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?

HW03 preview

👻

functions

anatomy of a function

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`

anatomy of a function (2/2)

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

return

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

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;
    }
}
```

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;
}
```

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);
}
```

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

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) { ... }`

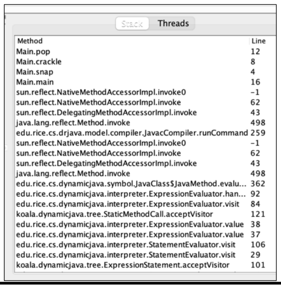
      -
        ```
        // no need to return a reference to array
        // (user of the function already has one)
        ```
  - in Java, the **main method** is a void function (it doesn't return anything)
    - `public static void main(String[] arguments) { ... }`

the call stack

the call stack

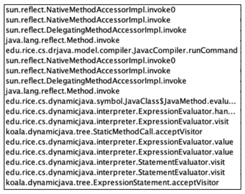
- functions can call other functions
  - the resulting "stack" of function calls is called the **call stack**

```
class Main {

    static void snap() {
        crackle();
    }

    static void crackle() {
        pop();
    }

    static void pop() {
        return;
    }

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



```
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]

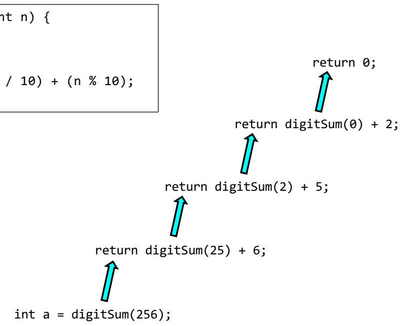
recursion

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
    }
}
```

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



return 0;

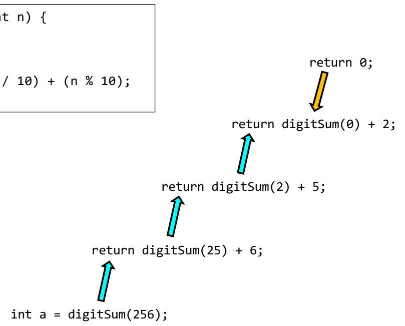
return digitSum(0) + 2;

return digitSum(2) + 5;

return digitSum(25) + 6;

int a = digitSum(256);

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



return 0;

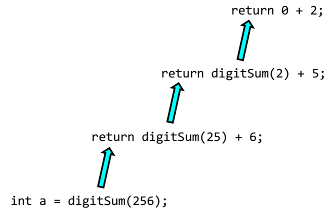
return digitSum(0) + 2;

return digitSum(2) + 5;

return digitSum(25) + 6;

int a = digitSum(256);

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



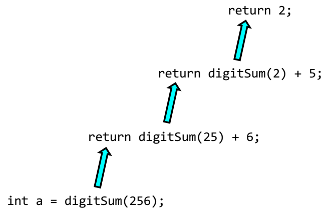
return 0 + 2;

return digitSum(2) + 5;

return digitSum(25) + 6;

int a = digitSum(256);

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

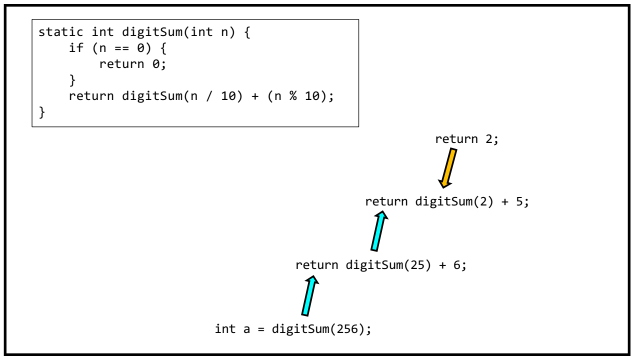


return 2;

return digitSum(2) + 5;

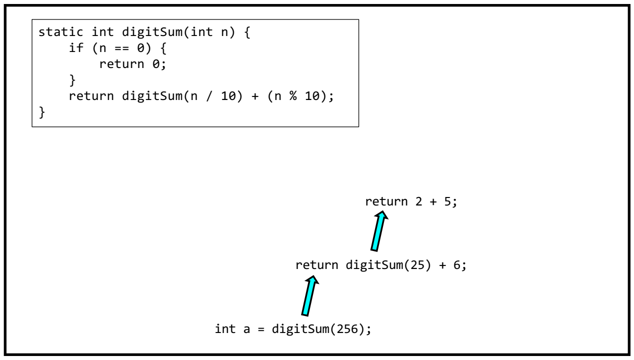
return digitSum(25) + 6;

int a = digitSum(256);



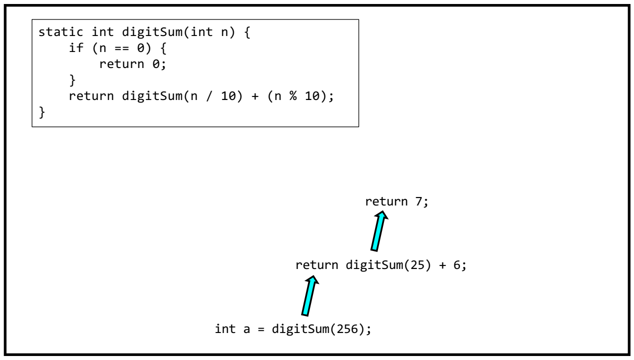
```
static int digitSum(int n) {
    if (n == 0) {
        return 0;
    }
    return digitSum(n / 10) + (n % 10);
}
```

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



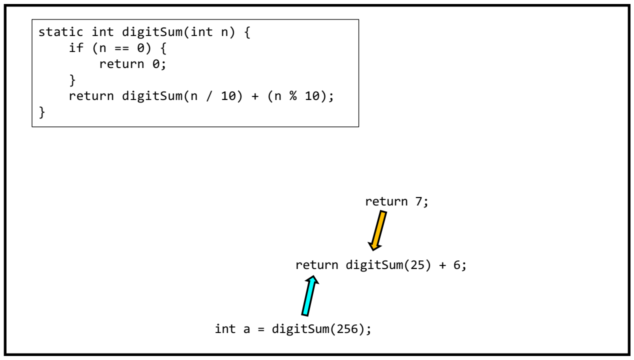
```
static int digitSum(int n) {
    if (n == 0) {
        return 0;
    }
    return digitSum(n / 10) + (n % 10);
}
```

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



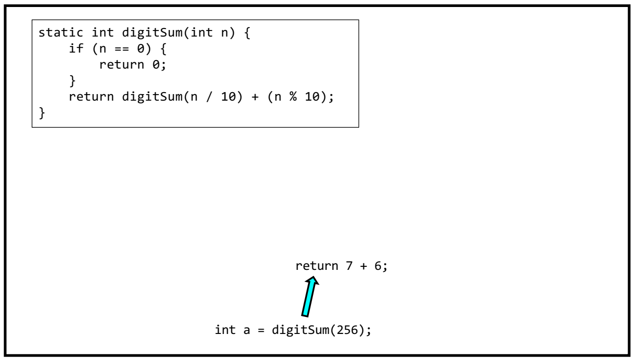
```
static int digitSum(int n) {
    if (n == 0) {
        return 0;
    }
    return digitSum(n / 10) + (n % 10);
}
```

```
return 7;
return digitSum(25) + 6;
int a = digitSum(256);
```



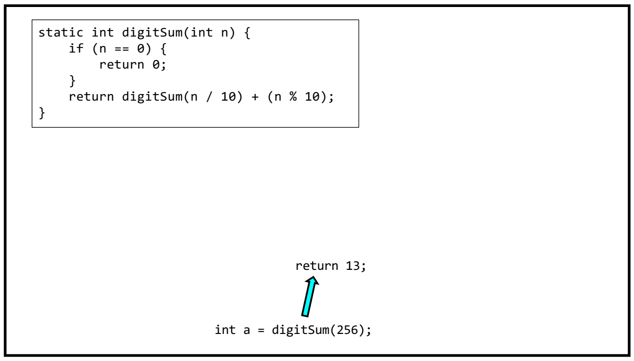
```
static int digitSum(int n) {
    if (n == 0) {
        return 0;
    }
    return digitSum(n / 10) + (n % 10);
}
```

```
return 7;
return digitSum(25) + 6;
int a = digitSum(256);
```



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

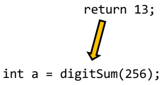
```
return 7 + 6;
int a = digitSum(256);
```



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

```
return 13;
int a = digitSum(256);
```

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



```
return 13;
```

```
int a = digitSum(256);
```

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

```
int a = 13;
```

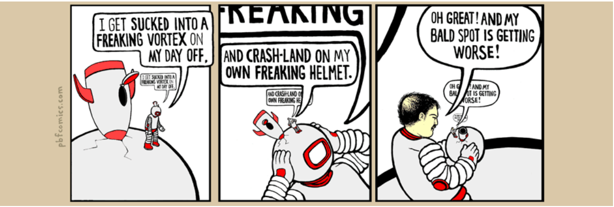
fin.

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


I GET SUCKED INTO A FREAKING VORTEX ON MY DAY OFF.
FREAKING
AND CRASH-LAND ON MY OWN FREAKING HELMET.
OH GREAT! AND MY BALD SPOT IS GETTING WORSE!
pbfcomics.com

classes

anatomy of a class

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`

anatomy of a class (2/2)

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

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

dot

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();
```

terminology

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();`

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

new and constructors

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***

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)`

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)
```

this

in Python, `this` is `self`

this (1/2)

- **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;
    }
```

TODO: record lecture

Week03b

Today is... Wednesday!

(136 is a rollercoaster that only goes up)

- ~~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?

static

(wrapping up last lecture)

static variables and static methods

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;
```

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(...)) { ... }
```

and now...
today's lecture!

☠️
this is the hardest lecture in 136

but learning this stuff is very worth
🙂 👍

but learning this stuff is very worth
🙂 👍
note: you will likely need to review this lecture a few times

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)

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 🥲)

memory FUNdaMENTALs

the two kinds of variables in Java

a variable in Java is either...

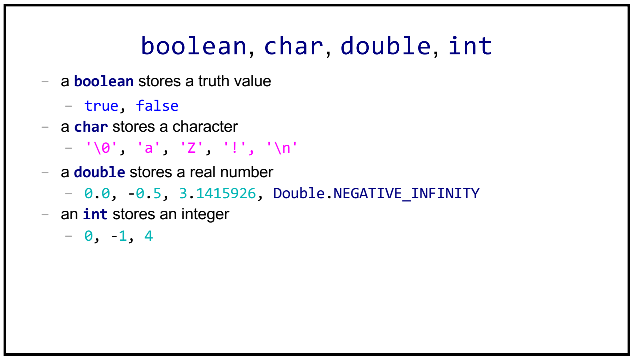
a primitive
or
a reference to an Object

primitives (review)

primitive types

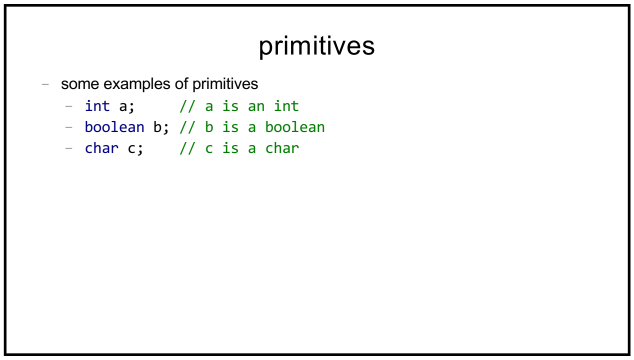
- in this class, "a variable being a **primitive**" means that the variable is a `boolean`, `char`, `double`, or `int`

- 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)



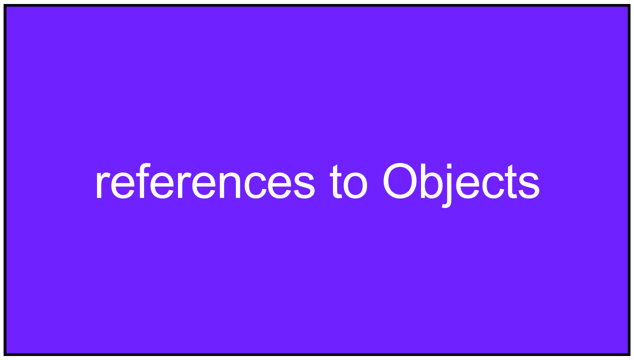
boolean, char, double, int

- 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`

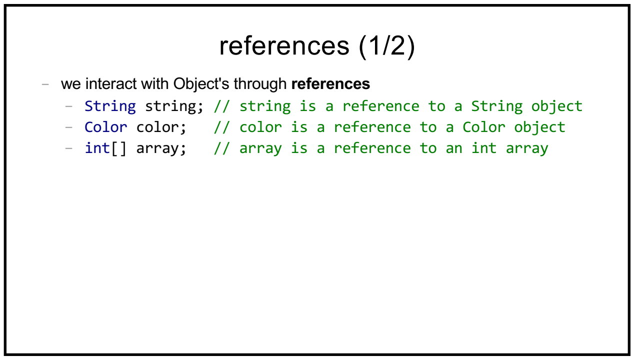


primitives

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

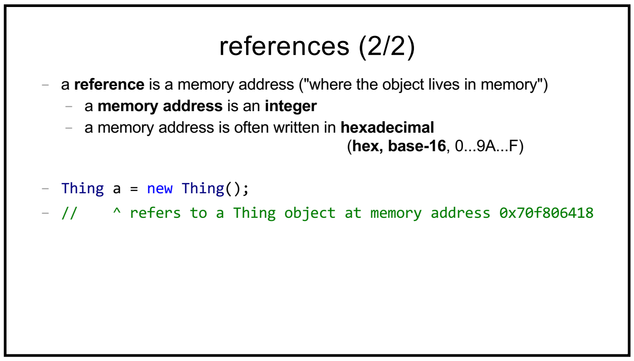


references to Objects



references (1/2)

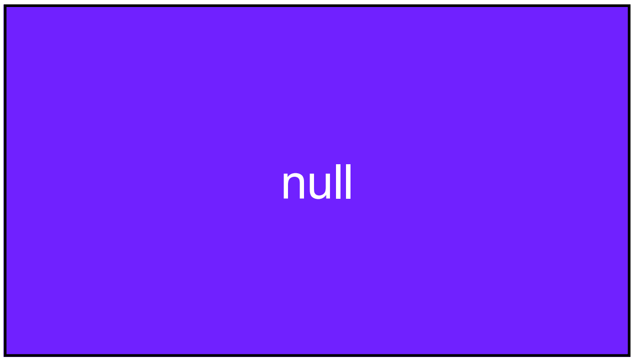
- we interact with Object's through **references**
  - `String string; // string is a reference to a String object`
  - `Color color;   // color is a reference to a Color object`
  - `int[] array;   // array is a reference to an int array`



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
```



null

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)
```

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
```

the stack and the heap

(where stuff lives)

overview

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

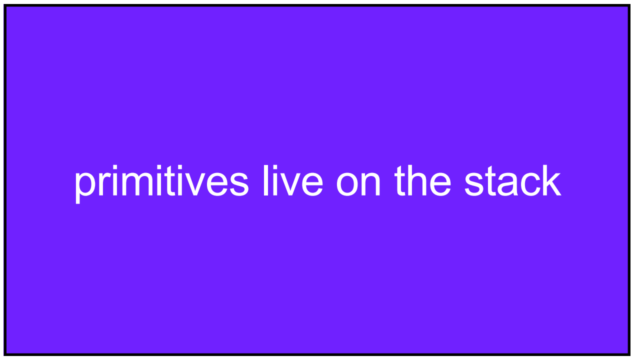
we divide memory into two parts: **the stack** and **the heap**

"the stack"

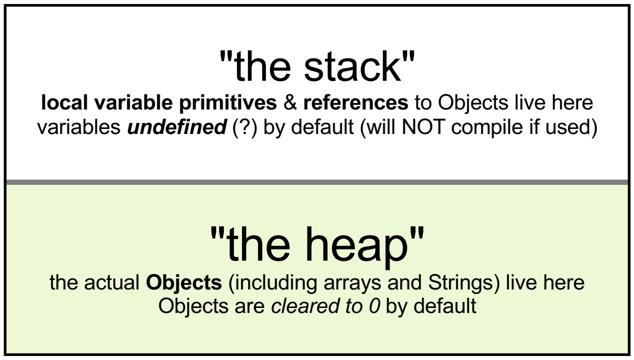
local variable primitives & **references** to Objects live here
variables ***undefined*** (?) by default (will NOT compile if used)

"the heap"

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



primitives live on the stack

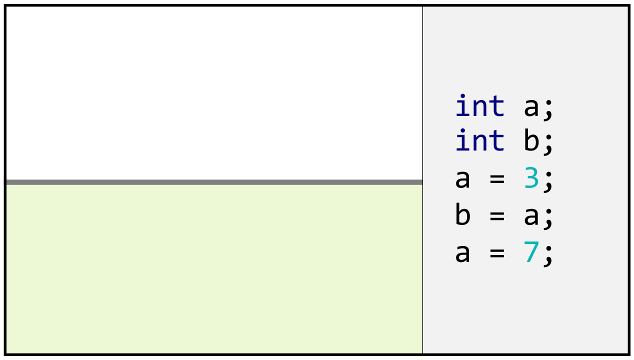


"the stack"

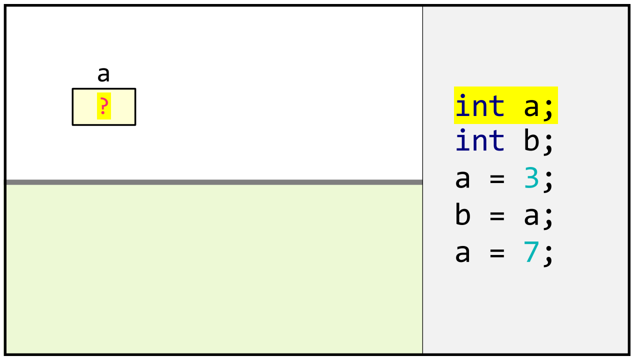
local variable primitives & **references** to Objects live here
variables ***undefined*** (?) by default (will NOT compile if used)

"the heap"

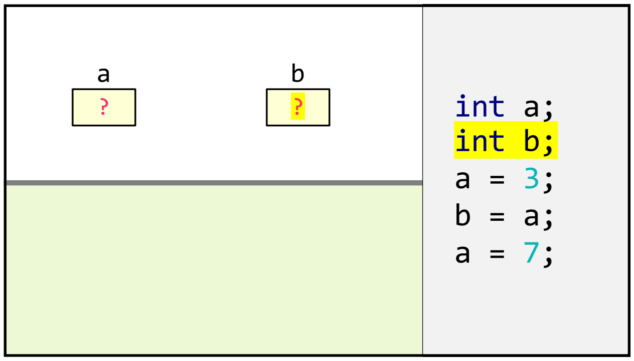
the actual **Objects** (including arrays and Strings) live here
Objects are *cleared to 0* by default



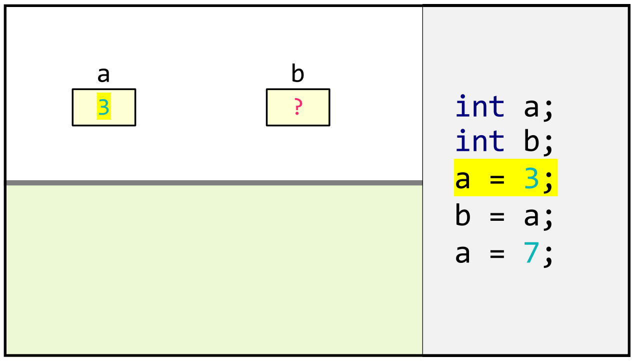
```
int a;
int b;
a = 3;
b = a;
a = 7;
```



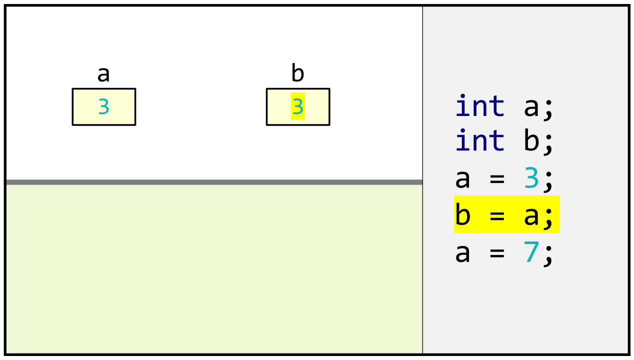
```
int a;
int b;
a = 3;
b = a;
a = 7;
```



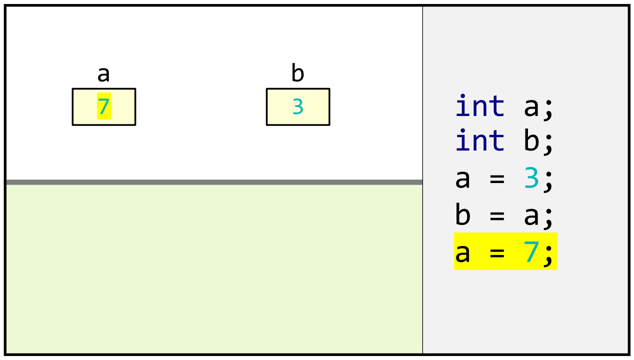
```
int a;
int b;
a = 3;
b = a;
a = 7;
```



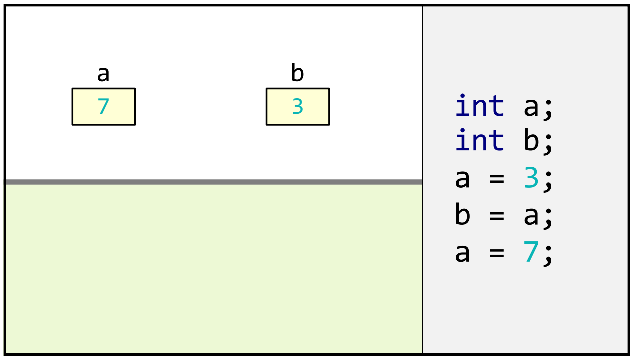
```
int a;
int b;
a = 3;
b = a;
a = 7;
```



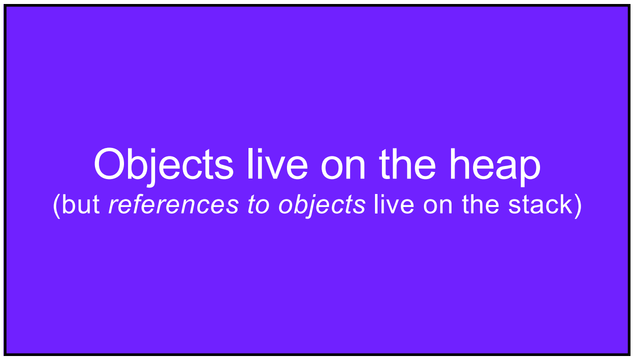
```
int a;
int b;
a = 3;
b = a;
a = 7;
```



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

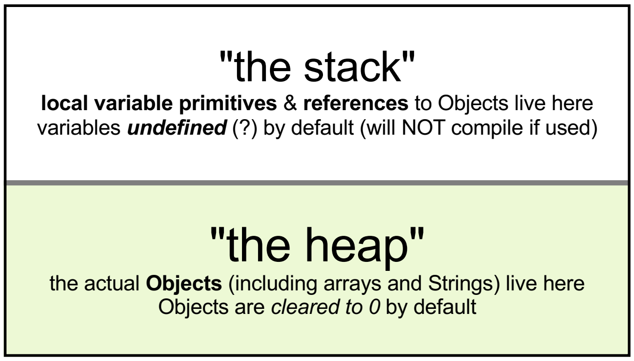


```
int a;
int b;
a = 3;
b = a;
a = 7;
```



Objects live on the heap

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

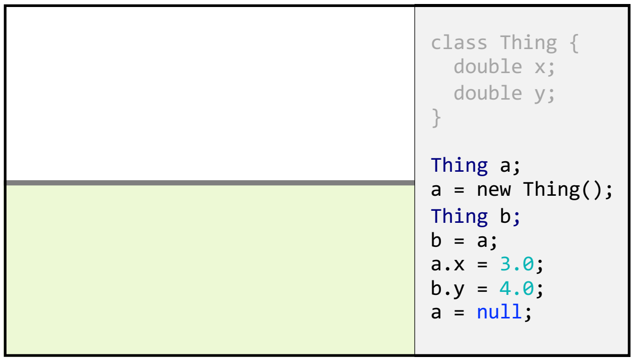


"the stack"

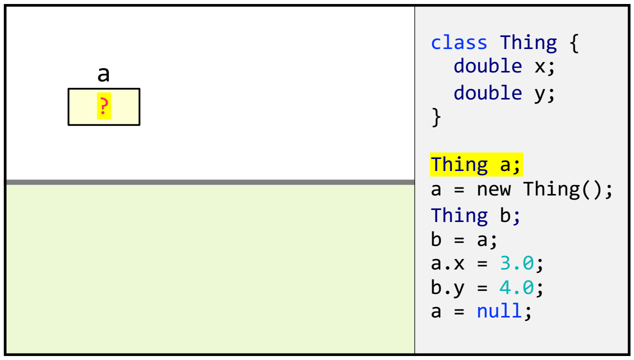
local variable primitives & **references** to Objects live here
variables ***undefined*** (?) by default (will NOT compile if used)

"the heap"

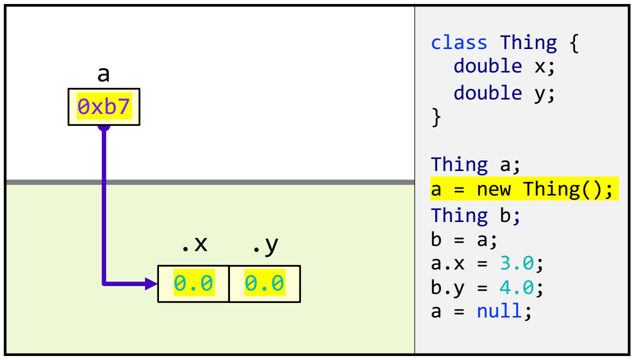
the actual **Objects** (including arrays and Strings) live here
Objects are *cleared to 0* by default



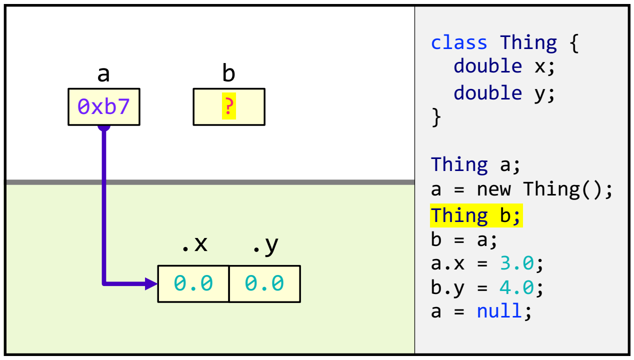
```
class Thing {
  double x;
  double y;
}

Thing a;
a = new Thing();
Thing b;
b = a;
a.x = 3.0;
b.y = 4.0;
a = null;
```



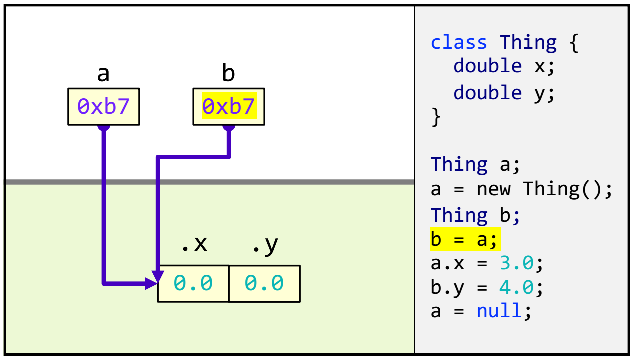
```
class Thing {
  double x;
  double y;
}

Thing a;
a = new Thing();
Thing b;
b = a;
a.x = 3.0;
b.y = 4.0;
a = null;
```



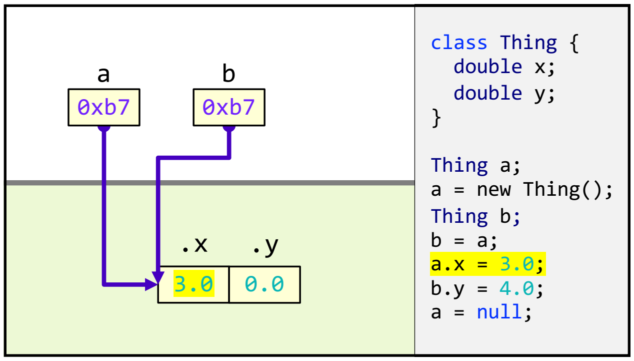
```
class Thing {
  double x;
  double y;
}

Thing a;
a = new Thing();
Thing b;
b = a;
a.x = 3.0;
b.y = 4.0;
a = null;
```



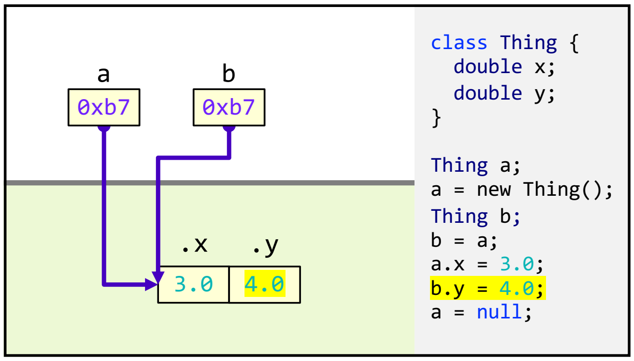
```
class Thing {
  double x;
  double y;
}

Thing a;
a = new Thing();
Thing b;
b = a;
a.x = 3.0;
b.y = 4.0;
a = null;
```



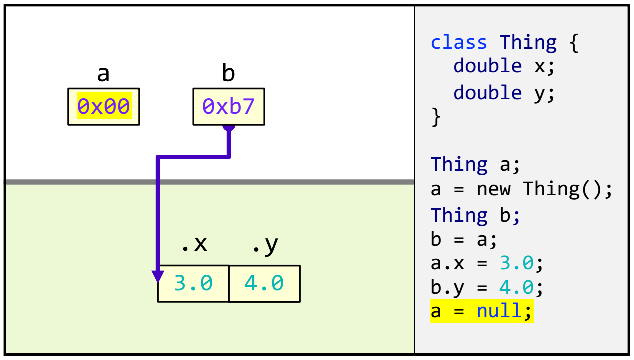
```
class Thing {
  double x;
  double y;
}

Thing a;
a = new Thing();
Thing b;
b = a;
a.x = 3.0;
b.y = 4.0;
a = null;
```



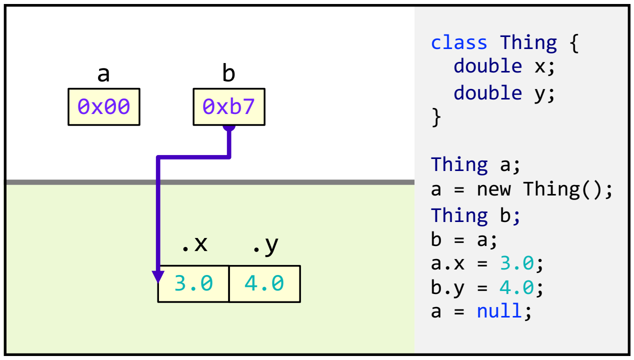
```
class Thing {
  double x;
  double y;
}

Thing a;
a = new Thing();
Thing b;
b = a;
a.x = 3.0;
b.y = 4.0;
a = null;
```



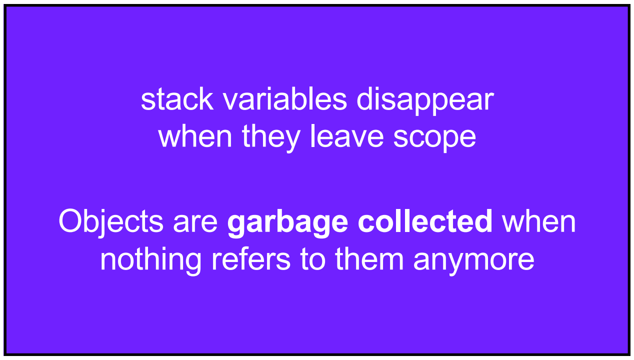
```
class Thing {
  double x;
  double y;
}

Thing a;
a = new Thing();
Thing b;
b = a;
a.x = 3.0;
b.y = 4.0;
a = null;
```



```
class Thing {
  double x;
  double y;
}

Thing a;
a = new Thing();
Thing b;
b = a;
a.x = 3.0;
b.y = 4.0;
a = null;
```

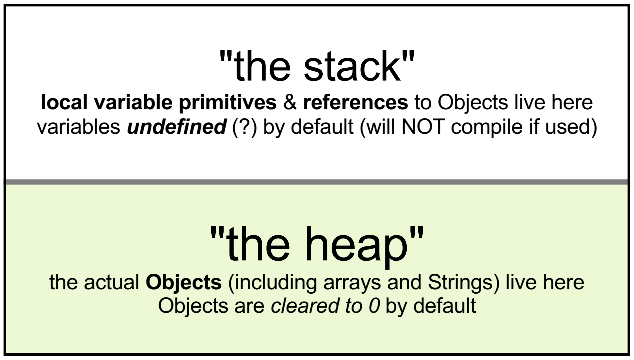


```
class Thing {
  double x;
  double y;
}

Thing a;
a = new Thing();
Thing b;
b = a;
a.x = 3.0;
b.y = 4.0;
a = null;
```



stack variables disappear
when they leave scope

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

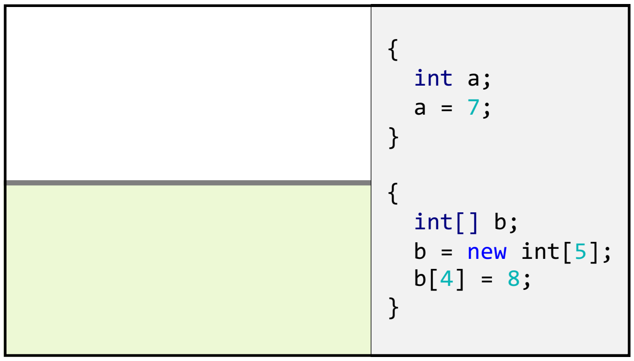


"the stack"

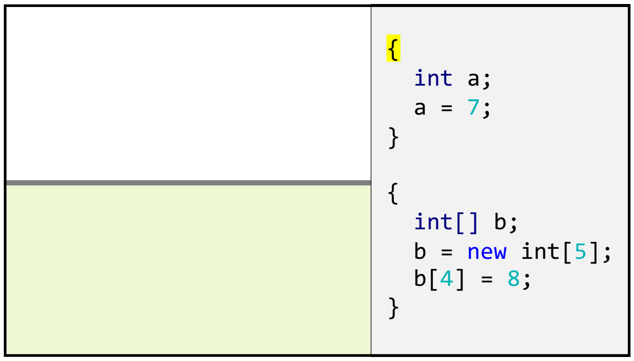
local variable primitives & **references** to Objects live here
variables ***undefined*** (?) by default (will NOT compile if used)

"the heap"

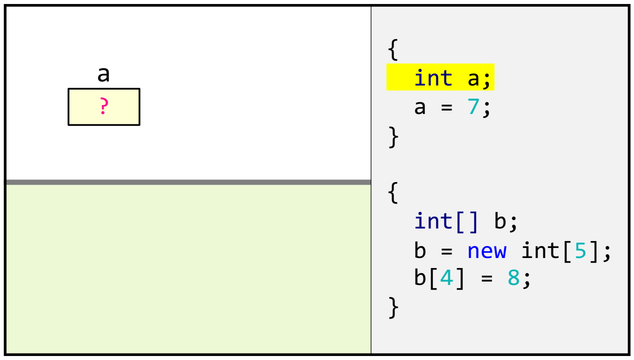
the actual **Objects** (including arrays and Strings) live here
Objects are *cleared to 0* by default



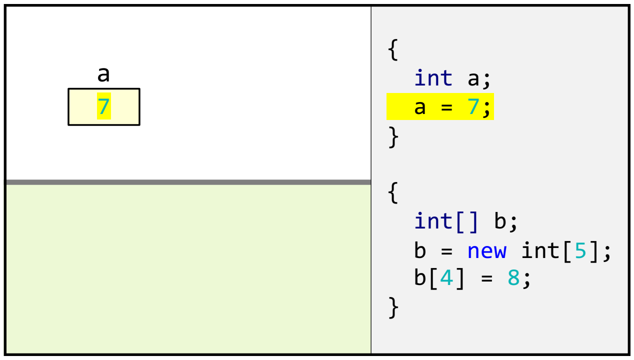
```
{
  int a;
  a = 7;
}

{
  int[] b;
  b = new int[5];
  b[4] = 8;
}
```



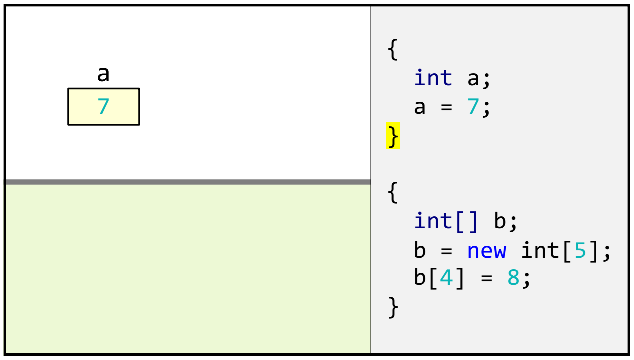
```
{
  int a;
  a = 7;
}

{
  int[] b;
  b = new int[5];
  b[4] = 8;
}
```



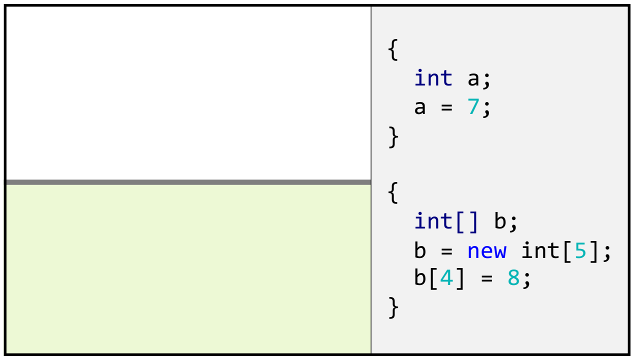
```
{
  int a;
  a = 7;
}

{
  int[] b;
  b = new int[5];
  b[4] = 8;
}
```



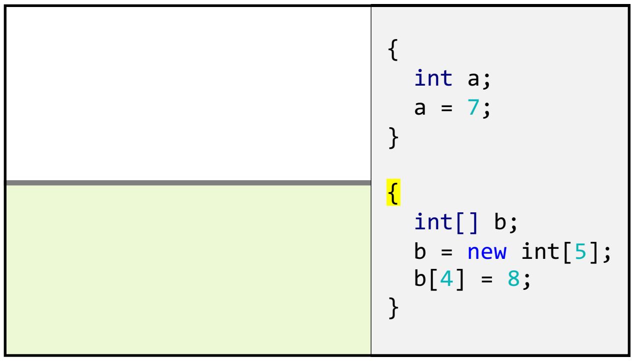
```
{
  int a;
  a = 7;
}

{
  int[] b;
  b = new int[5];
  b[4] = 8;
}
```



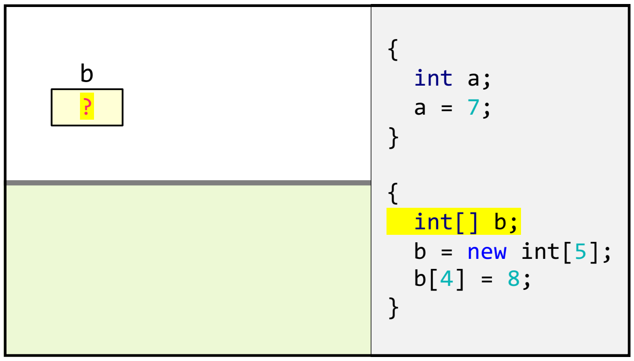
```
{
  int a;
  a = 7;
}

{
  int[] b;
  b = new int[5];
  b[4] = 8;
}
```



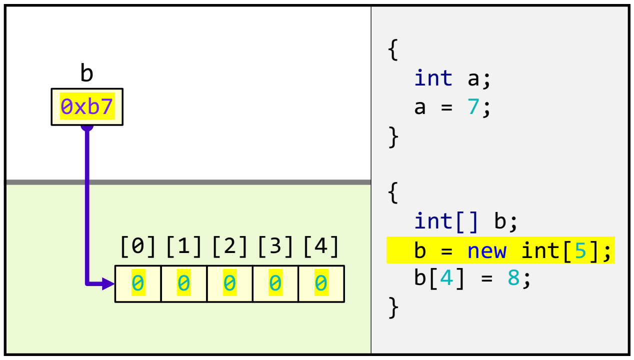
```
{
  int a;
  a = 7;
}

{
  int[] b;
  b = new int[5];
  b[4] = 8;
}
```



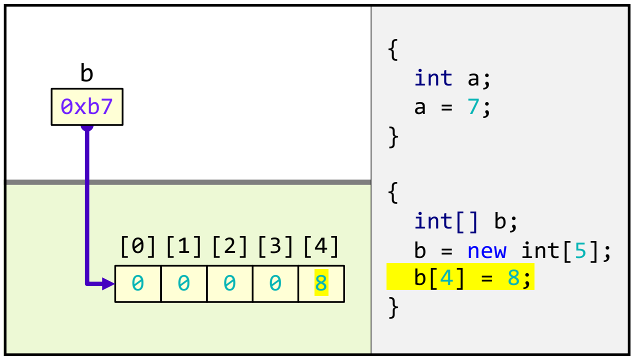
```
{
  int a;
  a = 7;
}

{
  int[] b;
  b = new int[5];
  b[4] = 8;
}
```



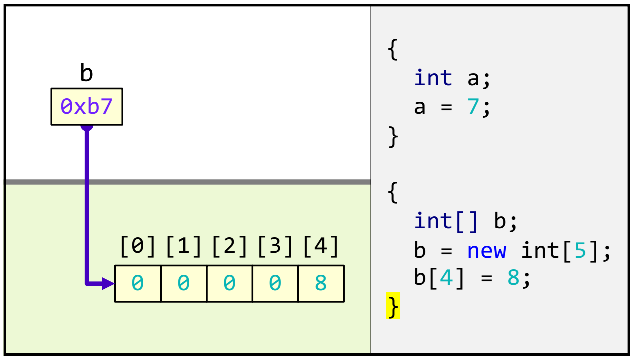
```
{
  int a;
  a = 7;
}

{
  int[] b;
  b = new int[5];
  b[4] = 8;
}
```



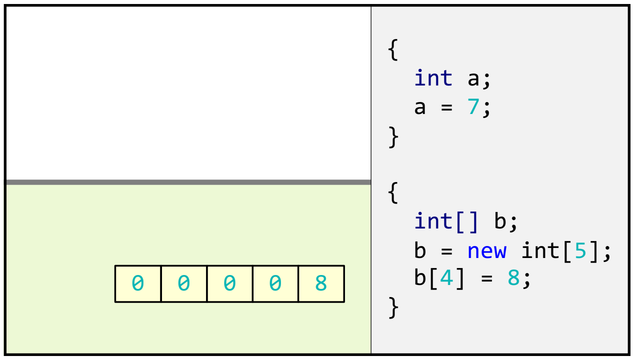
```
{
  int a;
  a = 7;
}

{
  int[] b;
  b = new int[5];
  b[4] = 8;
}
```



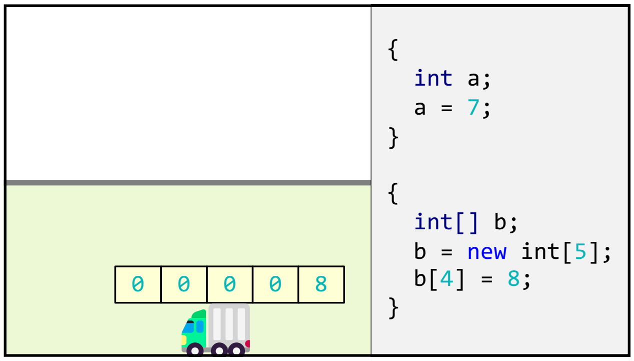
```
{
  int a;
  a = 7;
}

{
  int[] b;
  b = new int[5];
  b[4] = 8;
}
```



```
{
  int a;
  a = 7;
}

{
  int[] b;
  b = new int[5];
  b[4] = 8;
}
```



```
{
  int a;
  a = 7;
}

{
  int[] b;
  b = new int[5];
  b[4] = 8;
}
```

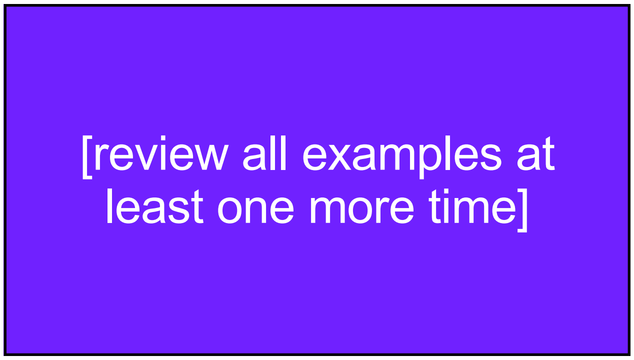


```
{
  int a;
  a = 7;
}

{
  int[] b;
  b = new int[5];
  b[4] = 8;
}
```

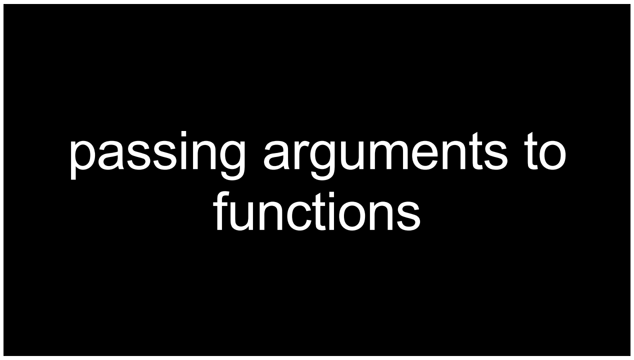
garbage collector

garbage collector

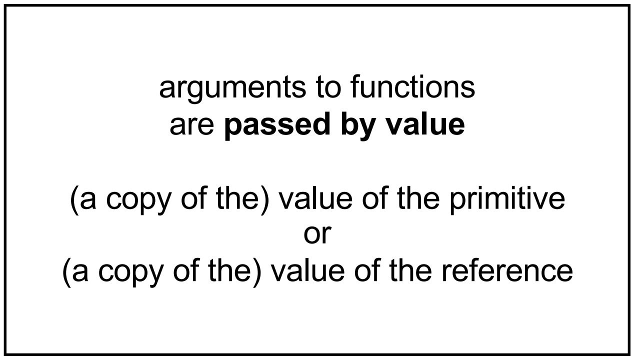
- the **garbage collector** is like a trash truck that drives around in the heap; when it notices an object that your program no longer has any references to, it frees up that memory for future use
  - 🌊 C does NOT have a garbage collector;
    in C you free heap-allocated memory yourself by calling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



[review all examples at least one more time]

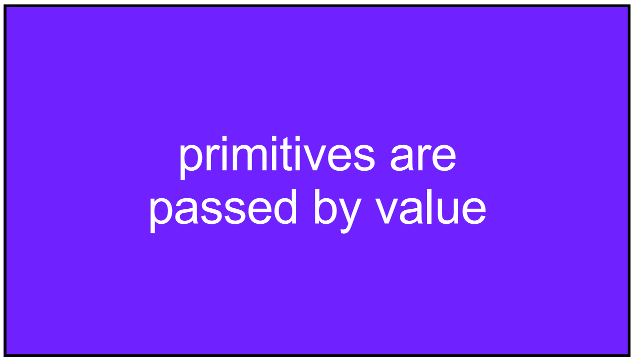


passing arguments to functions

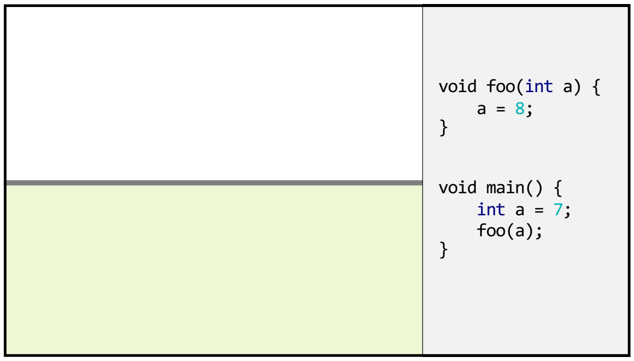


arguments to functions
are **passed by value**

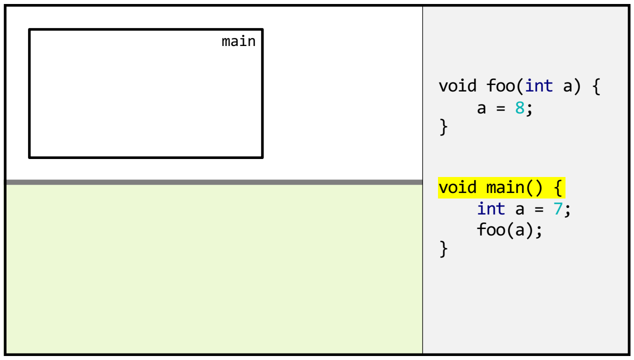
(a copy of the) value of the primitive
or
(a copy of the) value of the reference



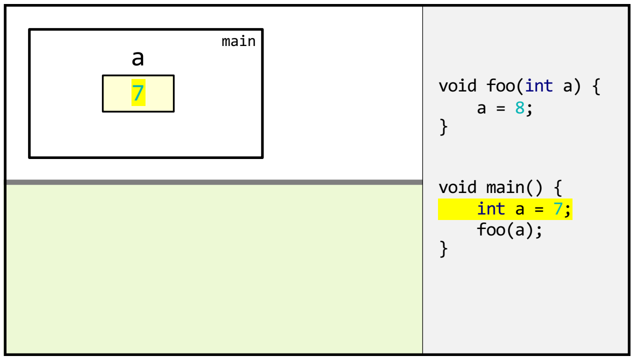
primitives are passed by value



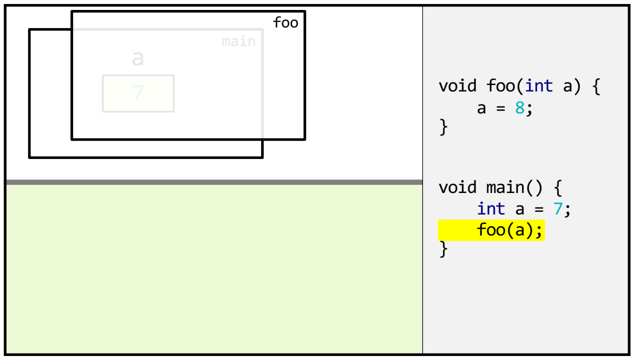
```
void foo(int a) {
    a = 8;
}

void main() {
    int a = 7;
    foo(a);
}
```



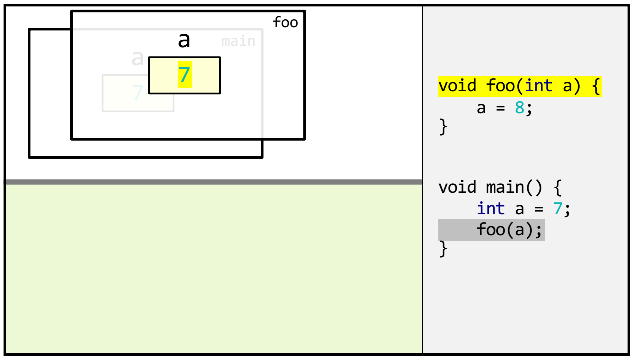
```
void foo(int a) {
    a = 8;
}

void main() {
    int a = 7;
    foo(a);
}
```



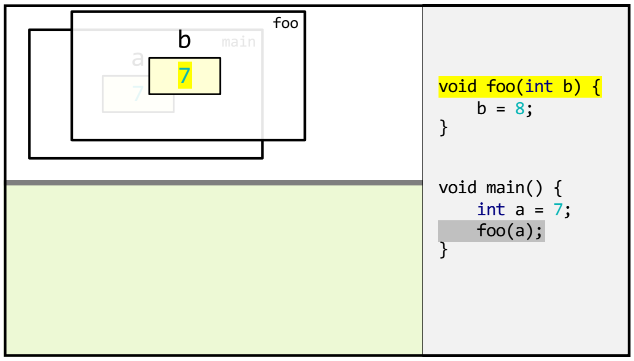
```
void foo(int a) {
    a = 8;
}

void main() {
    int a = 7;
    foo(a);
}
```



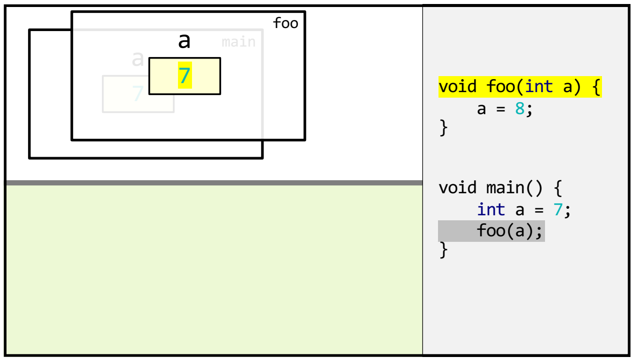
```
void foo(int a) {
    a = 8;
}

void main() {
    int a = 7;
    foo(a);
}
```



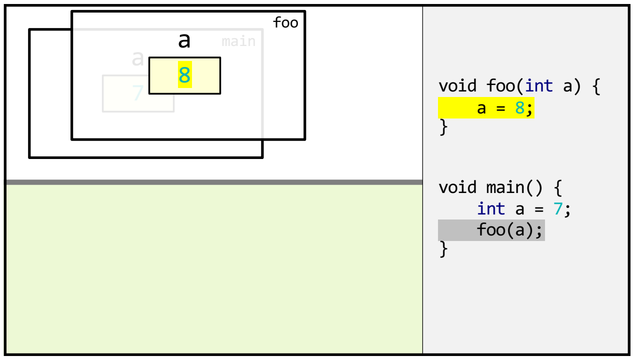
```
void foo(int a) {
    a = 8;
}

void main() {
    int a = 7;
    foo(a);
}
```



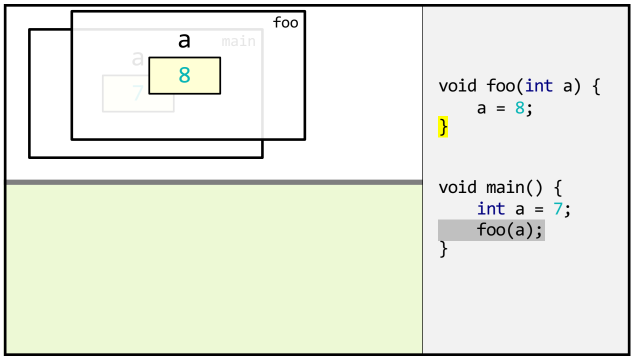
```
void foo(int b) {
    b = 8;
}

void main() {
    int a = 7;
    foo(a);
}
```



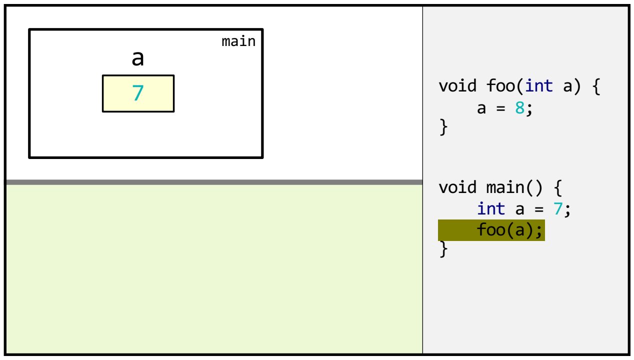
```
void foo(int a) {
    a = 8;
}

void main() {
    int a = 7;
    foo(a);
}
```



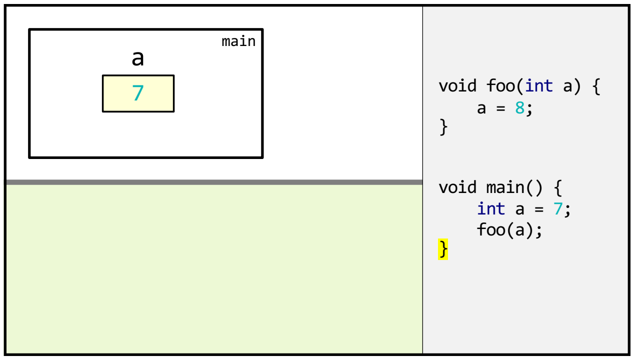
```
void foo(int a) {
    a = 8;
}

void main() {
    int a = 7;
    foo(a);
}
```



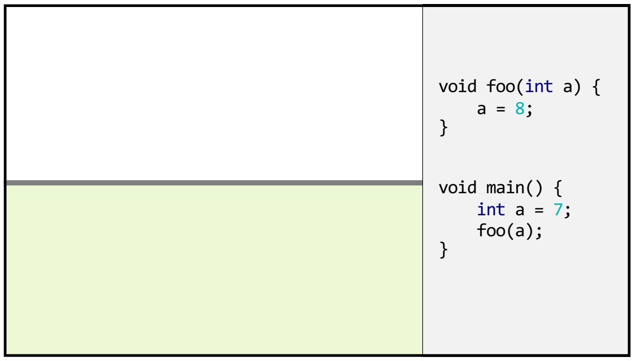
```
void foo(int a) {
    a = 8;
}

void main() {
    int a = 7;
    foo(a);
}
```



```
void foo(int a) {
    a = 8;
}

void main() {
    int a = 7;
    foo(a);
}
```

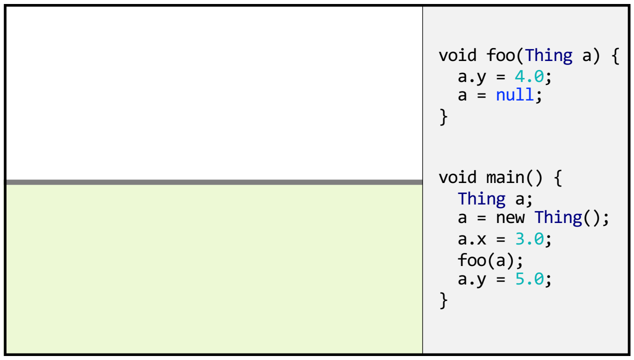


```
void foo(int a) {
    a = 8;
}

void main() {
    int a = 7;
    foo(a);
}
```

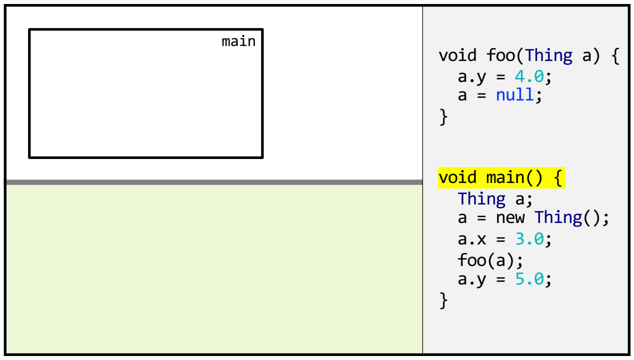


```
void foo(int a) {
    a = 8;
}

void main() {
    int a = 7;
    foo(a);
}
```

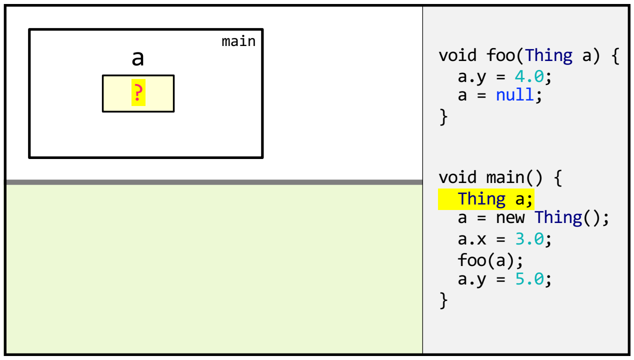
references are also passed by value



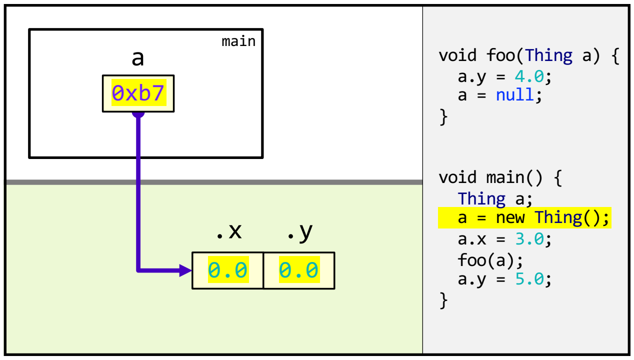
```
void foo(Thing a) {
  a.y = 4.0;
  a = null;
}

void main() {
  Thing a;
  a = new Thing();
  a.x = 3.0;
  foo(a);
  a.y = 5.0;
}
```



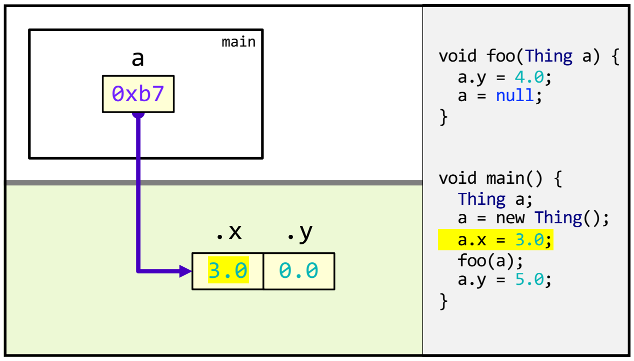
```
void foo(Thing a) {
  a.y = 4.0;
  a = null;
}

void main() {
  Thing a;
  a = new Thing();
  a.x = 3.0;
  foo(a);
  a.y = 5.0;
}
```



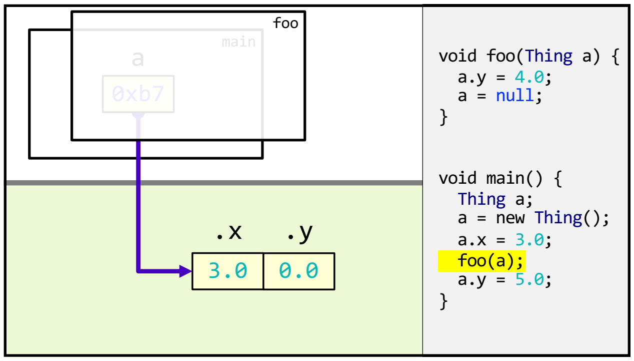
```
void foo(Thing a) {
  a.y = 4.0;
  a = null;
}

void main() {
  Thing a;
  a = new Thing();
  a.x = 3.0;
  foo(a);
  a.y = 5.0;
}
```



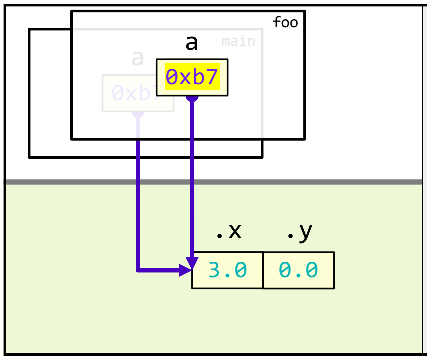
```
void foo(Thing a) {
  a.y = 4.0;
  a = null;
}

void main() {
  Thing a;
  a = new Thing();
  a.x = 3.0;
  foo(a);
  a.y = 5.0;
}
```



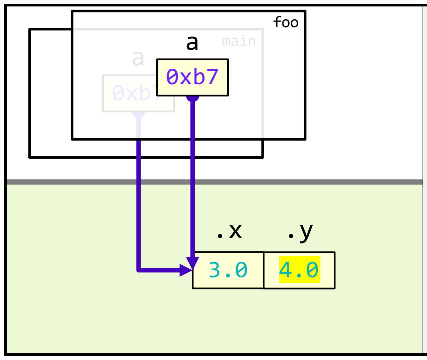
```
void foo(Thing a) {
  a.y = 4.0;
  a = null;
}

void main() {
  Thing a;
  a = new Thing();
  a.x = 3.0;
  foo(a);
  a.y = 5.0;
}
```



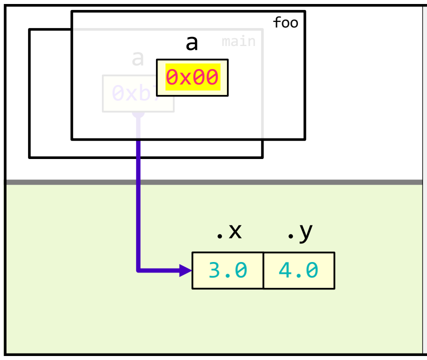
```
void foo(Thing a) {
  a.y = 4.0;
  a = null;
}

void main() {
  Thing a;
  a = new Thing();
  a.x = 3.0;
  foo(a);
  a.y = 5.0;
}
```



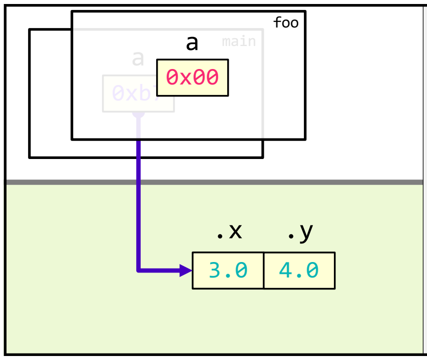
```
void foo(Thing a) {
  a.y = 4.0;
  a = null;
}

void main() {
  Thing a;
  a = new Thing();
  a.x = 3.0;
  foo(a);
  a.y = 5.0;
}
```



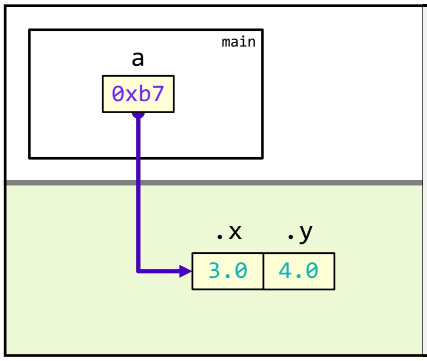
```
void foo(Thing a) {
  a.y = 4.0;
  a = null;
}

void main() {
  Thing a;
  a = new Thing();
  a.x = 3.0;
  foo(a);
  a.y = 5.0;
}
```



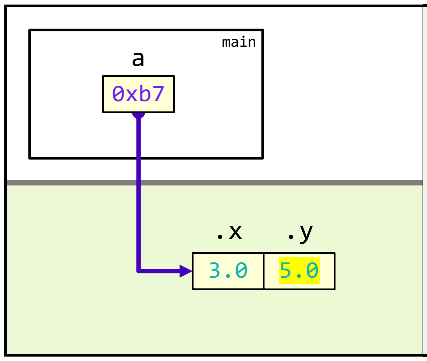
```
void foo(Thing a) {
  a.y = 4.0;
  a = null;
}

void main() {
  Thing a;
  a = new Thing();
  a.x = 3.0;
  foo(a);
  a.y = 5.0;
}
```



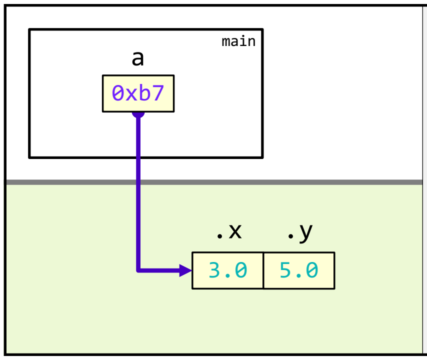
```
void foo(Thing a) {
  a.y = 4.0;
  a = null;
}

void main() {
  Thing a;
  a = new Thing();
  a.x = 3.0;
  foo(a);
  a.y = 5.0;
}
```



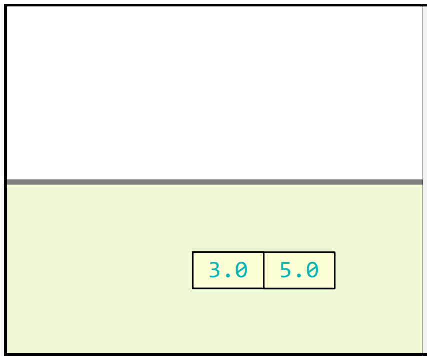
```
void foo(Thing a) {
  a.y = 4.0;
  a = null;
}

void main() {
  Thing a;
  a = new Thing();
  a.x = 3.0;
  foo(a);
  a.y = 5.0;
}
```



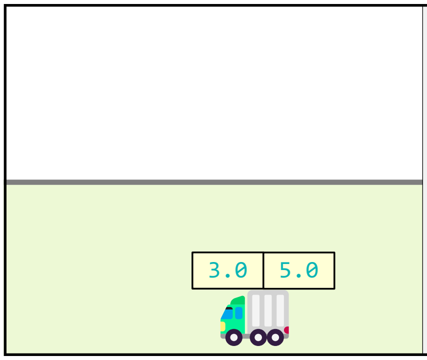
```
void foo(Thing a) {
  a.y = 4.0;
  a = null;
}

void main() {
  Thing a;
  a = new Thing();
  a.x = 3.0;
  foo(a);
  a.y = 5.0;
}
```



```
void foo(Thing a) {
  a.y = 4.0;
  a = null;
}

void main() {
  Thing a;
  a = new Thing();
  a.x = 3.0;
  foo(a);
  a.y = 5.0;
}
```



```
void foo(Thing a) {
  a.y = 4.0;
  a = null;
}

void main() {
  Thing a;
  a = new Thing();
  a.x = 3.0;
  foo(a);
  a.y = 5.0;
}
```



```
void foo(Thing a) {
  a.y = 4.0;
  a = null;
}

void main() {
  Thing a;
  a = new Thing();
  a.x = 3.0;
  foo(a);
  a.y = 5.0;
}
```

today is Jim codes at you for 50 minutes straight Friday