Objects: Where? Heap store

Objects are more permanent

- method called
- method creates object
- method returns
- object lives on

"objects" includes arrays

∴ can't store objects in frames

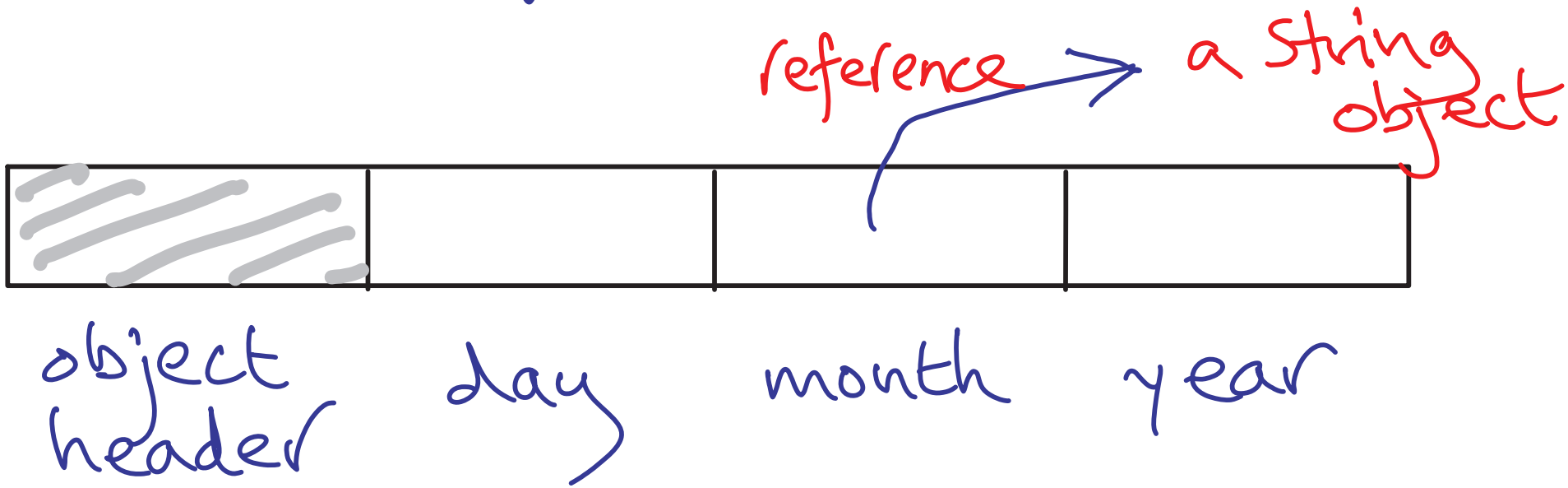
The memory where objects are stored is called the heap.

Example

```
public class Date{
    private int day;
    private String month;
    private int year;
```

instance variables (non-static fields) – stored in each instance (object) of class Date

Each instance of Date laid out as –



Memory contains :

- frames - seen already
- heap
  - objects (including arrays)
- method area
  - method bytecode
  - constant pools
  - class variables (static fields)

chain of linked frames is called a stack - don't confuse it with operand stacks

Stack

One record (frame) per method call

Parameters, local variables operand stacks

Heap

One record per object (instance)

Instance variables (non-static fields)

Method area

One record per class

Constant data : constant pool, method bytecode

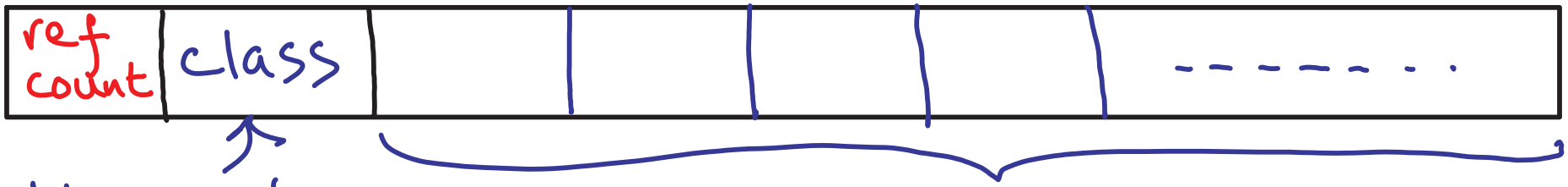
Class variables (static fields)

you'll see these later in SW workshop

An object

Exact layout depends on JVM

Number of references to this object ... explained later



address of method area for class of this object

values of instance variables (fields)

object header includes these

For local variables:
use iload, fload etc.

Accessing instance variables using bytecode

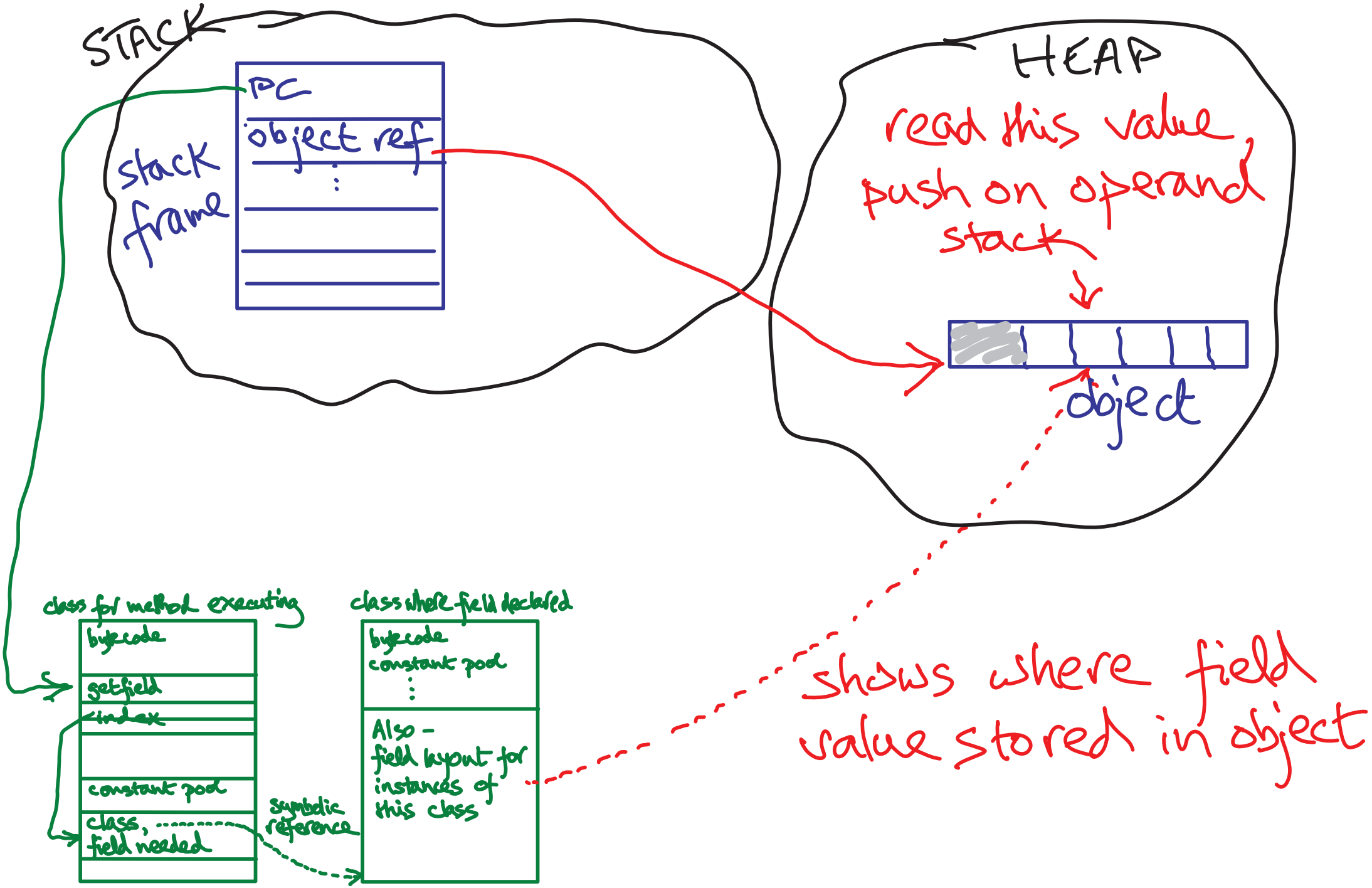
getfield 2-byte index

..., object ref ⟹ ..., value of variable

pushes variable value onto operand stack

index is index into constant pool

- object reference is address of object
- constant pool entry is symbolic reference - names of instance variable + class declaring it


STACK
PC
object ref
stack frame
HEAP
read this value, push on operand stack
object
class for method executing
bytecode
getfield
index
constant pool
class, field needed
symbolic reference
class where field declared
bytecode
constant pool
Also - field layout for instances of this class
shows where field value stored in object

Accessing fields (non-static) using bytecode

putfield 2-byte index

is similar, to pop value from operand stack:

...., object ref, new field value ⇒ ...

Also getstatic, putstatic (understand these later)

for static fields (class variables)

– but no object ref used for these

Calling an instance method

non-static

invokevirtual 2-byte index

Similar to invokestatic for static methods but operand stack has ref to "this" object that will execute method

this..

..., objectref, actual params ⟹ ...

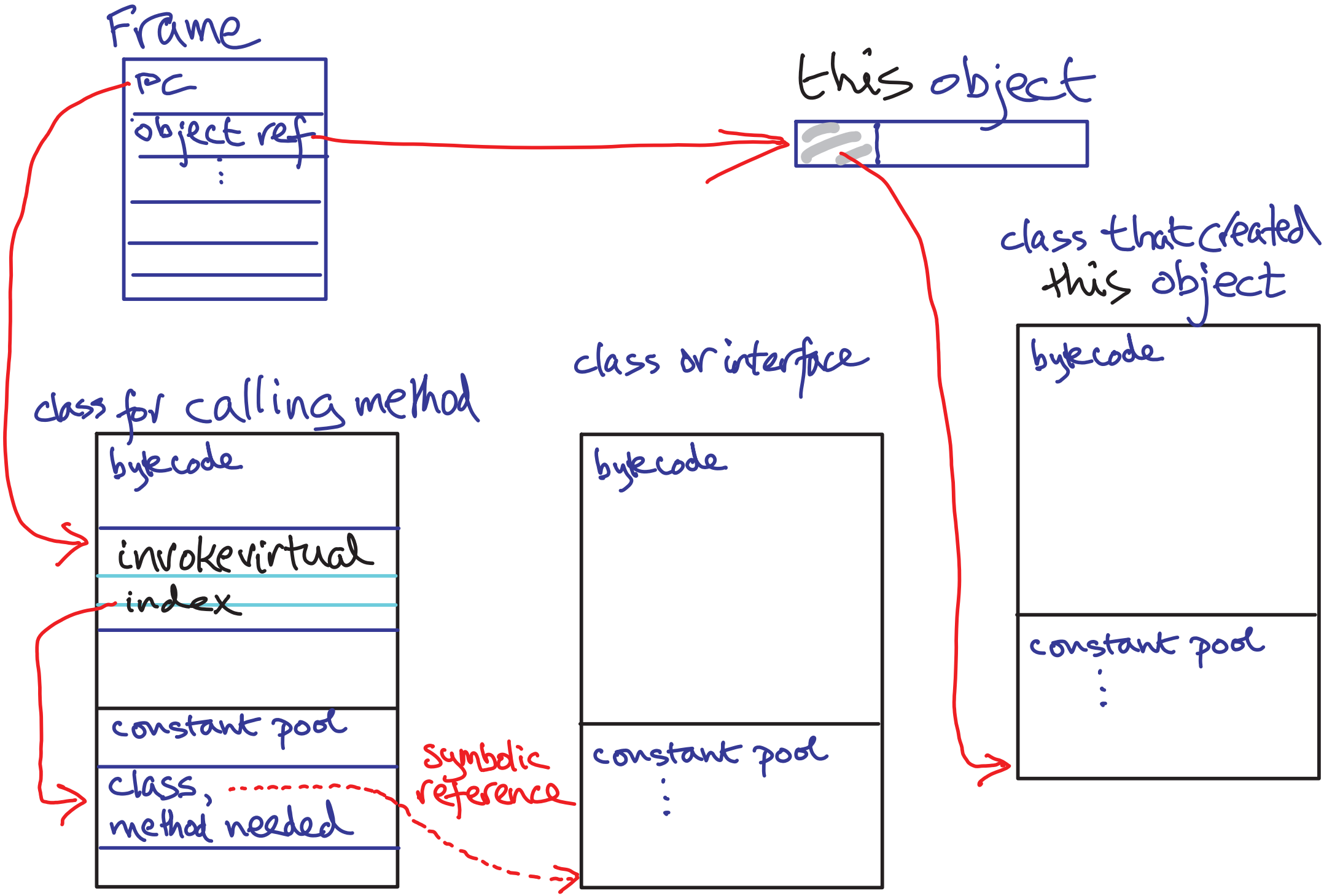
copied to variable #0 in frame

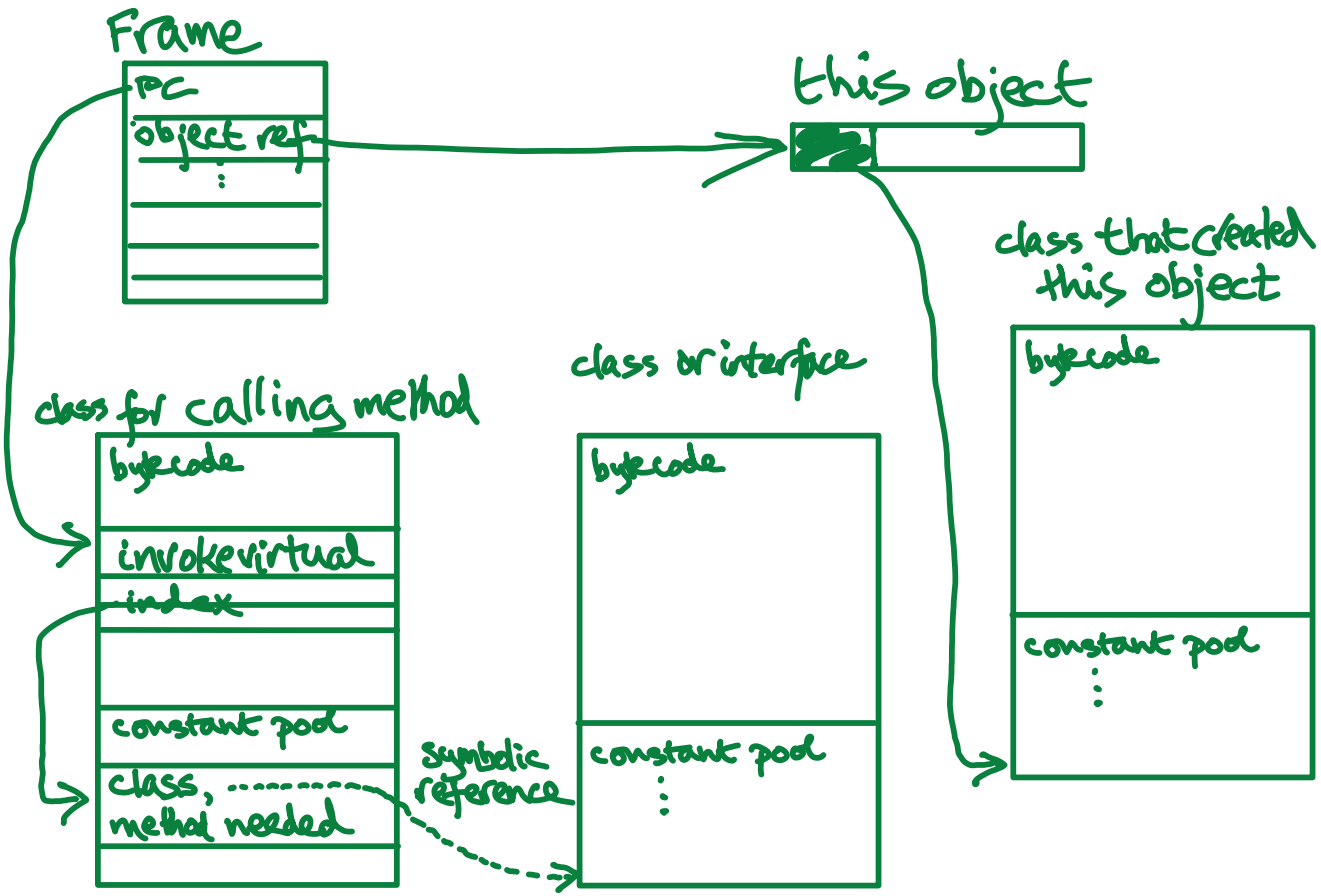
Method calls – static

invokestatic 2-byte index

- index is used as index into constant pool of current class of method currently being executed
- constants pool entry must be for a method. From it can be found
  - number & types of parameters & result
  - address of bytecode for method
  - size of operand stack, number of local variables } needed

access permission also checked


Frame
PC
object ref
this object
class that created this object
bytecode
constant pool
class or interface
bytecode
constant pool
class for calling method
bytecode
invokevirtual
index
constant pool
class, method needed
symbolic reference



Class of object: found at run-time

Invoice

These two might be different!

Class or interface for declared type of a variable. Known by compiler

Payable

e.g.

```
Payable x = new Invoice(29, 6,
      "printer cartridge",
      "222333", 31);
int a = x.paymentAmount();
```

<u>Rule</u> - of "method dispatch"

It is always the class of the object that governs which method definition is chosen.

- This makes a difference if the type of the variable is itself a class.
- It means the method must be chosen at run-time - the compiler can't work it out
- C++ is different! But the Java rule is secure.

Which method definition?

Every class extends Object

e.g. toString

- If you don't define toString:
  there's a default definition
  (in Object class)
  Your class inherits the default

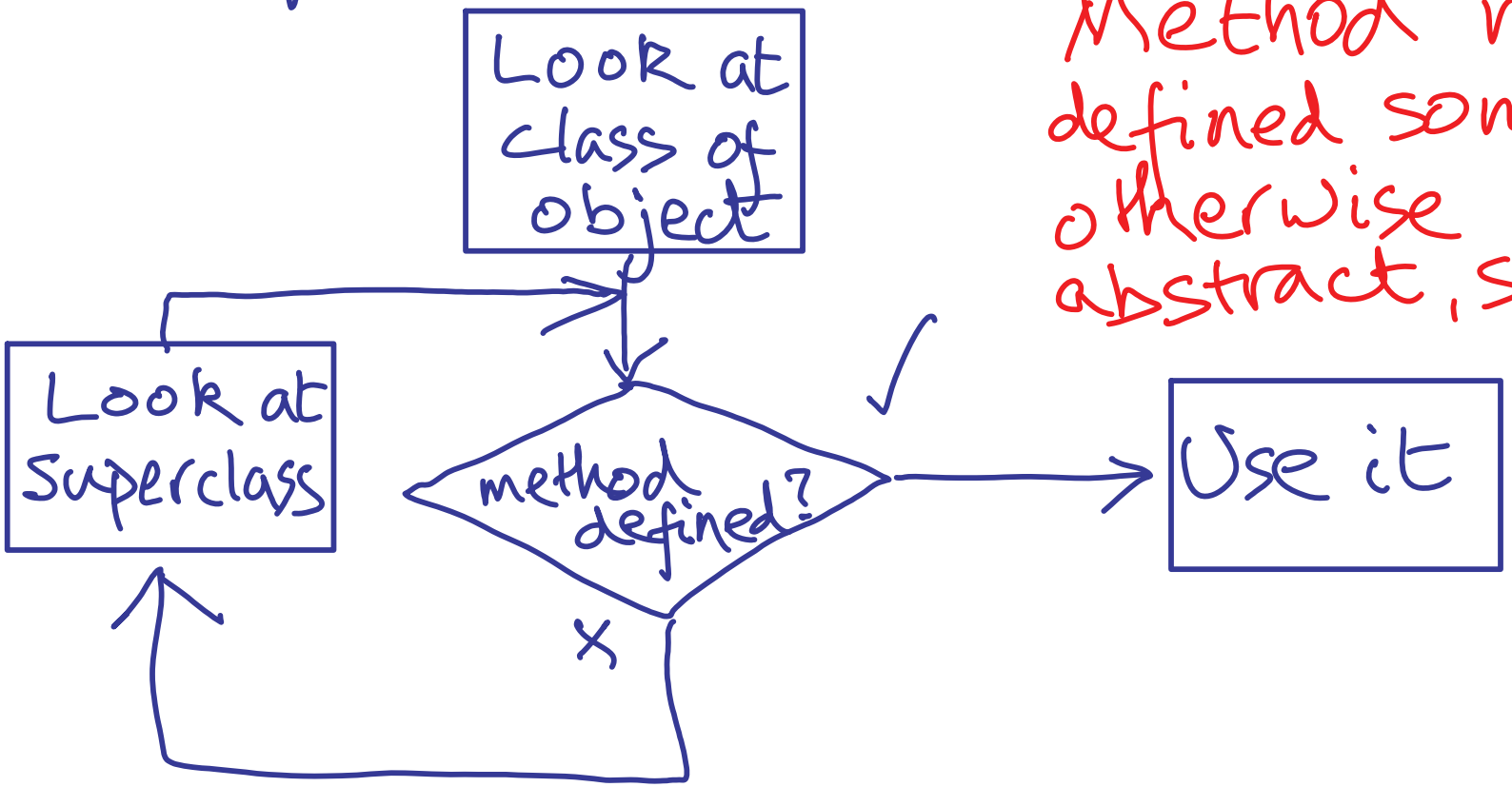
- If you do:
  your definition overrides the default

Chains of overriding

see this later

- arise from class extension (subclasses)

invokevirtual must find correct definition (that overrides all the others)



Method must be defined somewhere - otherwise class is abstract, so can't have instances. (compiler enforces this.)

Creating an object in bytecode

Two steps

1. Allocate space, with default values
2. Execute constructor to do proper initialization

Creating an object in bytecode

① Allocate space

new 2-byte index

- index is to constant pool entry with name of class for new object
- finds class definition
- allocates heap space for object with correct fields, initialized to defaults
- pushes reference to new object

0 for numbers
null for refs
etc

.... ⟹ ...., object ref

Creating an object in bytecode

② Execute constructor

invokespecial 2-byte index

- index into constant pool gives entry for constructor
- expects ref to new object on stack

..., objectref, parameters ⇒ ...

cf. invokevirtual

①② usually done as

```
new ...
dup
invokespecial
```

dup ← duplicates top of stack so objectref still on stack at end

e.g.

```
Date myBirthday = new Date(36,"Julember", 2015);

  0: new            #13     // class introcs/Date
  3: dup
  4: bipush         36
  6: ldc            #14     // String Julember
  8: sipush         2015
 11: invokespecial #15     // Method
                  "<init>" (ILjava/lang/String;I)V
 14: astore_1
```

= constructor

int string int

three parameters

e.g.

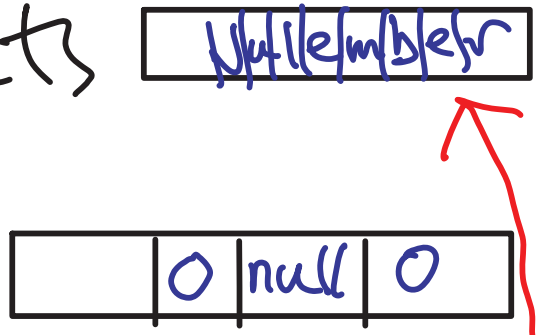
```
Date myBirthday = new Date(36,"Julember", 2015);
```

```
 0: new           #13     // class introcs/Date
 3: dup
 4: bipush        36
 6: ldc           #14     // String Julember
 8: sipush        2015
11: invokespecial #15     // Method
                     "<init>" (ILjava/lang/String;I)V
14: astore_1
```

line	loc.var. myBirthday	opnd stack
start	null	empty
0	null	ref
3	null	ref, ref
4	null	ref, ref, 36
6	null	ref, ref, 36, ref
8	null	ref, ref, 36, ref, 2015

parameters for constructor

objects



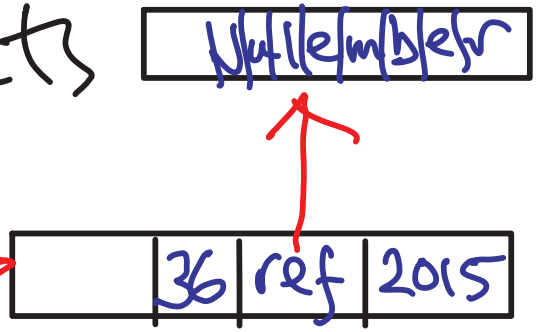
e.g.

```
Date myBirthday = new Date(36,"Julember", 2015);
```

```
 0: new           #13    // class introcs/Date
 3: dup
 4: bipush        36
 6: ldc           #14    // String Julember
 8: sipush        2015
11: invokespecial #15    // Method
                    "<init>" (ILjava/lang/String;I)V
14: astore_1
```

line	loc.var. myBirthday	opnd stack
start	null	empty
11	null	ref
14	ref	empty

objects



same object as before but instance variables updated

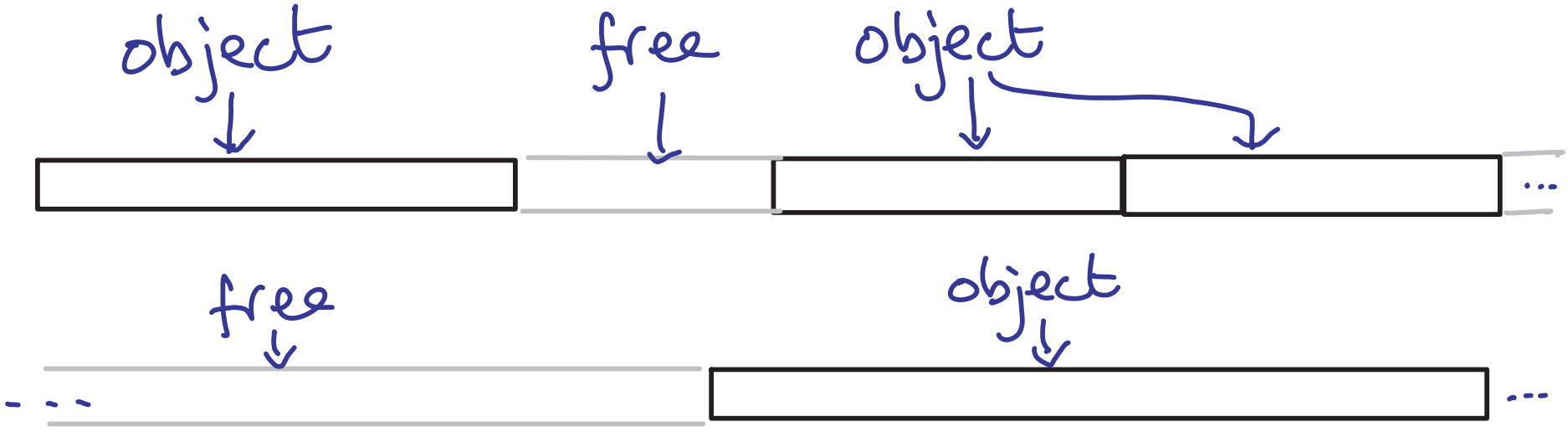
Verifier checks on object creation

1. new always followed by appropriate invokespecial
2. extra checks associated with class extension

⇒ object always created consistent with what its designers intend

Heap

Some parts used for objects, some parts free



When new object is created: space found in free areas.

When object is destroyed: its space made free again

Garbage collection

In some languages (e.g. C++)
you destroy objects explicitly
when you have finished with them.
Then their storage area can be recycled
In Java you don't.
The garbage collector is part of the JVM.
It works out when objects are finished with
& then recycles storage area –
marks it "free"

When is an object finished with?

Each object is like Isla de Muerta in Pirates of the Caribbean:

"It's an island that cannot be found, except by those who already know where it is."

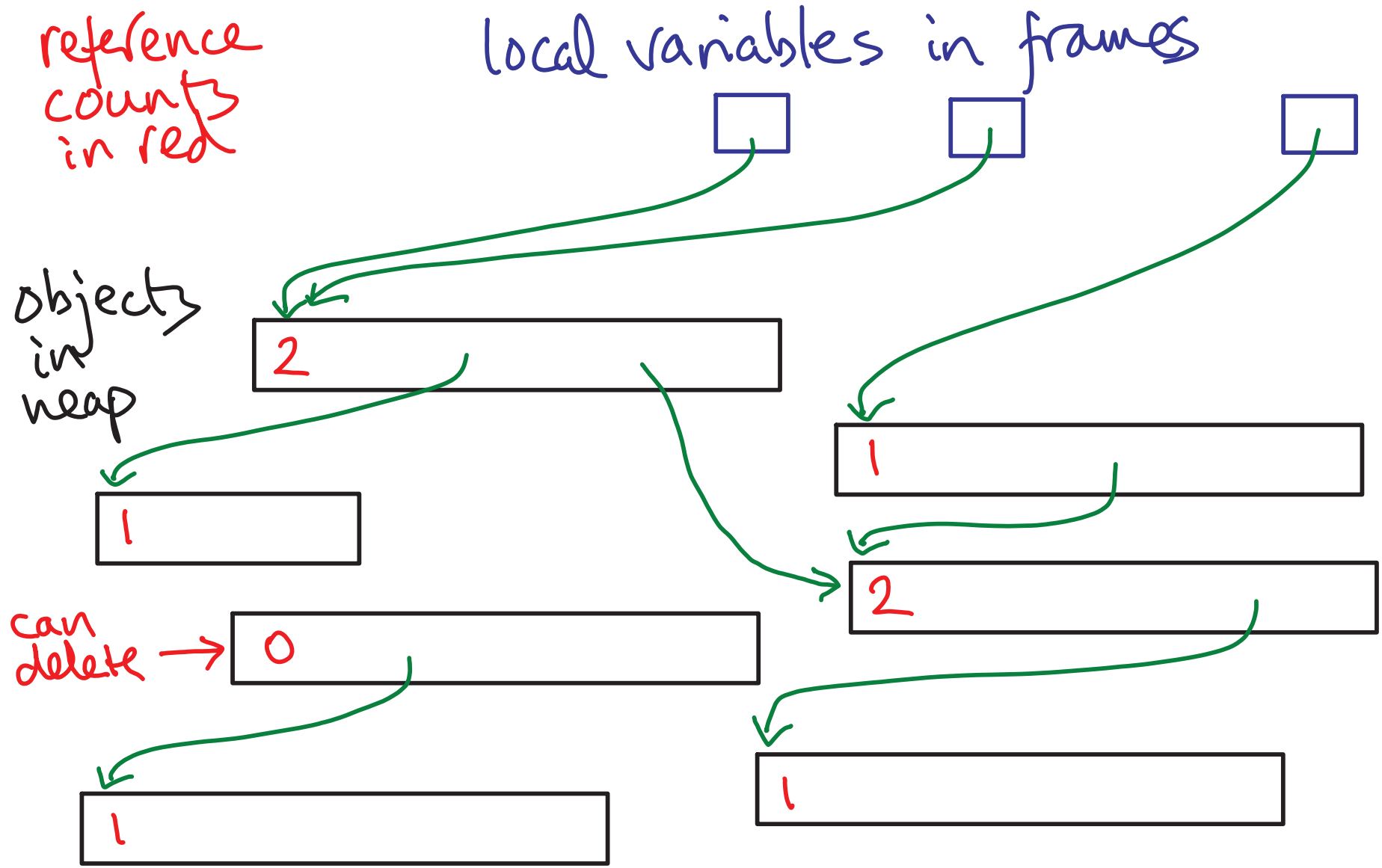
To know where an object is:

have a reference to it.

When there are no references to an object:

it cannot be used any more.

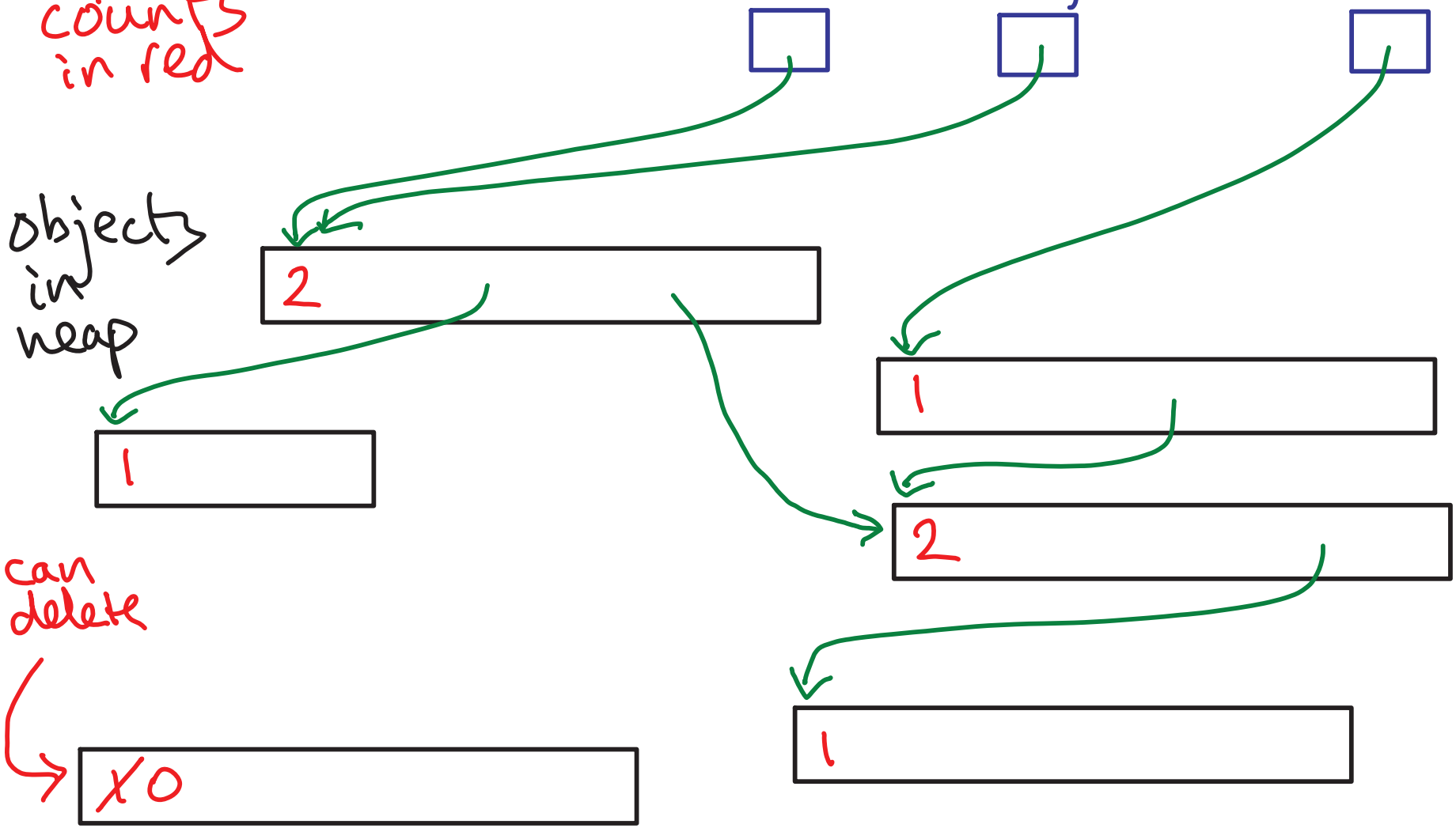
∴ garbage collector can safely delete it.


reference counts in red
local variables in frames
objects in heap
2
1
1
2
can delete
0
1
1

reference counts in red

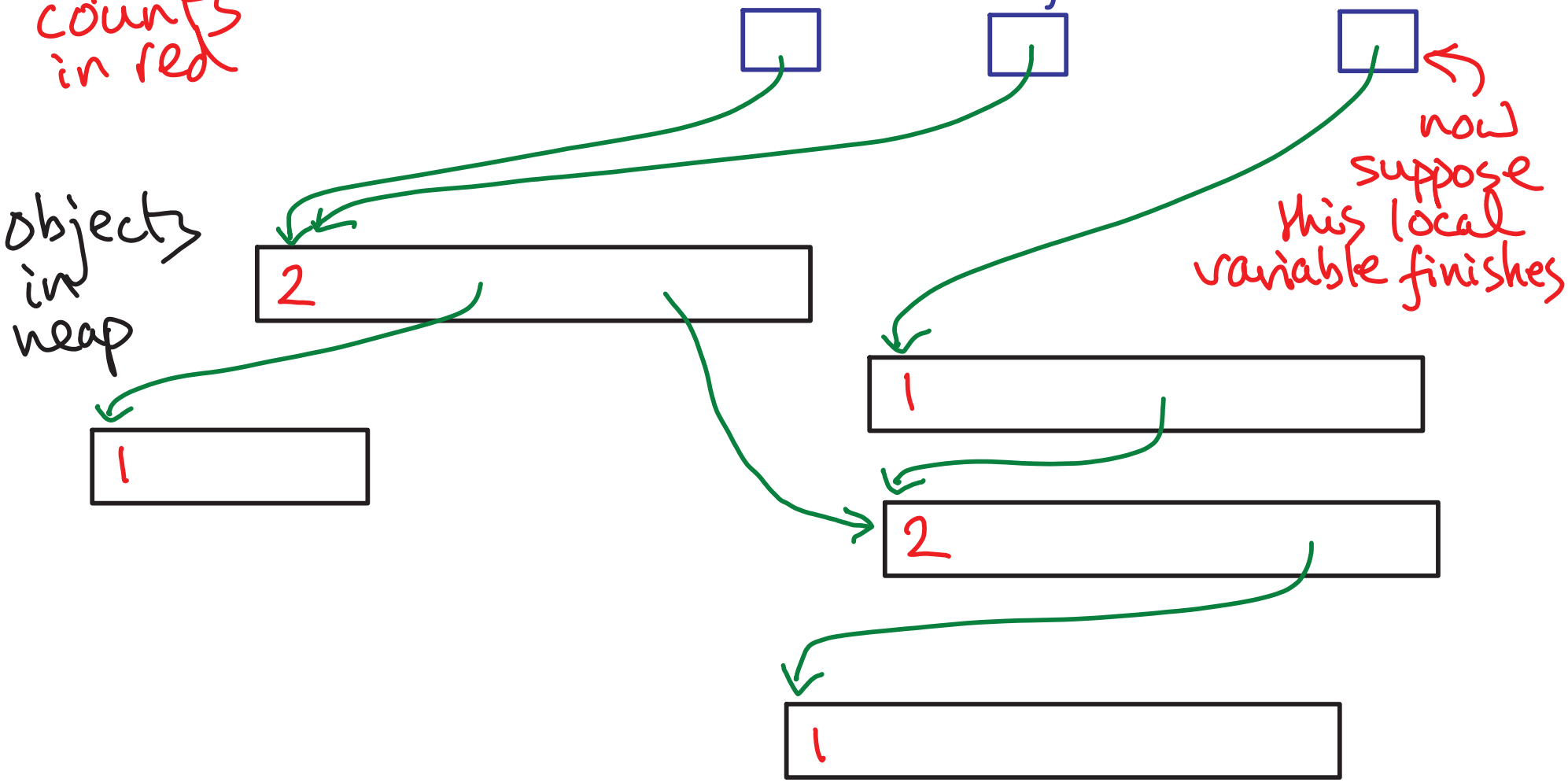
local variables in frames

objects in heap

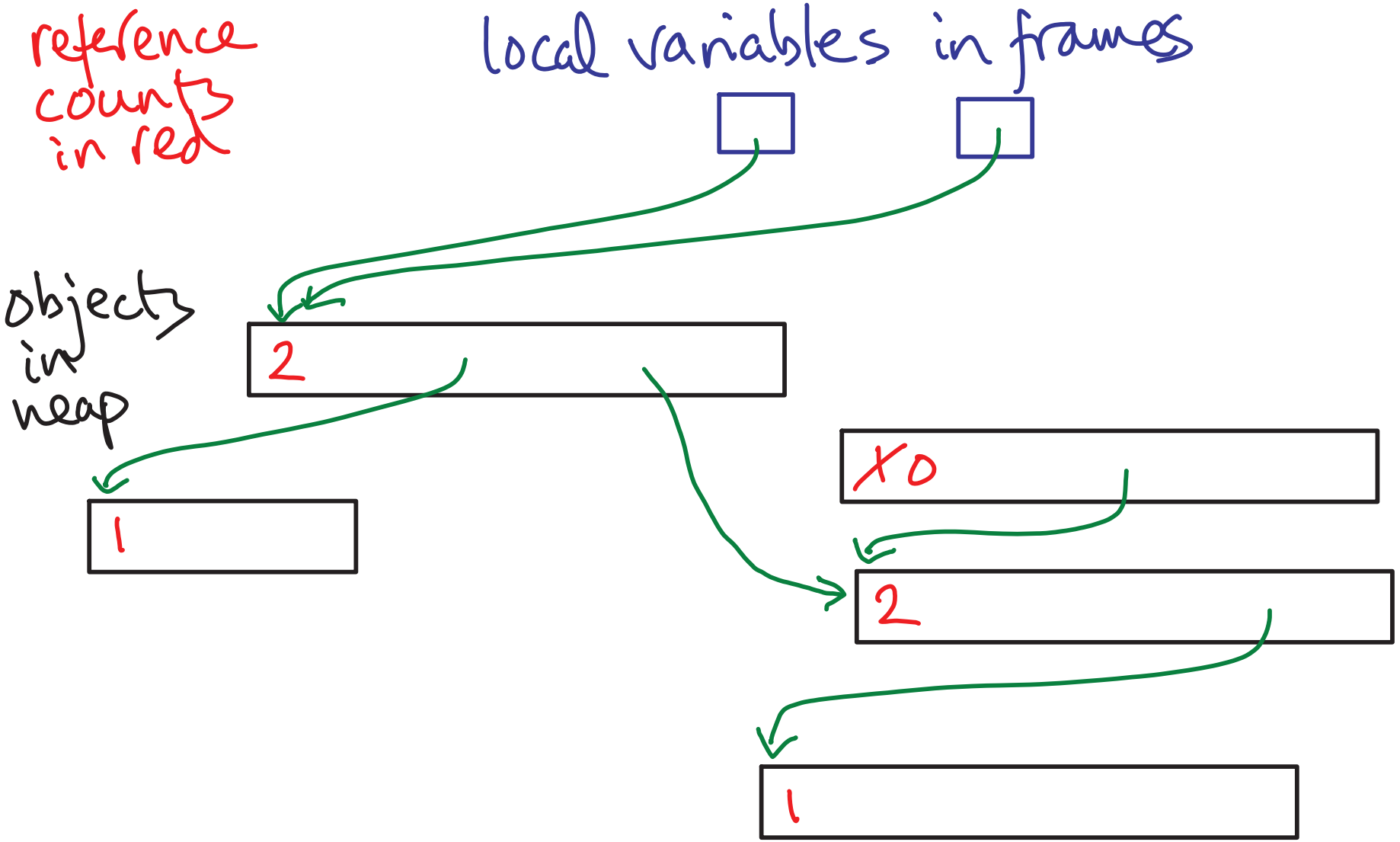


reference counts in red

local variables in frames



reference counts in red



reference counts in red