting plant fibers together.[1] String may also be a component in other tools, and in devices as diverse as weapons, musical instruments, and toys.

390

Two pieces of string under a light microscope

391

392

built-in classes

(classes that are "built in" to Java)

393

built-in classes are "special"

(they break the rules of other classes in Java)

394

review:
instantiating a class
(creating an object)

395

constructors

```
// Option A: Use default constructor and initialize using dot
Color color = new Color(); // (r=0.0, g=0.0, b=0.0)
color.r = 1.0;             // (r=1.0, g=0.0, b=0.0)
color.g = 1.0;             // (r=1.0, g=1.0, b=0.0)
color.b = 1.0;             // (r=1.0, g=1.0, b=1.0)
```

```
// Option B: Use a custom constructor
Color color = new Color(1.0, 1.0, 1.0); // (r=1.0, g=1.0, b=1.0)
```

396

review: array

397

creating an array (1/2)

- you can create a new array by
  specifying its length inside of square brackets;
  if you do, the array is **zero-initialized** (all elements are initially set to zero)
  - `int[]     A = new int[8];     // { 0, 0, 0, 0, 0, 0, 0, 0 }`
  - `double[]  B = new double[1];  // { 0.0 }`
  - `boolean[] C = new boolean[2]; // { false, false }`
  - `char[]    D = new char[4];    // { '\0', '\0', '\0', '\0' }`
  - `String[]  E = new String[3];  // { null, null, null }`

398

creating an array (2/2)

- you can also create a new array by
  specifying its elements inside of curly braces;
  if you do, the array's length is the number of elements you specified
  - `int[] array = { 7, 7, 9 }; // new int[3] is implied`
    - NOTE: this syntax can only be used for **creation**
      - 🙅 NOT okay to do later: `array = { 4, 5, 6 };`
      - 🙆 OKAY to do later:
        - `array[0] = 4;`
        - `array[1] = 5;`
        - `array[2] = 6;`
  - `boolean[] array = { true };`
  - `String[] array = { "hello", "world" };`

399

String

400

`String` (1/2)

- a **`String`** is a sequence of characters
  - `"Hello World"`, `"Foo123"`

401

`String` (2/2)

- you can create a new String by
  specifying its elements inside of double quotation marks
  - `String string = "Hello";`
- `String` has some useful instance methods (see Docs)
  - `PRINT(string.length()); // 5`
  - `PRINT(string.charAt(1)); // 'e'`
  - `PRINT(string.substring(1, 3)); // "el"`

402

`String` concatenation (1/2)

- **+ concatenates two `String`'s**
  - `// String foo = "Hello".concat("World");`
    `String foo = "Hello" + "World"; // "HelloWorld"`
- **it also does some conversions for you (very convenient; kind of confusing)**
  - `// String foo = "Hello".concat(Integer.toString(2));`
    `String foo = "Hello" + 2; // "Hello2"`

403

`String` concatenation (2/2)

- `stringA + foo` is the ONLY example of **operator overloading** in Java
  - (the * operator just does multiplication)
  - (the / operator just does division)
  - ...but the + operator does both addition and string concatenation
- Python and C++ let you implement your *own* operator overloading
  - (very fun; very confusing)

404

inside of a String (1/4)

- inside of `String`, there must be a `char[]`
  - **motivation:** as you experienced in HW02, working directly with character arrays is hard / painful
  - **idea:** `String` "wraps up" a `char[]` into an easy-to-use package

405

inside of a String (2/4)

- if a `String` is really just a `char[]`...
  - ...then what does this code actually *do*? 🧠

```
String a = "Hello";
String b = "World";
String c = a + b;
PRINT(c);
```

406

XStringExample

407

inside of a String (3/4)

- if a `String` is really just a `char[]`...
  - ...then what does this code actually *do*? 🧠
  - .. how many `char[]`'s are created?

```
String a = "Hello";
String b = " ";
String c = "World";
String d = "!";
String e = a + b + c + d;
PRINT(e);
```

408

interlude

-
  -
  - ...what is this code's big O runtime? 🧠

```
int foo = 0;
for (int i = 0; i < n; ++i) { // O(n)
    foo += 1;                 // * O(1)
}
PRINT(foo);
```


inside a String (4/4)

- if a `String` is really just a `char[]`...
  - ...then what does this code *do*?
  - ...what is its big O runtime? 🧠

```
String scream = "";
for (int i = 0; i < n; ++i) {
    scream += "A";
}
PRINT(scream);
```


questions on HW03?

`pool[i]` inside of specific updates

(don't define `enemy1` and `enemy2`)

why i gave you a `player` reference

`this` inside of `fireBullet`

the art of coding things in the right order

compile and run early, compile and run often

415

well, let us imagine...

416

let's imagine...

- ...you decide to develop n different things at once...
  - ...and then, finally, you compile and run...
    - ...aaaaand you have a bug! 🐛
- so...what is the cause of the bug? what code is broken? 🤔
  - well...i guess it could be any of the n things you just wrote!
- **and while $\mathcal{O}(n)$ possible sources of bugs might seem bad...**

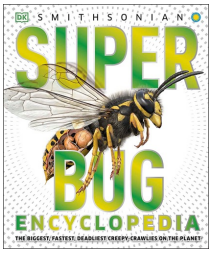
417

...the reality is

✨ **so much worse** ✨ !

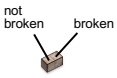
418

because broken code likes to interact with other broken code, to make ✨ **super bugs** ✨ !



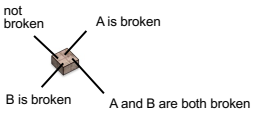
419

420



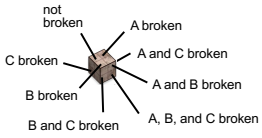
developing one
thing at a time

421



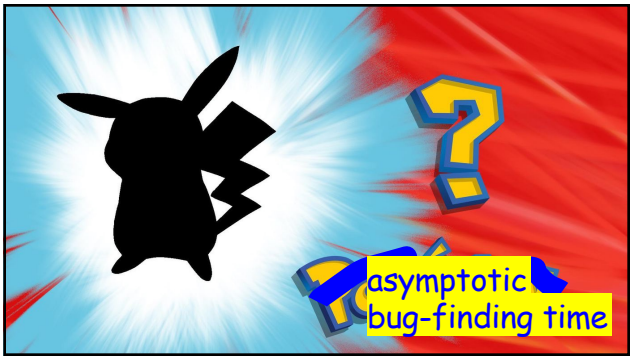
developing two
things at a time

422

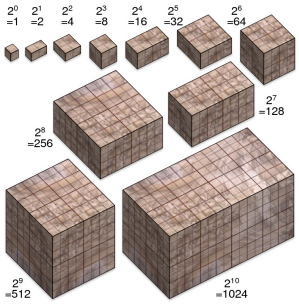


developing three
things at a time

423



424



$\mathcal{O}(2^n)$

gl;hf 😊 👍

425

big lesson

- implement code one thing at a time
- compile and run and test after implementing every single thing
- figuring out the order to do things in is a learned skill!
  - sometimes i choose the wrong order...
  - ...and then i usually have to start over 😊 👍

426

this is a learned skill!

427

in summary

428