Object Oriented Programming

Dr Robert Harle

IA CST, PBST (CS) and NST (CS)

Michaelmas 2016/17

The OOP Course

- So far you have studied some procedural programming in Java and functional programming in ML
- Here we take your procedural Java and build on it to get object-oriented Java
- You have ticks in Java
  - This course ***complements*** the practicals
  - Some material appears only here
  - Some material appears only in the practicals
  - Some material appears in both: deliberately*!

* Some material may be repeated unintentionally. If so I will claim it was deliberate.

Outline

1. Types, Objects and Classes
2. Designing Classes
3. Pointers, References and Memory
4. Inheritance
5. Polymorphism
6. Lifecycle of an Object
7. Error Handling
8. Copying Objects
9. Java Collections
10. Object Comparison
11. Design Patterns
12. Design Pattern (cont.)

Books and Resources I

- OOP Concepts
  - Look for books for those learning to first program in an OOP language (Java, C++, Python)
  - *Java: How to Program* by Deitel & Deitel (also C++)
  - *Thinking in Java* by Eckels
  - *Java in a Nutshell* (O' Reilly) if you already know another OOP language
  - Java specification book: http://java.sun.com/docs/books/jls/
  - Lots of good resources on the web

- Design Patterns
  - *Design Patterns* by Gamma et al.
  - Lots of good resources on the web

Books and Resources II

- Also check the course web page
  - Updated notes (with annotations where possible)
  - Code from the lectures
  - Sample tripos questions

http://www.cl.cam.ac.uk/teaching/current/OOProg/

- **And the Moodle site (which you'll be enrolled on automatically today)**

Lecture 1:
Types, Objects and Classes

Types of Languages

- Declarative - specify <u>what</u> to do, not <u>how</u> to do it. i.e.
  - E.g. HTML describes what should appear on a web page, and not how it should be drawn to the screen
  - E.g. SQL statements such as "select * from table" tell a program to get information from a database, but not how to do so
- Imperative – specify <u>both</u> what and how
  - E.g. "triple x" might be a declarative instruction that you want the variable x tripled in value. Imperatively we would have "x=x*3" or "x=x+x+x"

Top 20 Languages 2016

Oct 2016	Oct 2015	Change	Programming Language	Ratings	Change
1	1		Java	18.799%	-0.74%
2	2		C	9.835%	-6.35%
3	3		C++	5.797%	+0.05%
4	4		C#	4.367%	-0.46%
5	5		Python	3.775%	-0.74%
6	8	^	JavaScript	2.751%	+0.46%
7	6	v	PHP	2.741%	+0.18%
8	7	v	Visual Basic .NET	2.660%	+0.20%
9	9		Perl	2.495%	+0.25%
10	14	^^	Objective-C	2.263%	+0.84%
11	12	^	Assembly language	2.232%	+0.66%
12	15	^	Swift	2.004%	+0.73%
13	10	v	Ruby	2.001%	+0.18%
14	13	v	Visual Basic	1.987%	+0.47%
15	11	vv	Delphi/Object Pascal	1.875%	+0.24%
16	65	^^	Go	1.809%	+1.67%
17	32	^^	Groovy	1.769%	+1.19%
18	20	^	R	1.741%	+0.75%
19	17	v	MATLAB	1.619%	+0.46%
20	18	v	PL/SQL	1.531%	+0.46%

Top 20 Languages 2016 (Cont)

Position	Programming Language	Ratings
21	SAS	1.443%
22	ABAP	1.257%
23	Scratch	1.132%
24	COBOL	1.127%
25	Dart	1.099%
26	D	1.047%
27	Lua	0.827%
28	Fortran	0.742%
29	Lisp	0.742%
30	Transact-SQL	0.721%
31	Ada	0.652%
32	F#	0.633%
33	Scala	0.611%
34	Haskell	0.522%
35	Logo	0.500%
36	Prolog	0.495%
37	LabVIEW	0.455%
38	Scheme	0.444%
39	Apex	0.349%
40	Q	0.303%

Top 20 Languages 2016 (Cont Cont)

41	Erlang	0.300%
42	Rust	0.296%
43	Bash	0.286%
44	RPG (OS/400)	0.273%
45	Ladder Logic	0.266%
46	VHDL	0.220%
47	Alice	0.205%
48	Awk	0.203%
49	CL (OS/400)	0.170%
50	Clojure	0.169%

Top 20 Languages 2016 (Cont Cont Cont)

The Next 50 Programming Languages

The following list of languages denotes #51 to #100. Since the differences are relatively small, the programming languages are only listed (in alphabetical order).

- (Visual) FoxPro, 4th Dimension/4D, ABC, ActionScript, APL, AutoLISP, bc, BlitzMax, Bourne shell, C shell, CFML, cg, Common Lisp, Crystal, Eiffel, Elixir, Elm, Forth, Hack, Icon, IDL, Inform, Io, J, Julia, Korn shell, Kotlin, Maple, ML, MQL4, MS-DOS batch, NATURAL, NXT-G, OCaml, OpenCL, Oz, Pascal, PL/I, PowerShell, REXX, S, Simulink, Smalltalk, SPARK, SPSS, Stand[illegible] Stata, Tcl, VBScript, Verilog

ML as a Functional Language

- **Functional** languages are a subset of declarative languages
  - ML is a functional language
  - It may appear that you tell it how to do everything, but you should think of it as providing an explicit example of what should happen
  - The compiler may optimise i.e. replace your implementation with something entirely different but 100% equivalent.

Function Side Effects

- Functions in imperative languages can use or alter larger system state → *procedures*

Maths: $m(x,y) = xy$

ML: `fun m(x,y) = x*y;`

Java: `int m(int x, int y) = x*y;`

```
int y = 7;
int m(int x) {
        y=y+1;
        return x*y;
}
```

void Procedures

- A void procedure returns nothing:

```
int count=0;

void addToCount() {
  count=count+1;
}
```

Control Flow: Looping

for(*initialisation; termination; increment* **)**

```
for (int i=0; i<8; i++) …

int j=0;  for(; j<8; j++) …

for(int k=7;k>=0; j--) …
```

while(*boolean_expression* **)**

```
int i=0;  while (i<8) { i++; …}

int j=7; while (j>=0) { j--; …}
```

Control Flow: Looping Examples

```
int arr[] = {1,2,3,4,5};

for (int i=0; i<arr.length;i++) {
        System.out.println(arr[i]);
}

int i=0;
while (i<arr.length) {
        System.out.println(arr[i]);
        i=i+1;
}
```

Control Flow: Branching I

- Branching statements interrupt the current control flow
- **return**
  - Used to return from a function at any point

```
boolean linearSearch(int[] xs, int v) {
   for (int i=0;i<xs.length; i++) {
      if (xs[i]==v) return true;
   }
   return false;
}
```

Control Flow: Branching II

- Branching statements interrupt the current control flow
- **break**
  - Used to jump out of a loop

```
boolean linearSearch(int[] xs, int v) {
   boolean found=false;
   for (int i=0;i<xs.length; i++) {
      if (xs[i]==v) {
            found=true;
            break;   // stop looping
      }
   }
   return found;
}
```

Control Flow: Branching III

- Branching statements interrupt the current control flow
- **continue**
  - Used to skip the current iteration in a loop

```
void printPositives(int[] xs) {

   for (int i=0;i<xs.length; i++) {
      if (xs[i]<0) continue;
      System.out.println(xs[i]);
   }
}
```

Immutable to Mutable Data

ML

```
- val x=5;
> val x = 5 : int
- x=7;
> val it = false : bool
- val x=9;
> val x = 9 : int
```

Java

```
int x=5;
x=7;

int x=9;
```

Types and Variables

- Most imperative languages don't have type inference

```
int x = 512;
int y = 200;
int z = x+y;
```

- The high-level language has a series of *primitive* (built-in) types that we use to signify what's in the memory
    - The compiler then knows what to do with them
    - E.g. An "int" is a primitive type in C, C++, Java and many languages. It's usually a 32-bit signed integer
- A variable is a name used in the code to refer to a specific instance of a type
    - x,y,z are variables above
    - They are all of type int

E.g. Primitive Types in Java

- “Primitive” types are the built in ones.
  - They are building blocks for more complicated types that we will be looking at soon.
- boolean – 1 bit (true, false)
- char – 16 bits
- byte – 8 bits as a signed integer (-128 to 127)
- short – 16 bits as a signed integer
- int – 32 bits as a signed integer
- long – 64 bits as a signed integer
- float – 32 bits as a floating point number
- double – 64 bits as a floating point number

Overloading Functions

- Same function name
- Different arguments
- Possibly different return type

```
int myfun(int a, int b) {...}
float myfun(float a, float b) {...}
double myfun(double a, double b) {...}
```

- But <u>not</u> just a different return type

```
int myfun(int a, int b) {...}
float myfun(int a, int b) {...}
```

X

Function Prototypes

- Functions are made up of a prototype and a body
  - Prototype specifies the function name, arguments and possibly return type
  - Body is the actual function code

```
fun myfun(a,b) = ...;

int myfun(int a, int b) {...}
```

Custom Types

```
datatype 'a seq = Nil
                | Cons of 'a * (unit -> 'a seq);
```

```
public class Vector3D {
   float x;
   float y;
   float z;
}
```

State and Behaviour

```
datatype 'a seq = Nil
                | Cons of 'a * (unit -> 'a seq);

fun hd (Cons(x,_)) = x;
```

State and Behaviour

```
datatype 'a seq = Nil
                | Cons of 'a * (unit -> 'a seq);

fun hd (Cons(x,_)) = x;




public class Vector3D {
   float x;
   float y;
   float z;

   void add(float vx, float vy, float vz) {
      x=x+vx;
      y=y+vy;
      z=z+vz;
   }
}
```

Loose Terminology (again!)

State
Fields
Instance Variables
Properties
Variables
Members

Behaviour
Functions
Methods
Procedures

Classes, Instances and Objects

- Classes can be seen as templates for representing various ***concepts***
- We create ***instances*** of classes in a similar way. e.g.

```
MyCoolClass m = new  MyCoolClass();
MyCoolClass n = new  MyCoolClass();
```

  makes two instances of class MyCoolClass.
- An instance of a class is called an **object**

Defining a Class

```
public class Vector3D {
   float x;
   float y;
   float z;

   void add(float vx, float vy, float vz) {
      x=x+vx;
      y=y+vy;
      z=z+vz;
   }
}
```

Constructors

```
MyObject m = new MyObject();
```

- You will have noticed that the RHS looks rather like a function call, and that's exactly what it is.
- It's a method that gets called when the object is constructed, and it goes by the name of a **constructor** (it's not rocket science). It maps to the datatype constructors you saw in ML.
- We use constructors to initialise the state of the class in a convenient way
  - A constructor has the same name as the class
  - A constructor has no return type

Constructors with Arguments

```
public class Vector3D {
    float x;
    float y;
    float z;

    Vector3D(float xi, float yi, float zi) {
        x=xi;
        y=yi;
        z=zi;
    }


    // ...
}
```

```
Vector3D v = new Vector3D(1.f,0.f,2.f);
```

Overloaded Constructors

```
public class Vector3D {
   float x;
   float y;
   float z;

   Vector3D(float xi, float yi, float zi) {
      x=xi;
      y=yi;
      z=zi;
   }

    Vector3D() {
      x=0.f;
      y=0.f;
      z=0.f;
   }

   // ...
}
```

```
Vector3D v = new Vector3D(1.f,0.f,2.f);
Vector3D v2 = new Vector3D();
```

Default Constructor

```
public class Vector3D {
   float x;
   float y;
   float z;
}

Vector3D v = new Vector3D();
```

- No constructor provided
- So blank one generated with no arguments

Class-Level Data and Functionality I

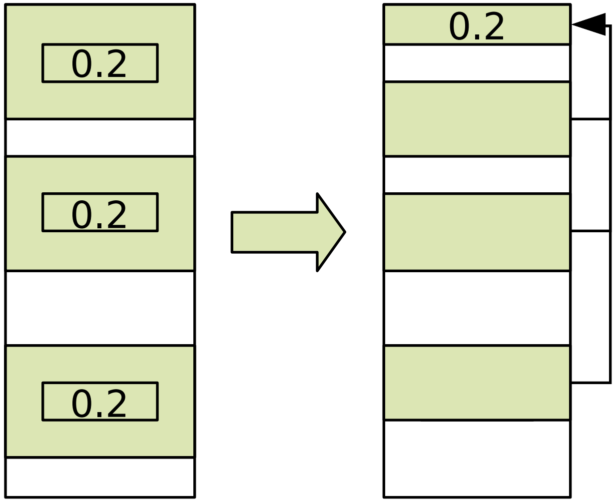
- A **static** field is created only once in the program's execution, despite being declared as part of a class

```
public class ShopItem {
   float mVATRate;
   static float sVATRate;
   ....
}
```

One of these created every time a new ShopItem is instantiated. Nothing keeps them all in sync.

Only one of these created ever. Every ShopItem object references it.

Class-Level Data and Functionality II



- Auto synchronised across instances
- Space efficient

- Also static methods:

```
public class Whatever {
   public static void main(String[] args) {
      ...
   }
}
```

Why use Static Methods?

- Easier to debug (only depends on static state)
- Self documenting
- Groups related methods in a Class without requiring an object
- The compiler can produce more efficient code since no specific object is involved

```
public class Math {
  public float sqrt(float x) {…}
  public double sin(float x) {…}
  public double cos(float x) {…}
}
```

```
…
Math mathobject = new Math();
mathobject.sqrt(9.0);
…
```

vs

```
public class Math {
  public static float sqrt(float x) {…}
  public static float sin(float x) {…}
  public static float cos(float x) {…}
}
```

```
…
Math.sqrt(9.0);
…
```

Lecture 2:
Designing Classes

What Not to Do

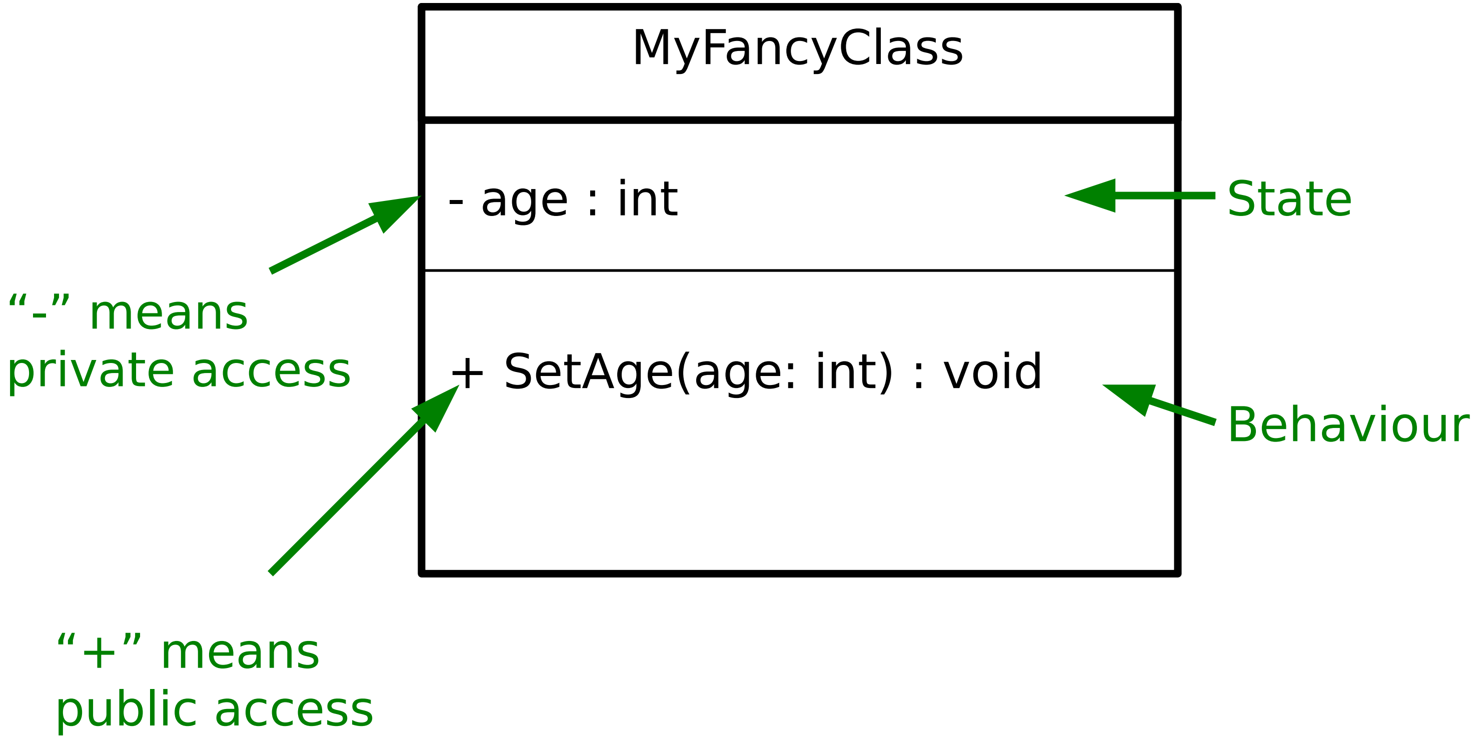
- Your ML has doubtless been one big file where you threw together all the functions and value declarations
- Lots of C programs look like this :-(
- We *could* emulate this in OOP by having one class and throwing everything into it
- We can do (much) better

Identifying Classes

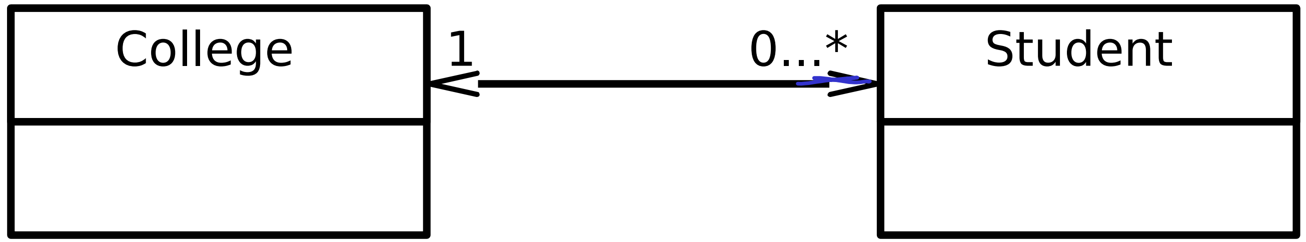
- We want our class to be a grouping of conceptually-related state and behaviour
- One popular way to group is using grammar
  - Noun → Object
  - Verb → Method

“A <u>simulation</u> of the <u>Earth</u>'s orbit around the <u>Sun</u>”

UML: Representing a Class Graphically



The has-a Association



- Arrow going left to right says "a College has zero or more students"
- Arrow going right to left says "a Student has exactly 1 College"
- What it means in real terms is that the College class will contain a variable that somehow links to a set of Student objects, and a Student will have a variable that references a College object.
- Note that we are only linking *classes*: we don't start drawing arrows to primitive types.

Anatomy of an OOP Program (Java)

Class name

```
public class MyFancyClass {

        public int someNumber;
        public String someText;

        public void someMethod() {

        }

        public static void main(String[] args) {
                MyFancyClass c = new
                        MyFancyClass();
        }

}
```

Access modifier

Class state (properties that an object has such as colour or size)

Class behaviour (actions an object can do)

'Magic' start point for the program (named main by convention)

Create a reference to a MyFancyClass object and call it c

Create an object of type MyFancyClass in memory

Anatomy of an OOP Program (C++)

Class name

```
class MyFancyClass {

public:
        int someNumber;
        public String someText;

        void someMethod() {

        }

};

void main(int argc, char **argv) {
        MyFancyClass c;

        MyFancyClass *cp = new MyFancyClass()

}
```

Access modifier

Class state

Class behaviour

'Magic' start point
for the program

Create an object of
type MyFancyClass and
call it cc

Create a pointer to a
MyFancyClass object and call it cp

Create an object of
type MyFancyClass and
return a reference to it

OOP Concepts

- OOP provides the programmer with a number of important concepts:
  - Modularity
  - Code Re-Use
  - Encapsulation
  - Inheritance
  - Polymorphism
- Let's look at these more closely...

Modularity and Code Re-Use

- You've long been taught to break down complex problems into more tractable sub-problems.
- Each class represents a sub-unit of code that (if written well) can be developed, tested and updated independently from the rest of the code.
- Indeed, two classes that achieve the same thing (but perhaps do it in different ways) can be swapped in the code
- Properly developed classes can be used in other programs without modification.

Encapsulation I

```
class Student {
   int age;
};

void main() {
   Student s = new Student();
   s.age = 21;

   Student s2 = new Student();
   s2.age=-1;

   Student s3 = new Student();
   s3.age=10055;
}
```

Encapsulation II

```
class Student {
   private int age;

   boolean setAge(int a) {
      if (a>=0 && a<130) {
         age=a;
         return true;
      }
      return false;
   }

   int getAge() {return age;}
}

void main() {
   Student s = new Student();
   s.setAge(21);
}
```

Encapsulation III

```
class Location {
   private float x;
   private float y;

   float getX() {return x;}
   float getY() {return y;}

   void setX(float nx) {x=nx;}
   void setY(float ny) {y=ny;}
}
```

```
class Location {

   private Vector2D v;

   float getX() {return v.getX();}
   float getY() {return v.getY();}

   void setX(float nx) {v.setX(nx);}
   void setY(float ny) {v.setY(ny);}
}
```

Access Modifiers

	Everyone	Subclass	Same package (Java)	Same Class
private				X
package (Java)			X	X
protected		X	X	X
public	X	X	X	X

Immutability

- Everything in ML was immutable (ignoring the reference stuff). Immutability has a number of advantages:
  - Easier to construct, test and use
  - Can be used in concurrent contexts
  - Allows lazy instantiation
- We can use our access modifiers to create immutable classes

Parameterised Classes

- ML's polymorphism allowed us to specify functions that could be applied to multiple types

```
> fun self(x)=x;
val self = fn : 'a -> 'a
```

- In Java, we can achieve something similar through *Generics*; C++ through *templates*
  - Classes are defined with placeholders (see later lectures)
  - We fill them in when we create objects using them

```
LinkedList<Integer> = new LinkedList<Integer>()

LinkedList<Double> = new LinkedList<Double>()
```

Creating Parameterised Types

- These just require a placeholder type

```
class Vector3D<T> {
    private T x;
    private T y;

    T getX() {return x;}
    T getY() {return y;}

    void setX(T nx) {x=nx;}
    void setY(T ny) {y=ny;}
}
```

Lecture 3:
Pointers, References and Memory

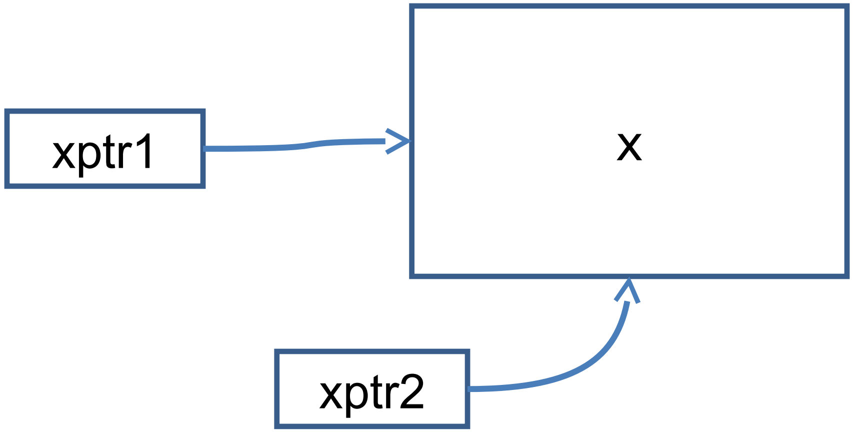
Memory and Pointers

- In reality the compiler stores a mapping from variable name to a specific memory address, along with the type so it knows how to interpret the memory (e.g. *"x is an int so it spans 4 bytes starting at memory address 43526"*).
- Lower level languages often let us work with memory addresses directly. Variables that store memory addresses are called **pointers** or sometimes **references**
- Manipulating memory directly allows us to write fast, efficient code, but also exposes us to bigger risks
  - Get it wrong and the program 'crashes' .

Pointers: Box and Arrow Model

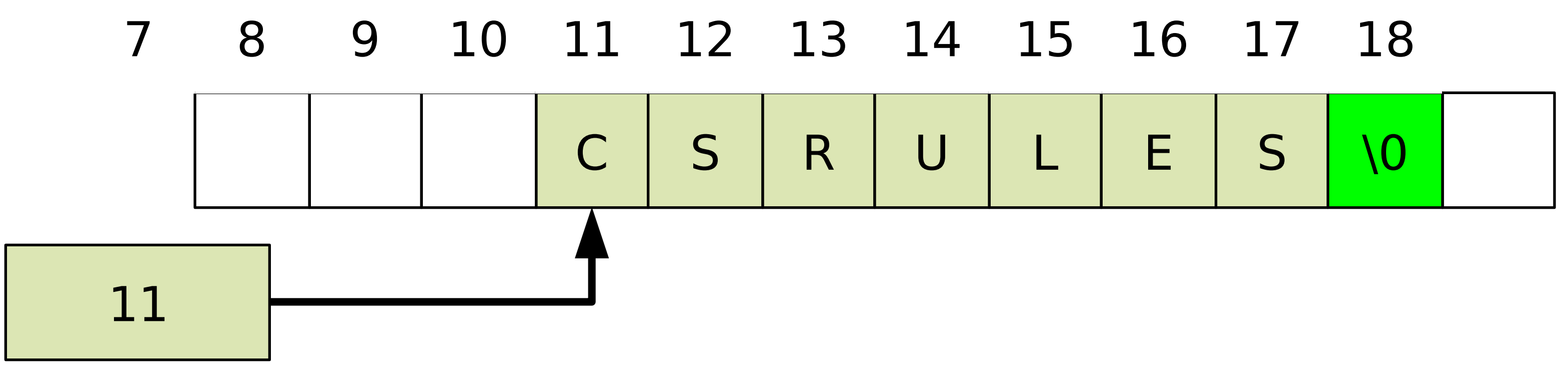
- A pointer is just the memory address of the first memory slot used by the variable
- The pointer type tells the compiler how many slots the whole object uses

```
int x = 72;
int *xptr1 = &x;
int *xptr2 = xptr1;
```



Example: Representing Strings I

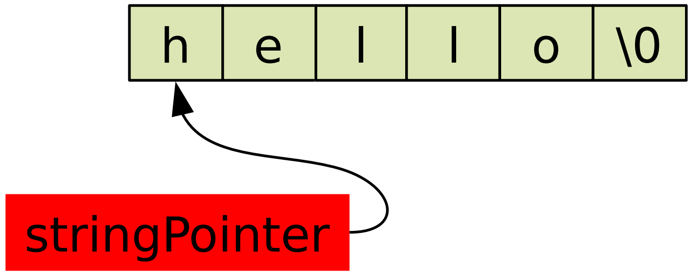
- A single character is fine, but a text string is of variable length – how can we cope with that?
- We simply store the start of the string in memory and require it to finish with a special character (the NULL or terminating character, aka '\0')
- So now we need to be able to store memory addresses → use pointers



- We think of there being an array of characters (single letters) in memory, with the string pointer pointing to the first element of that array

Example: Representing Strings II

```
char letterArray[] = {'h','e','l','l','o','\0'};

char *stringPointer = &(letterArray[0]);

printf("%s\n",stringPointer);

letterArray[3]='\0';

printf("%s\n",stringPointer);
```



References

- A reference is an **alias** for another thing (object/array/etc)
- When you use it, you are 'redirected' somehow to the underlying thing
- Properties:
  - Either assigned or unassigned
  - If assigned, it is valid
  - You can easily check if assigned

Implementing References

- A sane reference implementation in an imperative language is going to use pointers
- So each reference is the same as a pointer <u>except</u> that the compiler restricts operations that would violate the properties of references
- For this course, thinking of a reference as a restricted pointer is fine

Distinguishing References and Pointers

	Pointers	References
Can be unassigned (null)	Yes	Yes
Can be assigned to established object	Yes	Yes
Can be assigned to an arbitrary chunk of memory	Yes	**No**
Can be tested for validity	**No**	Yes
Can perform arithmetic	Yes	**No**

Languages and References

- Pointers are useful but dangerous
- C, C++: pointers *and* references
- Java: references *only*
- ML: references *only*

References in Java

- Declaring unassigned

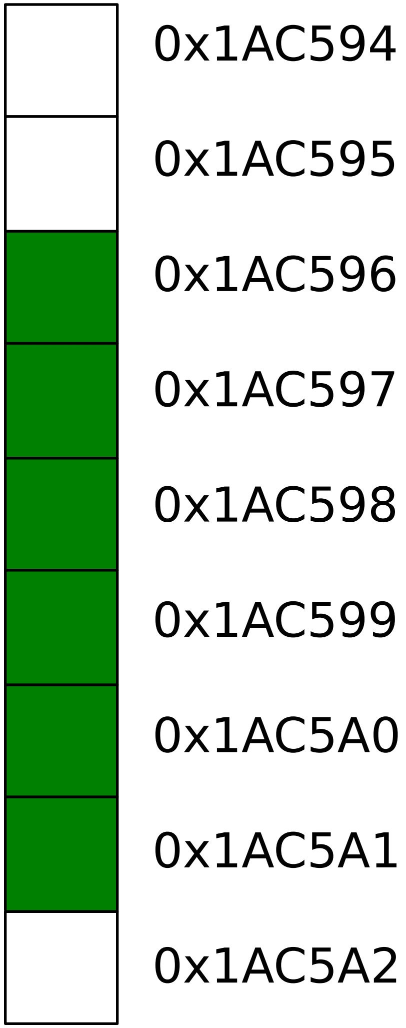
```
SomeClass ref = null;  // explicit

SomeClass ref2;  // implicit
```

- Defining/assigning

```
// Assign
SomeClass ref = new ClassRef();

// Reassign to alias something else
ref = new ClassRef();

// Reference the same thing as another reference
SomeClass ref2 = ref;
```

Arrays

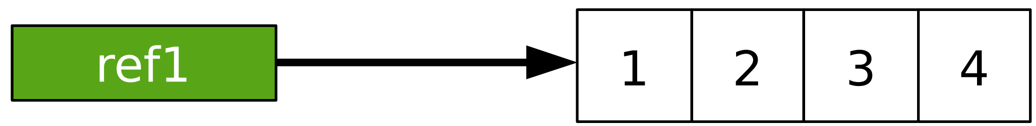
```
byte[] arraydemo1 = new byte[6];
byte   arraydemo2[] = new byte[6];
```



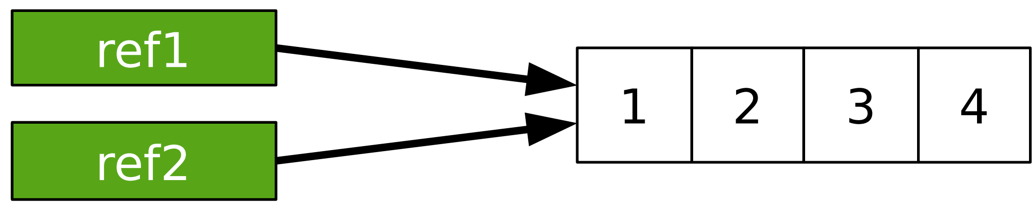
References Example (Java)

```
int[] ref1 = null;
```

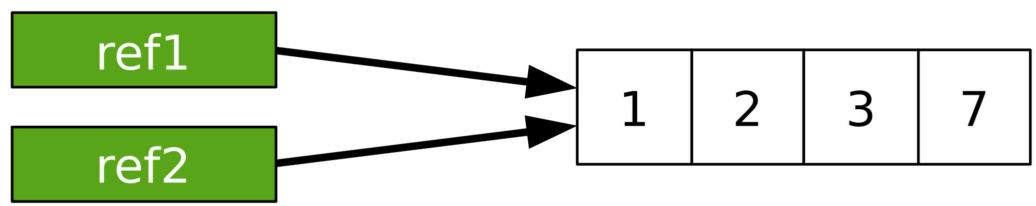

```
ref1 = new int[]{1,2,3,4};
```



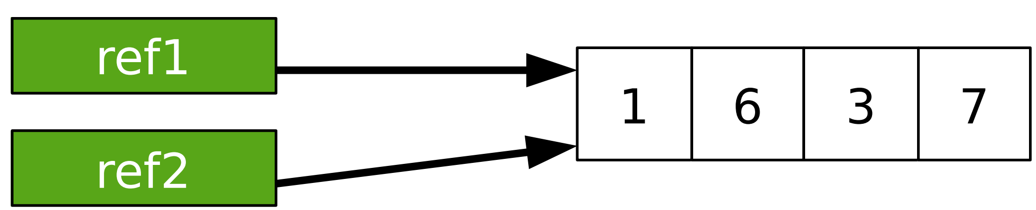

```
int[] ref2 = ref1;
```




```
ref1[3]=7;
```




```
ref2[1]=6;
```



Primitive types

You have "direct" access

Reference Types

Arrays
Objects

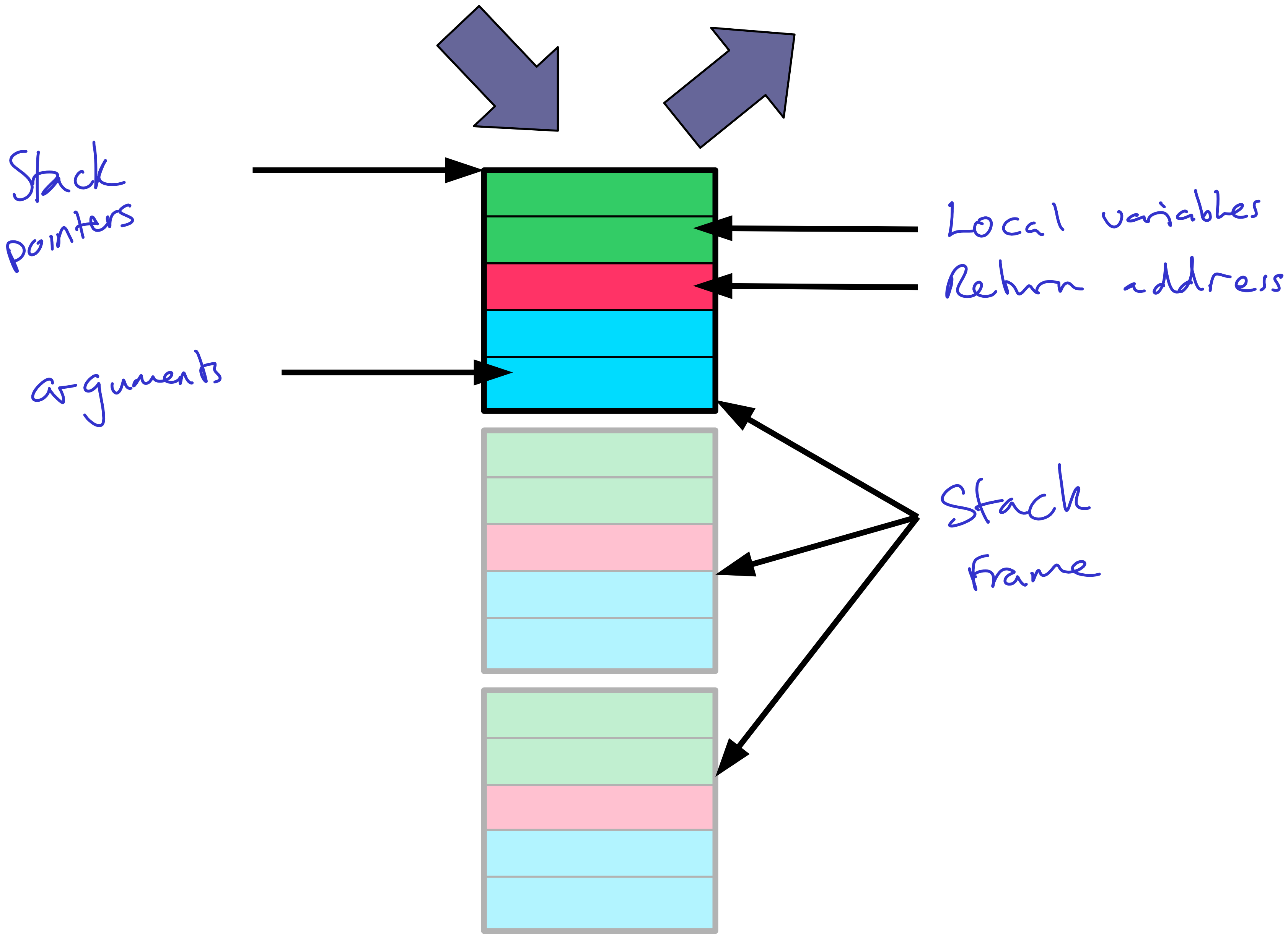
} You only get references

Keeping Track of Function Calls

- We need a way of keeping track of which functions are currently running

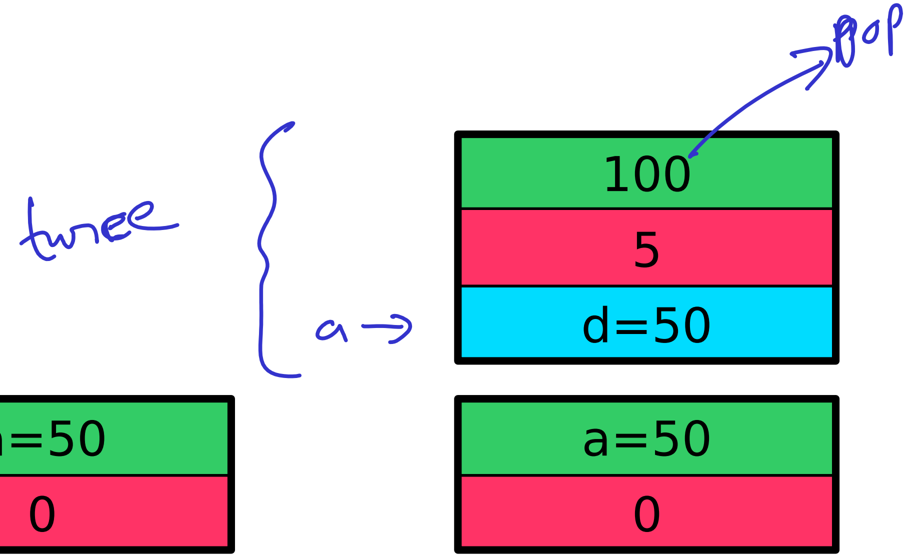
```
public void a() {
   //...
}

public void b() {
   a();
}
```

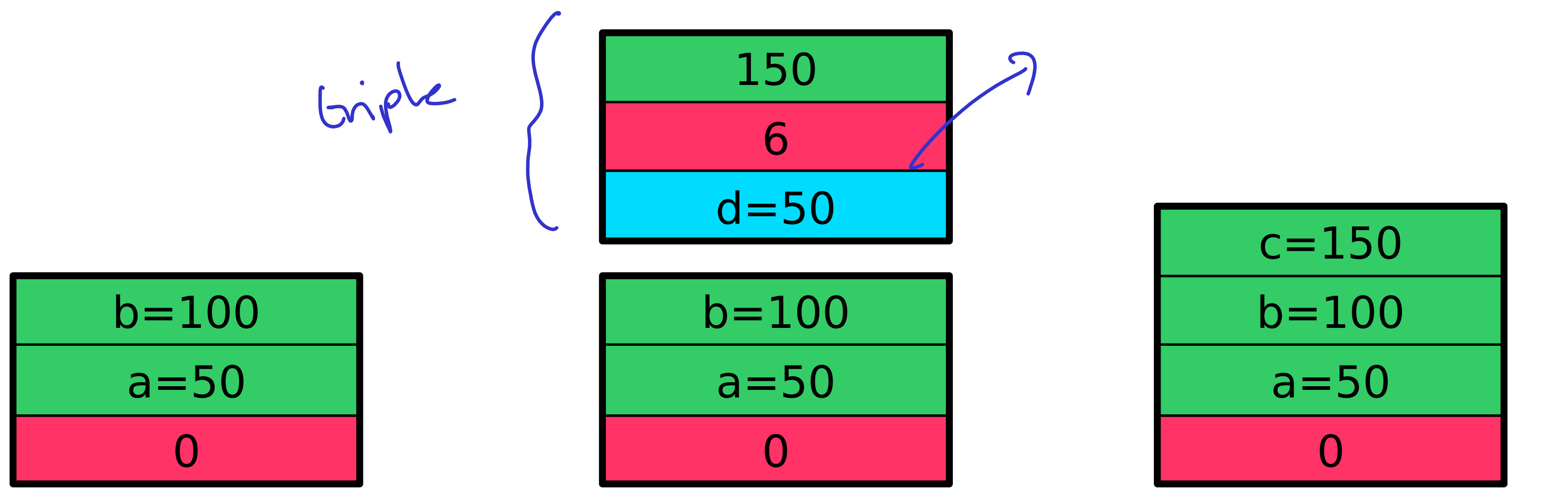
The Call Stack



The Call Stack: Example

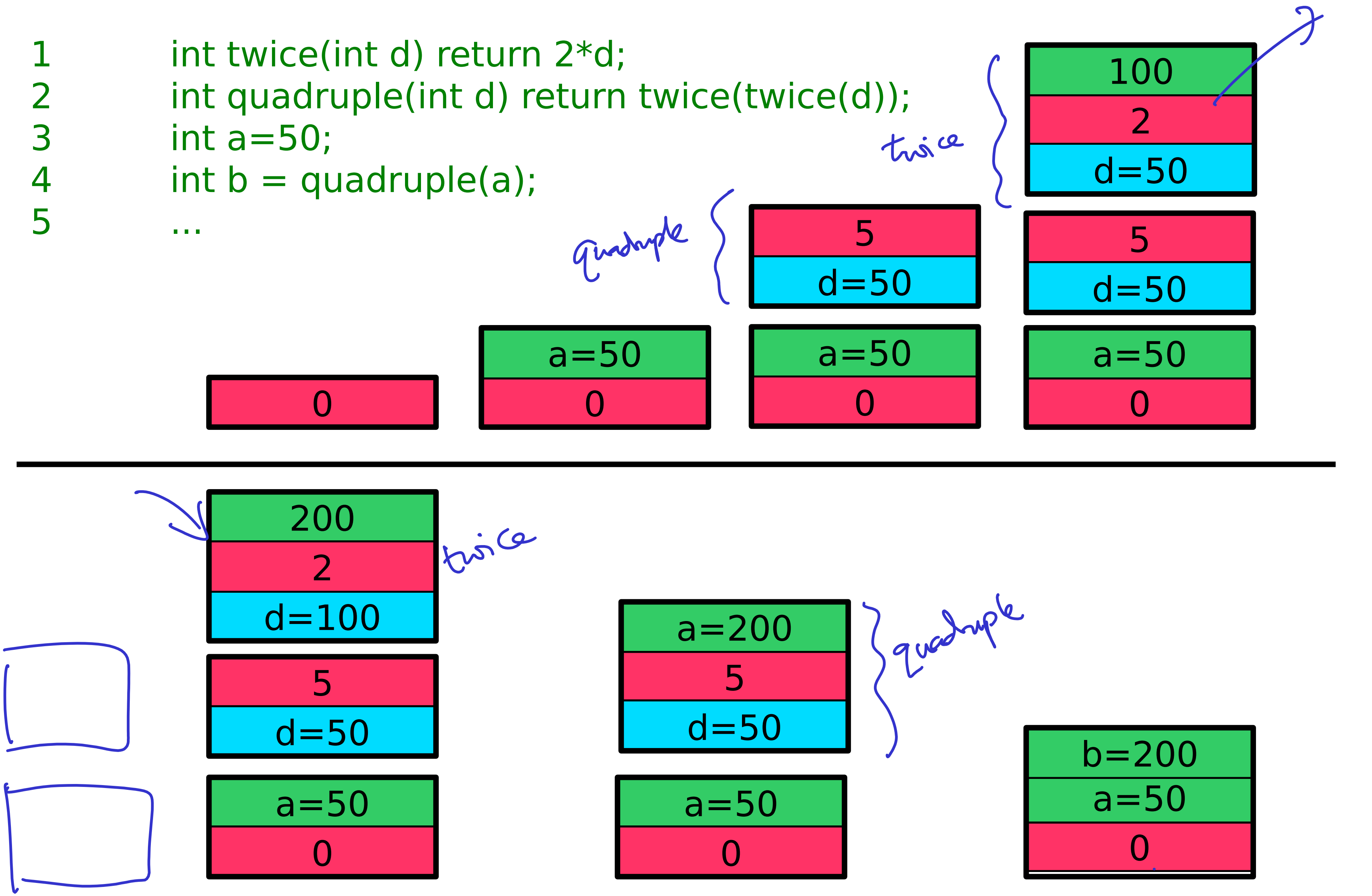
```
int twice(int d) return 2*d;
int triple(int d) return 3*d;
int a = 50;
int b = twice(a);
int c = triple(a);
...
```





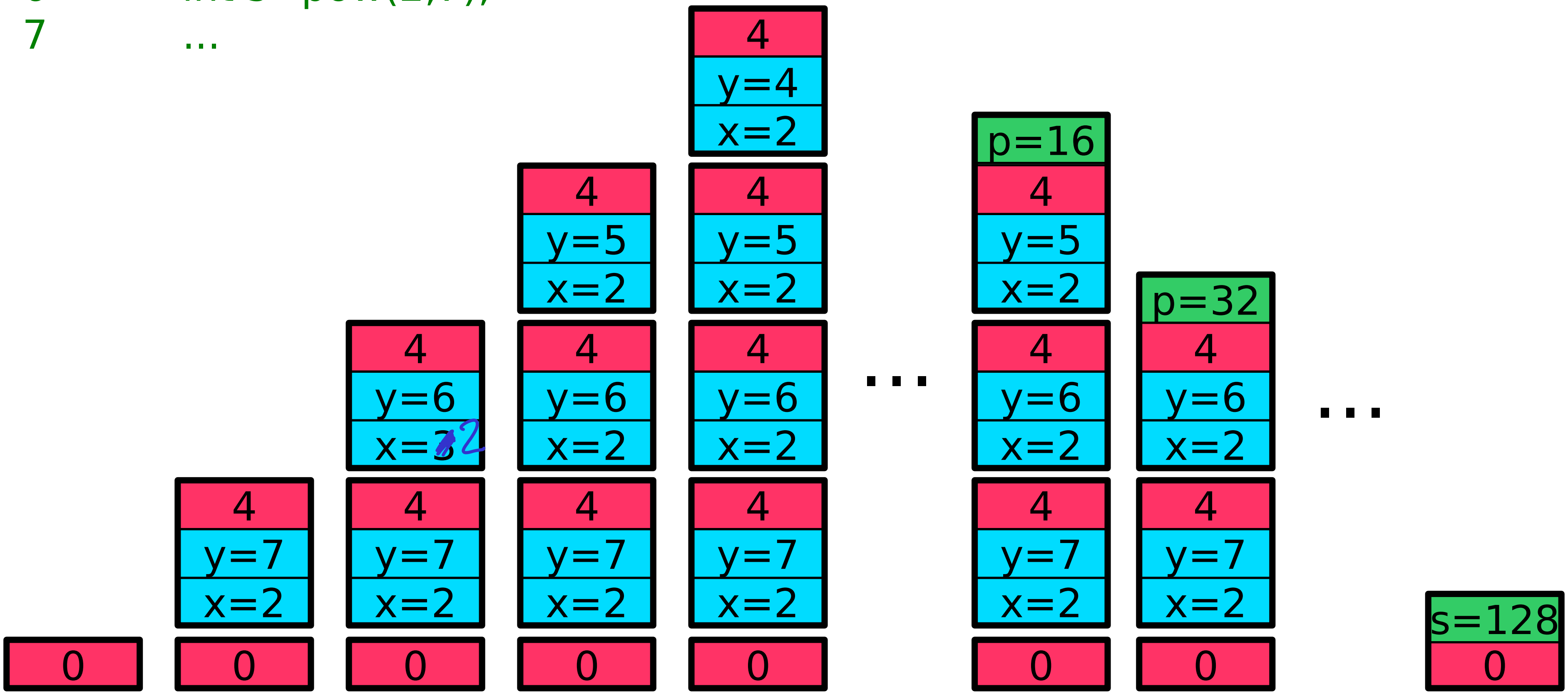
Nested Functions

```
int twice(int d) return 2*d;
int quadruple(int d) return twice(twice(d));
int a=50;
int b = quadruple(a);
...
```



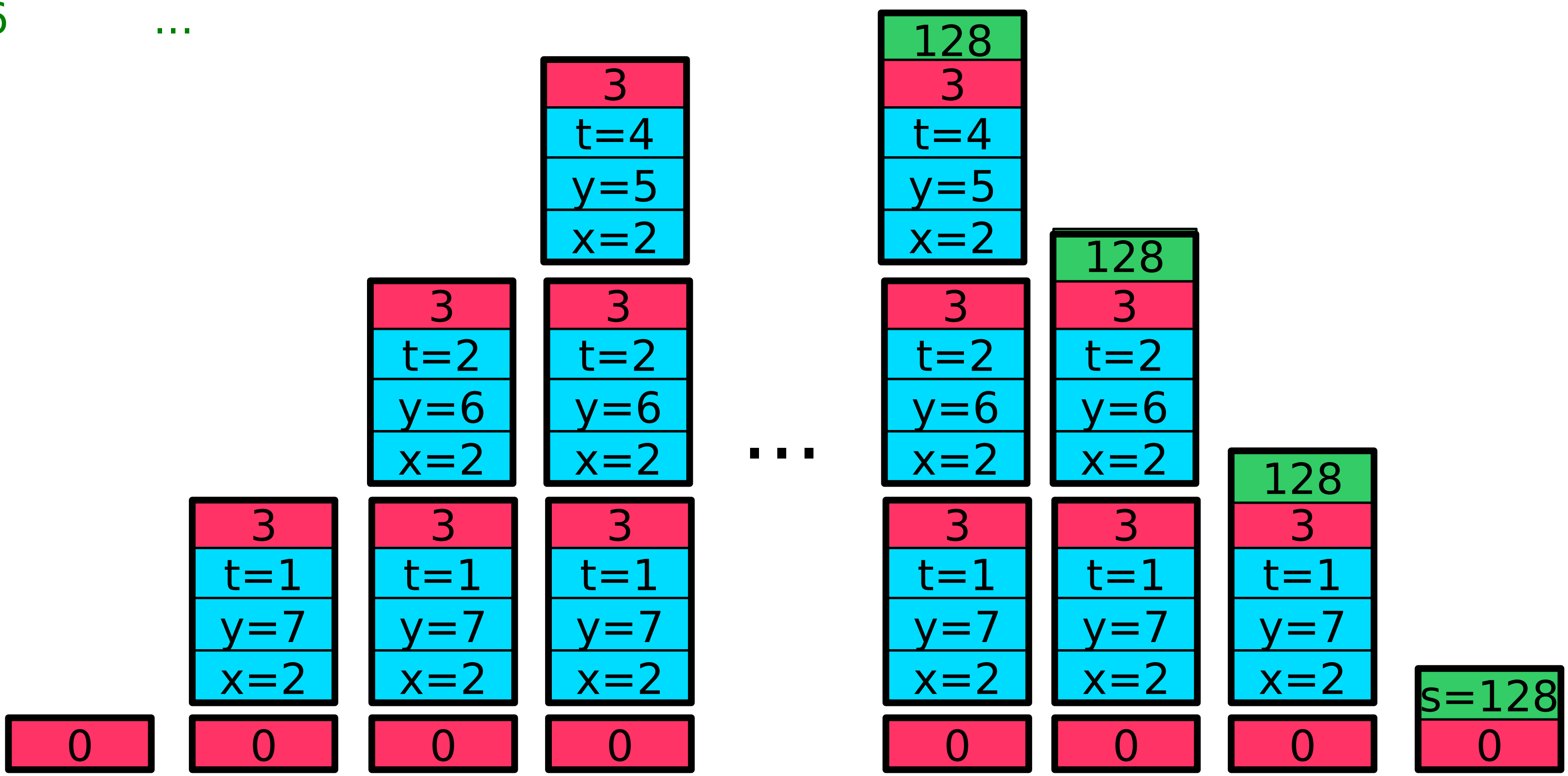
Recursive Functions

```
int pow (int x, int y) {
        if (y==0) return 1;
        int p = pow(x,y-1);
        return x*p;
}
int s=pow(2,7);
...
```



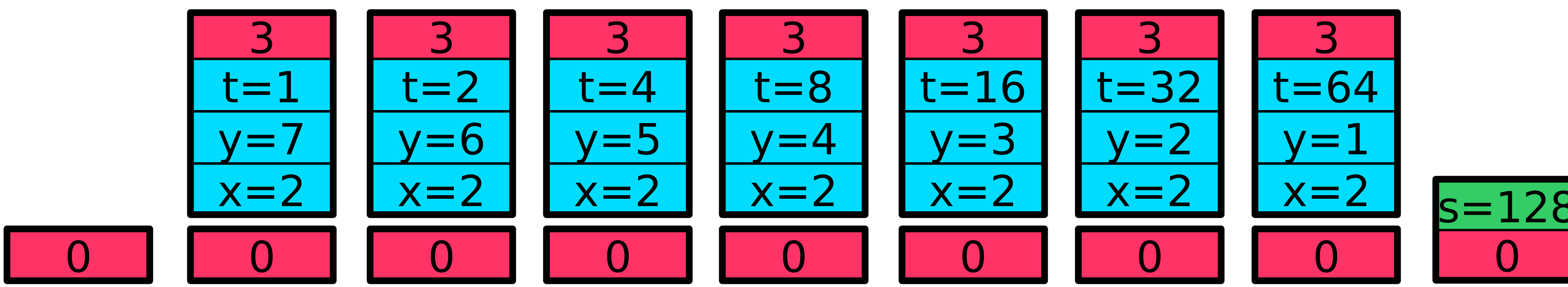
Tail-Recursive Functions I

```
int pow (int x, int y, int t) {
        if (y==0) return t;
        return pow(x,y-1, t*x);
}
int s = pow(2,7,1);
...
```



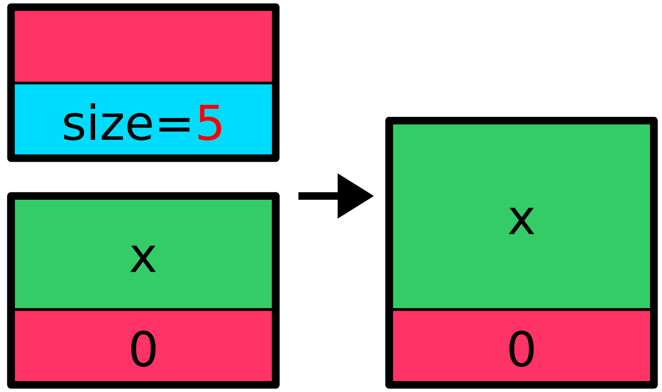
Tail-Recursive Functions II

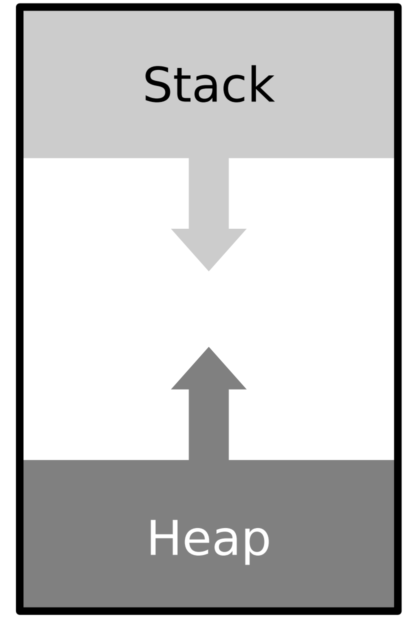
```
int pow (int x, int y, int t) {
        if (y==0) return t;
        return pow(x,y-1, t*x);
}
int s = pow(2,7,1);
...
```

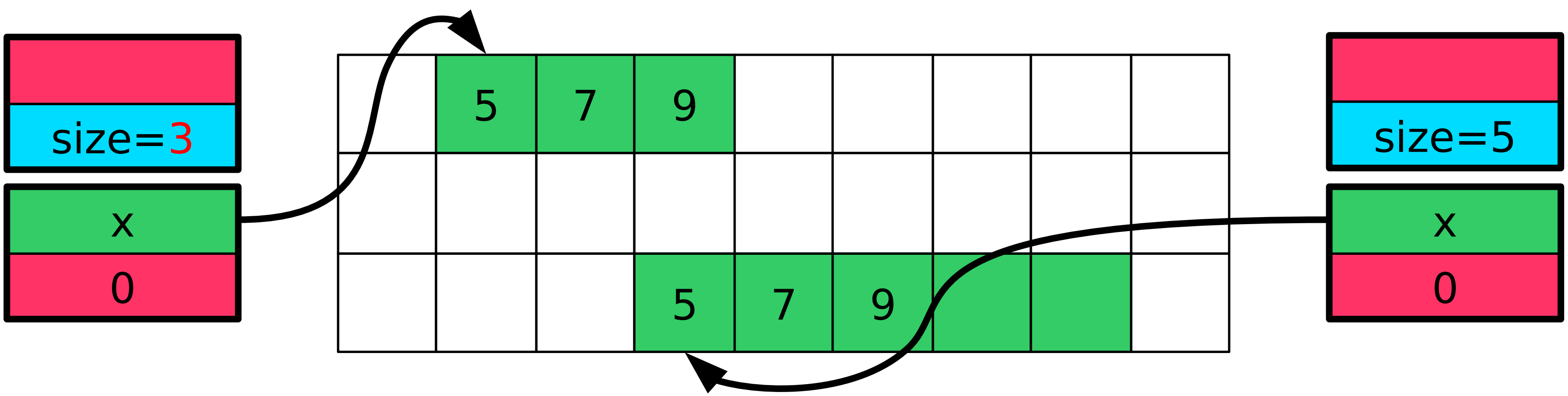


The Heap

```
int[] x = new int[3];
public void resize(int size) {
    int tmp=x;
    x=new int[size];
    for (int=0; i<3; i++)
        x[i]=tmp[i];
}
resize(5);
```

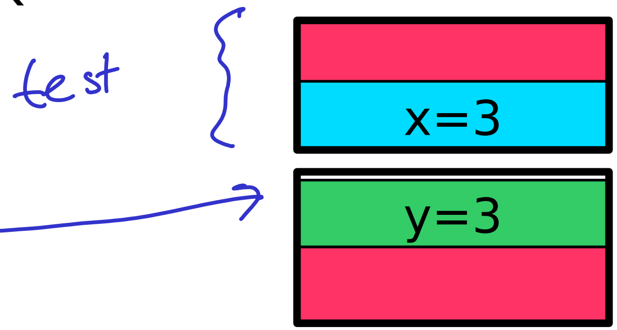






Argument Passing

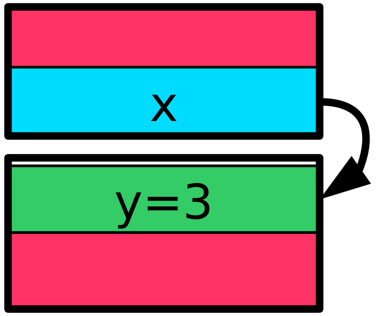
- **Pass-by-value**. Copy the object into a new value in the stack




```
void test(int x) {...}
int y=3;
test(y);
```

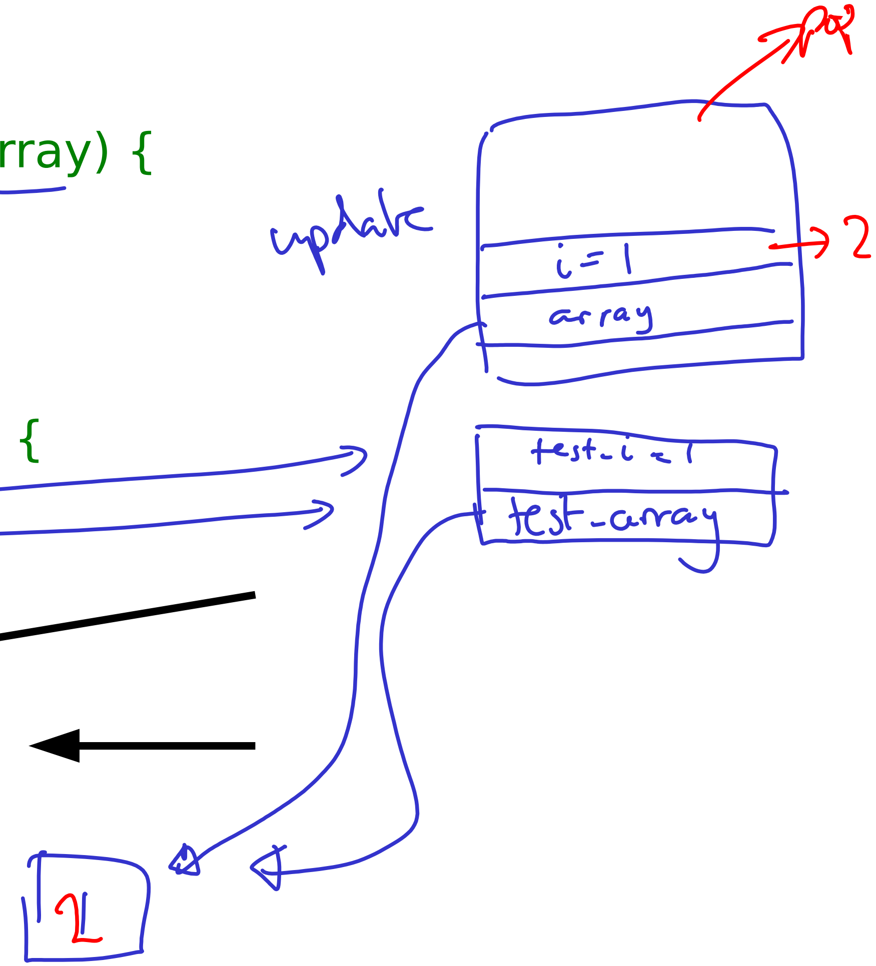
- **Pass-by-reference**. Create a reference to the object and pass that.

```
void test(int &x) {...}
int y=3;
test(y);
```



Passing Procedure Arguments In Java

```
class Reference {

  public static void update(int i, int[] array) {
    i++;
    array[0]++;
  }

  public static void main(String[] args) {
    int test_i = 1;
    int[] test_array = {1};
    update(test_i, test_array);
    System.out.println(test_i);
    System.out.println(test_array[0]);
  }

}
```



Passing Procedure Arguments In C++

```
void update(int i, int &iref){
  i++;
  iref++;
}

int main(int argc, char** argv) {
  int a=1;
  int b=1;
  update(a,b);
  printf("%d %d\n",a,b);
}
```

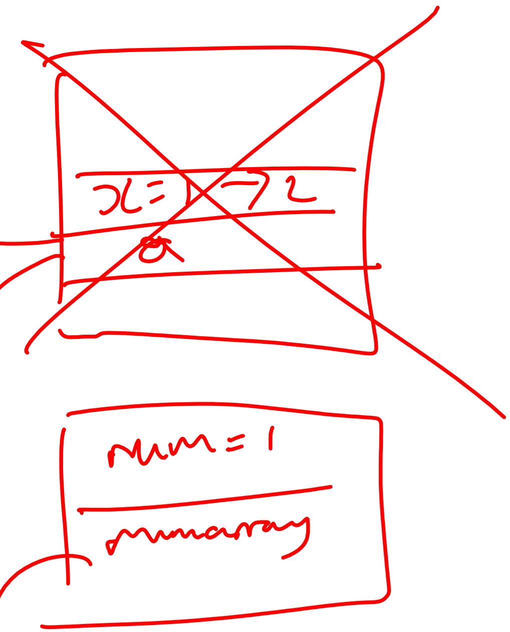
Check...

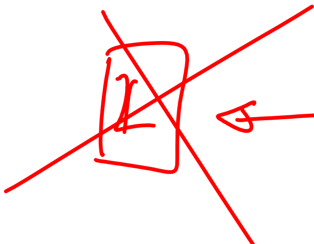
```
public static void myfunction  (int x, int[] a) {
        x=1;
        x=x+1;

        a[0]=a[0]+1;
}

public static void main(String[] arguments) {
        int num=1;
        int numarray[] = {1};

        myfunction  (num, numarray);
        System.out.println(num+" "+numarray[0]);
}
```

A. “1 1”
B. “1 2”
C. “2 1”
D. “2 2”

Check...

```
public static void myfunction2(int x, int[] a) {
        x=1;
        x=x+1;
        a = new int[]{1};
        a[0]=a[0]+1;
}

public static void main(String[] arguments) {
        int num=1;
        int numarray[] = {1};

        myfunction2(num, numarray);
        System.out.println(num+" "+numarray[0]);
}
```





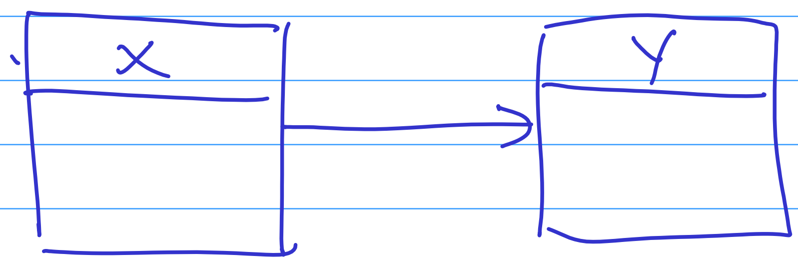
A. “1 1”
B. “1 2”
C. “2 1”
D. “2 2”

Lecture 4:
Inheritance

Recap

OOP is about structuring code

- Modularity
- Reuse
- Encapsulation
  - change underlying representation
  - add sanity checks
  - consistent interface



X "has-a" Y

Inheritance I

```
class Student {
   public int age;
   public String name;
   public int grade;
}

class Lecturer {
   public int age;
   public String name;
   public int salary;
}
```

- There is a lot of duplication here
- Conceptually there is a hierarchy that we're not really representing
- Both Lecturers and Students are people (no, really).
- We can view each as a kind of specialisation of a general person
  - They have all the properties of a person
  - But they also have some extra stuff specific to them

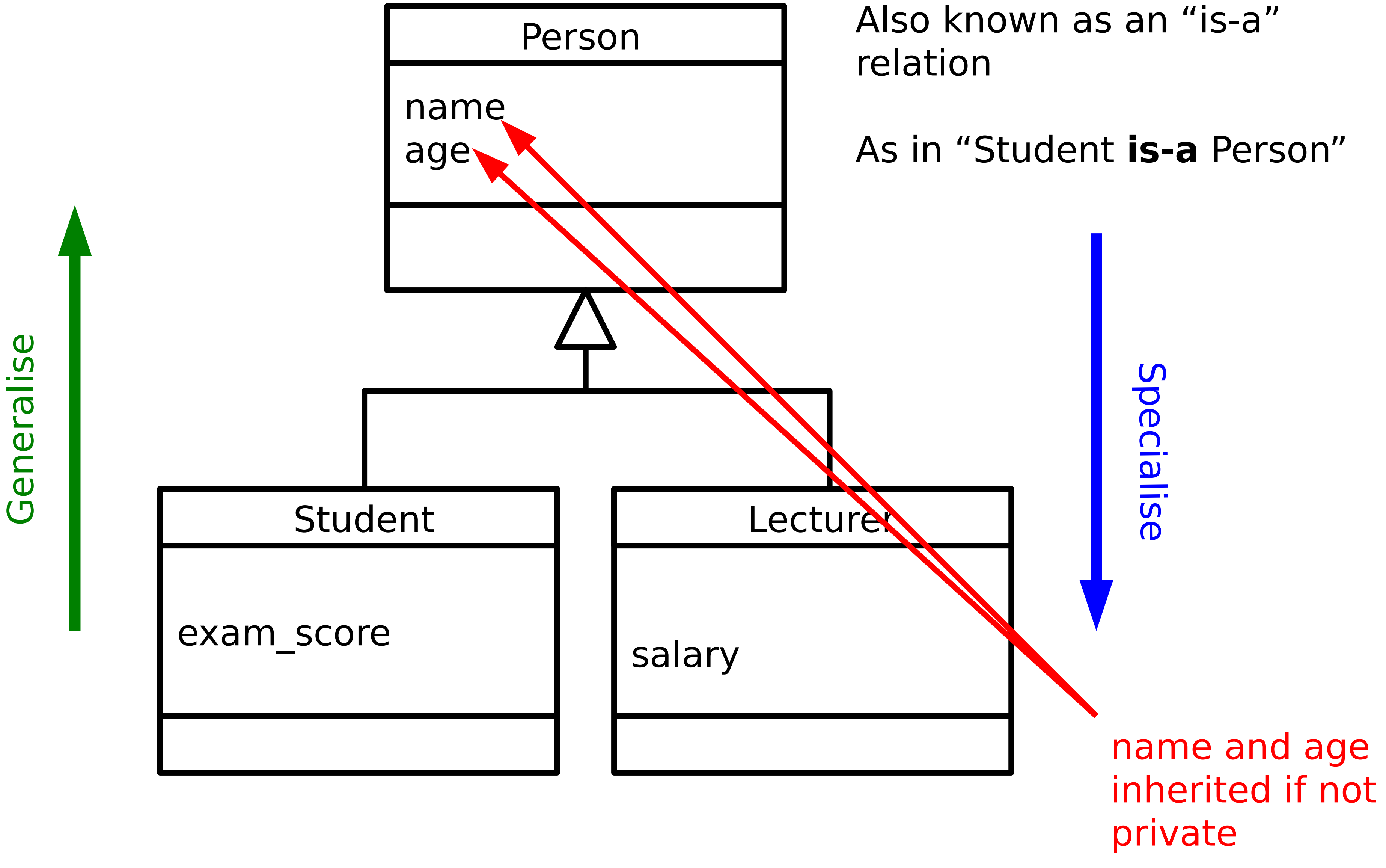
(I should not have used public variables here, but I did it to keep things simple)

Inheritance II

```
class Person {
   public int age;
   public String name;
}

class Student extends Person {
   public int grade;
}

class Lecturer extends Person {
   public int salary;
}
```

- We create a *base class* (Person) and add a new notion: classes can *inherit* properties from it
  - Both state and functionality
- We say:
  - Person is the *superclass* of Lecturer and Student
  - Lecturer and Student *subclass* Person

Representing Inheritance Graphically



Also known as an “is-a” relation

As in “Student **is-a** Person”

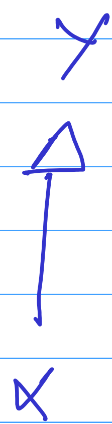
Arrows to know

$X \longrightarrow Y$

Association

"has-a"

X has a Y



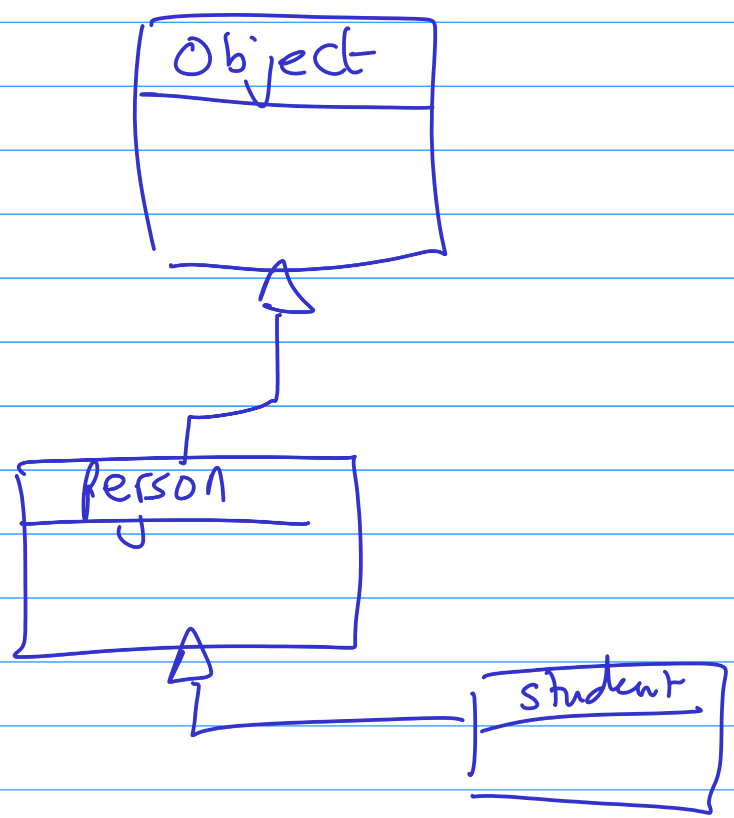
Inheritance

"is-a"

X is a Y

Java Oddity : Object

Everything inherits from Object



Casting

- Many languages support *type casting* between numeric types

Cast

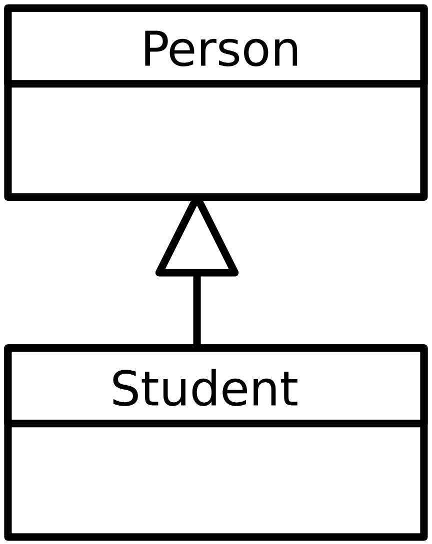
```
int i = 7;
float f = (float) i;   // f==7.0
double d = 3.2;
int i2 = (int) d;     // i2==3
```

increased precision
upcast
widening

downcast
narrowing

- With inheritance it is reasonable to type cast an object to any of the types above it in the inheritance tree...

Widening

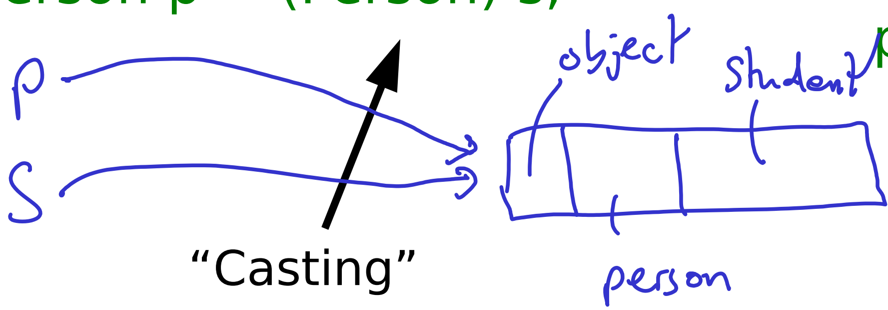


- Student is-a Person
- Hence we can use a Student object anywhere we want a Person object
- Can perform *widening* conversions (up the tree)

```
Student s = new Student()
```

ref

```
Person p = (Person) s;
```

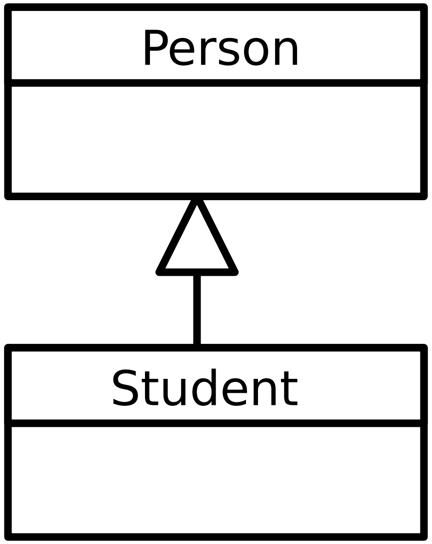


“Casting”

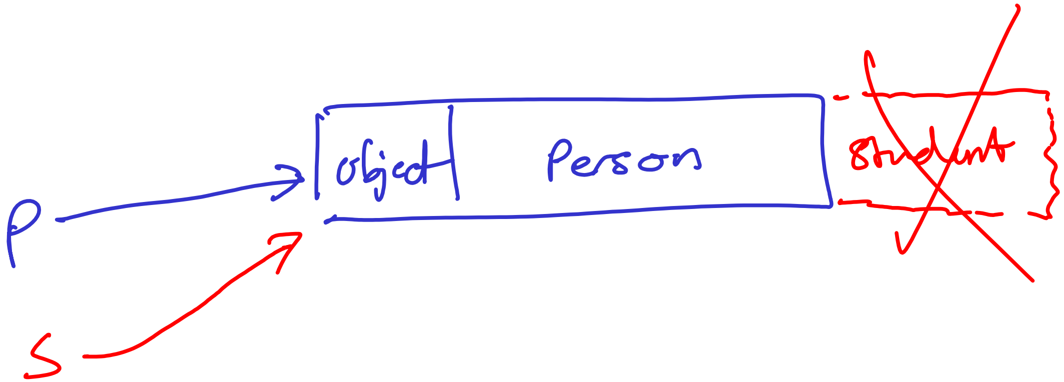
```
public void print(Person p) {...}

Student s = new Student();
print(s);
```

Implicit cast

Narrowing



- Narrowing conversions move down the tree (more specific)
- Need to take care...




```
Person p = new Person();

Student s = (Student) p;
```

FAILS. Not enough info
In the real object to represent
a Student

```
Student s = new Student();
Person p = (Person) s;
Students s2 = (Student) p;
```

OK because underlying object
really is a Student

Why is this important?

```
public int getInitials (Student s) {
    //
}

public int getInitials (Lecturer l) {
    //
}
```

Fields and Inheritance

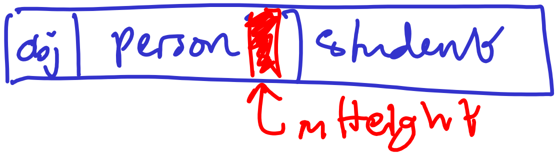
```
class Person {
   public String mName;
   protected int mAge;
   private double mHeight;
}

class Student extends Person {

  public void do_something() {
    mName="Bob";
    mAge=70;
    mHeight=1.70;
  }

}
```

Fails to compile

Student inherits this as a public variable and so can access it

Student inherits this as a protected variable and so can access it

Student inherits this but as a **private** variable and so cannot access it directly



public : Anyone can access

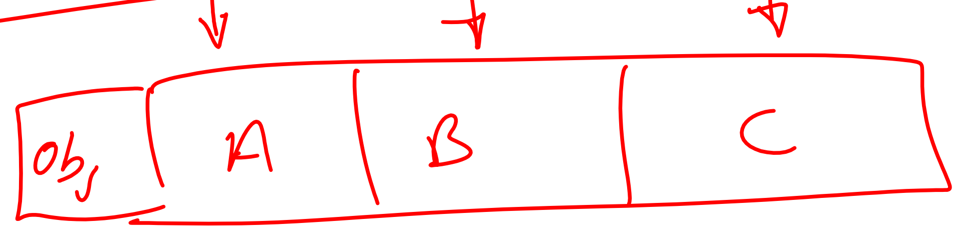
protected: subclasses can access. (+ package in Java)

private : Only this class

(package): Anything in same package

Fields and Inheritance: Shadowing

```
class A {   public int x; }

class B extends A {
   public int x;
}

class C extends B {
  public int x;

  public void action() {
     // Ways to set the x in C
     x = 10;
     this.x = 10;

     // Ways to set the x in B
     super.x = 10;
     ((B)this).x = 10;

     // Ways to set the x in A
     ((A)this.x = 10;
  }
}
```



Methods and Inheritance: Overriding

- We might want to require that every Person can dance. But the way a Lecturer dances is not likely to be the same as the way a Student dances...

```
class Person {
   public void dance() {
      jiggle_a_bit();
   }
}
```

Person defines a 'default' implementation of dance()

```
class Student extends Person {
   public void dance() {
      body_pop();
   }
}
```

Student overrides the default

```
class Lecturer extends Person {
}
```

Lecturer just inherits the default implementation and jiggles

Abstract Methods

- Sometimes we want to force a class to implement a method but there isn't a convenient default behaviour
- An **abstract** method is used in a base class to do this
- It has no implementation whatsoever

```
class abstract Person {
   public abstract void dance();
}

class Student extends Person {
   public void dance() {
      body_pop();
   }
}

class Lecturer extends Person {
   public void dance() {
      jiggle_a_bit();
   }
}
```

Person p = new Person()

Abstract Classes

- Note that I had to declare the class abstract too. This is because it has a method without an implementation so we can't directly instantiate a Person.

```java
public abstract class Person {
   public abstract void dance();
}
```

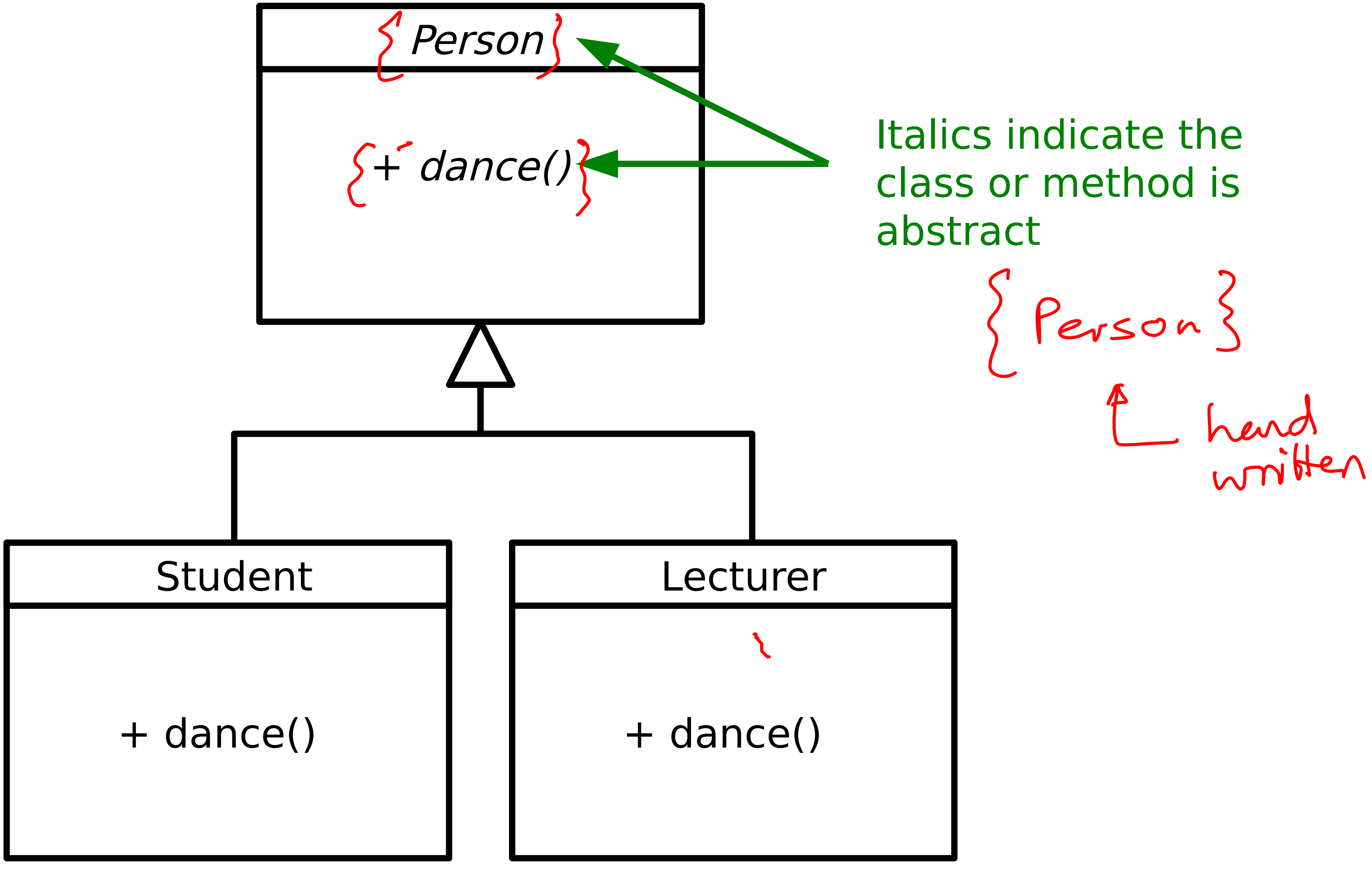
Java

```cpp
class Person {
   public:
      virtual void dance()=0;
}
```

C++

- All state and non-abstract methods are inherited as normal by children of our abstract class
- Interestingly, Java allows a class to be declared abstract even if it contains no abstract methods!

Representing Abstract Classes



Italics indicate the class or method is abstract

Lecture 5:
Polymorphism and Multiple Inheritance

Polymorphic Methods

```
Student s = new Student();
Person p = (Person)s;
p.dance();
```

- Assuming Person has a default dance() method, what should happen here??

- General problem: when we refer to an object via a parent type and both types implement a particular method: which method should it run?

Polymorphic Concepts I

- **Static** polymorphism
  - Decide at compile-time
  - Since we don't know what the true type of the object will be, we just run the parent method
  - Type errors give compile errors

```
Student s = new Student();
Person p = (Person)s;
p.dance();
```

- Compiler says “p is of type Person”
- So p.dance() should do the default dance() action in Person

Why can't the compiler figure out the true type?

```
Person p = null
if (something()) p = new Student();
else p = new TaxPayer();
p.dance();
```

Conditional branch

Compiler has no idea what is in memory

Static Polymorphism You've Seen Already

ML

```
fun cons a xs = a::xs;
```
← any type

```
cons 1 [2,3,4];
```
← compiler set it up for integers

Java

Template param

```
public class LinkedList <T> {
    ....
}

new LinkedList<Person>
```

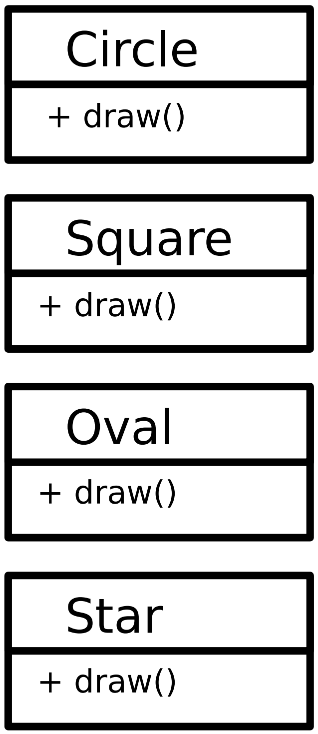
Polymorphic Concepts II

- **Dynamic** polymorphism
  - Run the method in the child
  - Must be done at run-time since that's when we know the child's type
  - Type errors cause run-time faults (crashes!)

```
Student s = new Student();
Person p = (Person)s;
p.dance();
```

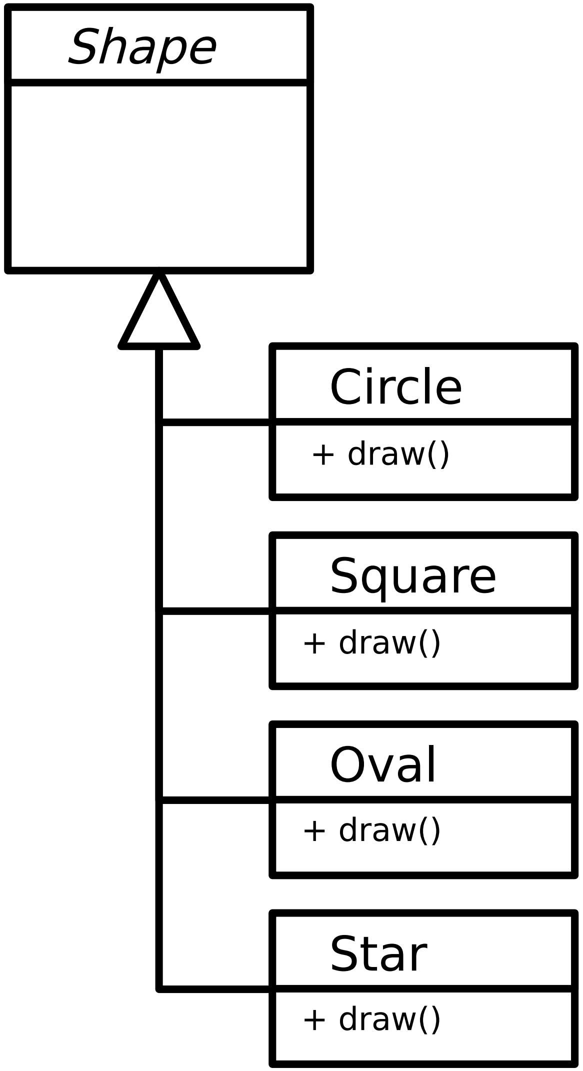
- Compiler looks in memory and finds that the object is really a Student
- So p.dance() runs the dance() action in Student

The Canonical Example I



- A drawing program that can draw circles, squares, ovals and stars
- It would presumably keep a list of all the drawing objects
- **Option 1**
  - Keep a list of Circle objects, a list of Square objects,...
  - Iterate over each list drawing each object in turn
  - What has to change if we want to add a new shape?

The Canonical Example II

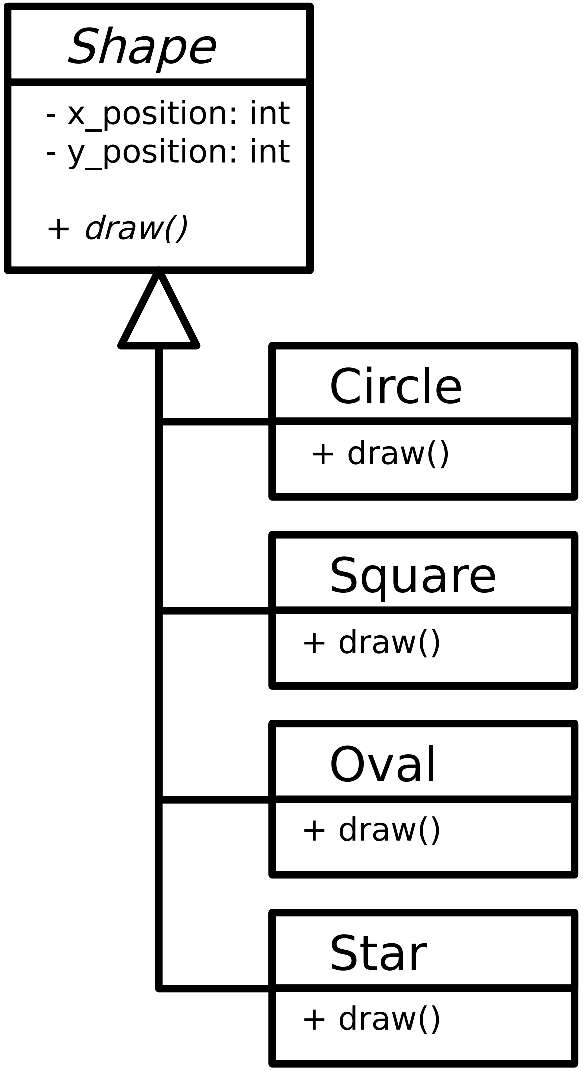


- **Option 2**
  - Keep a single list of Shape references
  - Figure out what each object really is, narrow the reference and then draw()

```
for every Shape s in myShapeList
   if (s is really a Circle)
      Circle c = (Circle)s;
      c.draw();
   else if (s is really a Square)
      Square sq = (Square)s;
      sq.draw();
   else if...
```

  - What if we want to add a new shape?

The Canonical Example III



- **Option 3 (Polymorphic)**
  - Keep a single list of Shape references
  - Let the compiler figure out what to do with each Shape reference

```
For every Shape s in myShapeList
   s.draw();
```

  - What if we want to add a new shape?

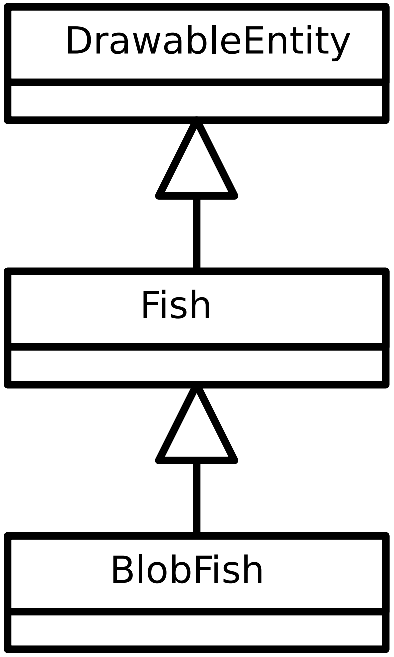
Implementations

- Java
  - All methods are dynamic polymorphic.
- Python
  - All methods are dynamic polymorphic.
- C++
  - Only functions marked *virtual* are dynamic polymorphic

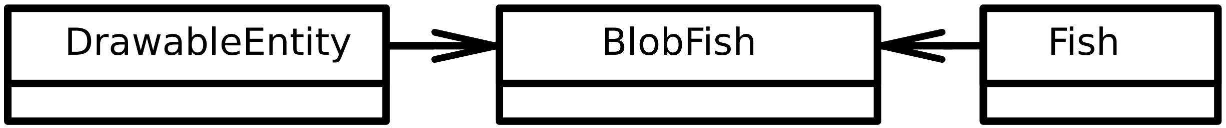
- Polymorphism in OOP is an extremely important concept that you need to make <u>sure</u> you understand...

Harder Problems

- Given a class Fish and a class DrawableEntity, how do we make a BlobFish class that is a drawable fish?

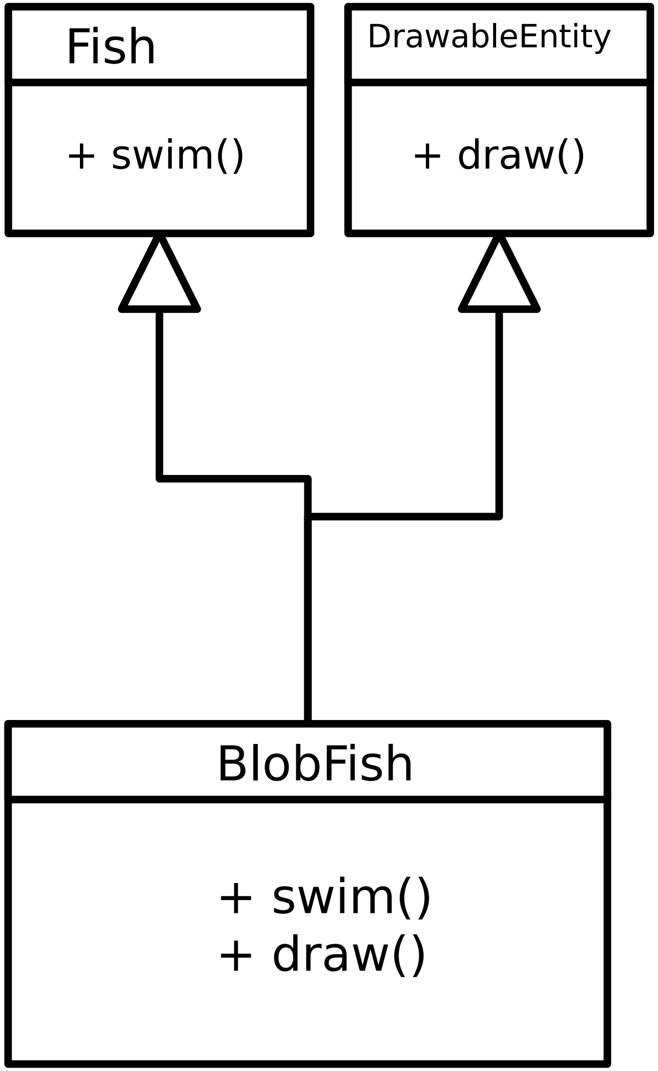


X Dependency between two independent concepts



X Conceptually wrong

Multiple Inheritance



- If we multiple inherit, we capture the concept we want
- BlobFish inherits from both and is-a Fish and is-a DrawableEntity
- C++:

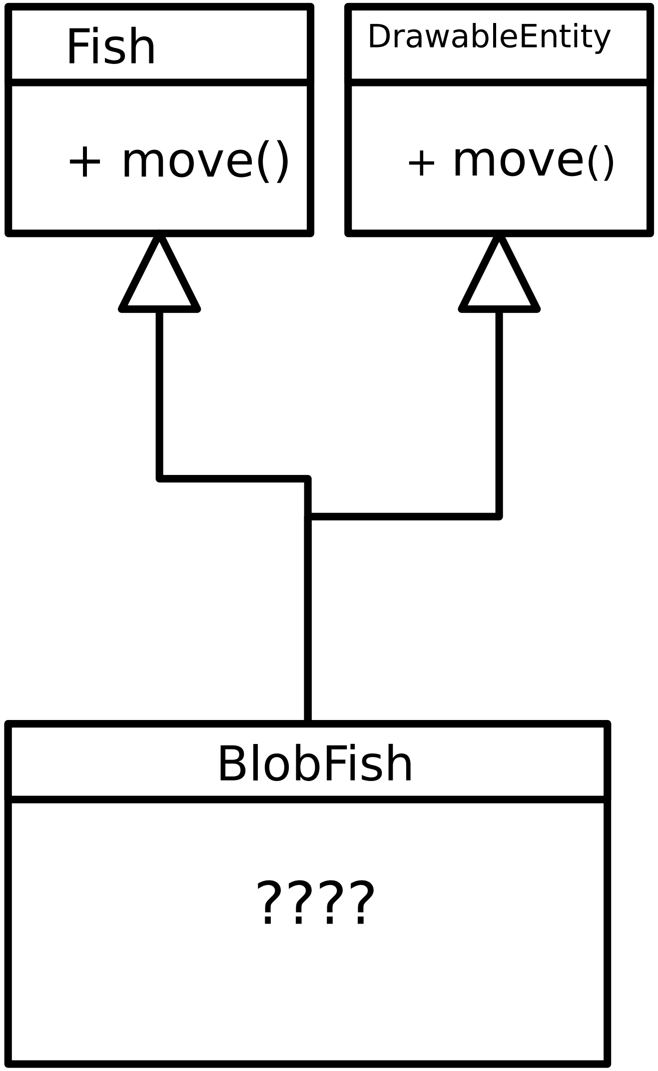
```
class Fish {…}
class DrawableEntity {…}

class BlobFish : public Fish,
                 public DrawableEntity {…}
```

- But…

BlobFish *b = new Blobfish();

b->swim();

Multiple Inheritance Problems

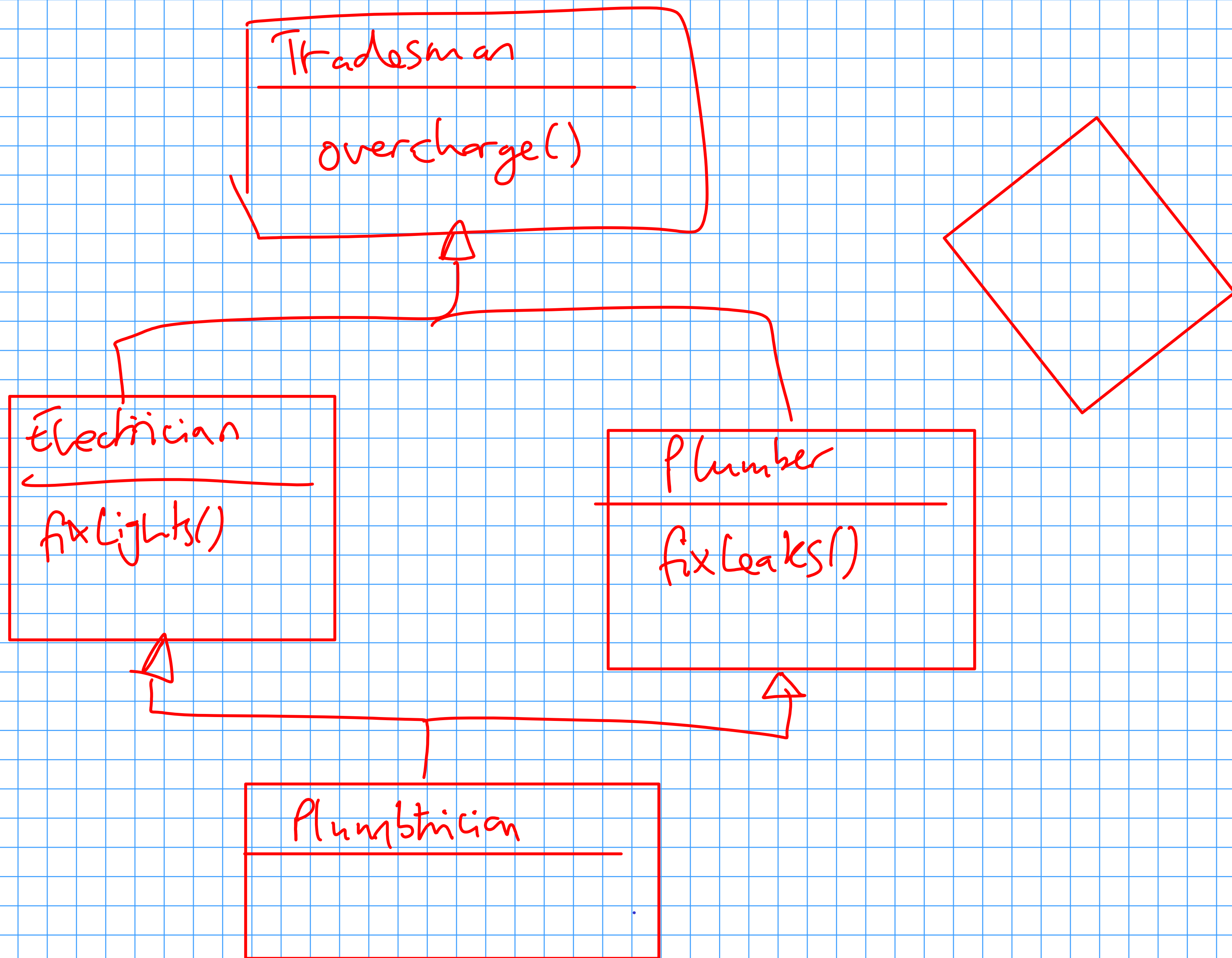


- What happens here? Which of the move() methods is inherited?
- Have to add some grammar to make it explicit
- C++:

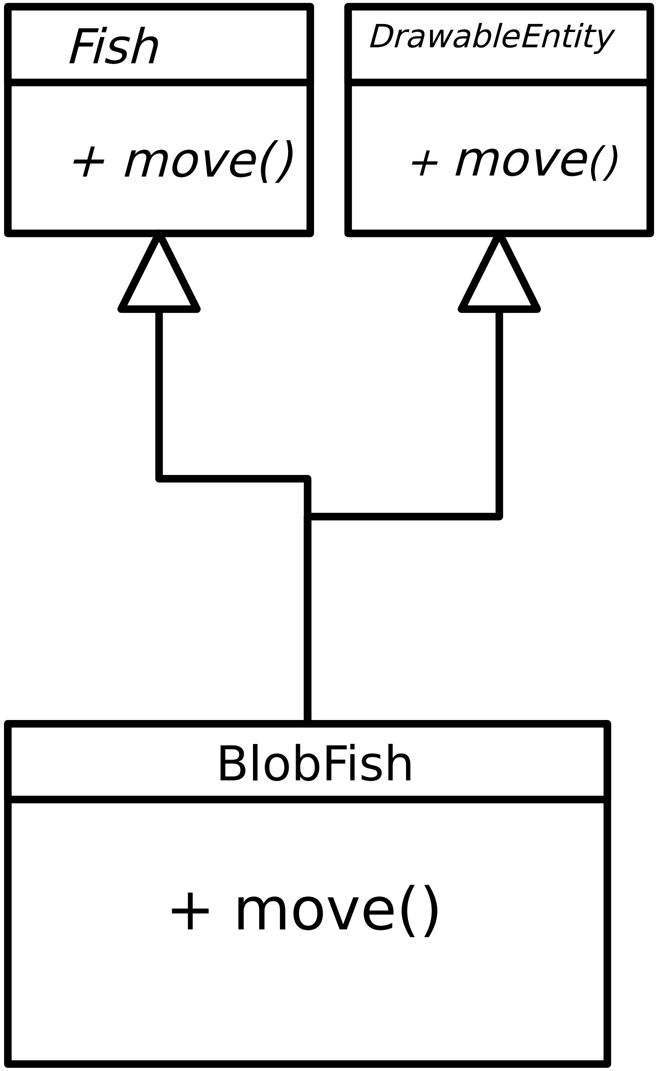
```
BlobFish *bf = new BlobFish();
bf->Fish::move();
bf->DrawableEntity::move();
```

- Yuk.

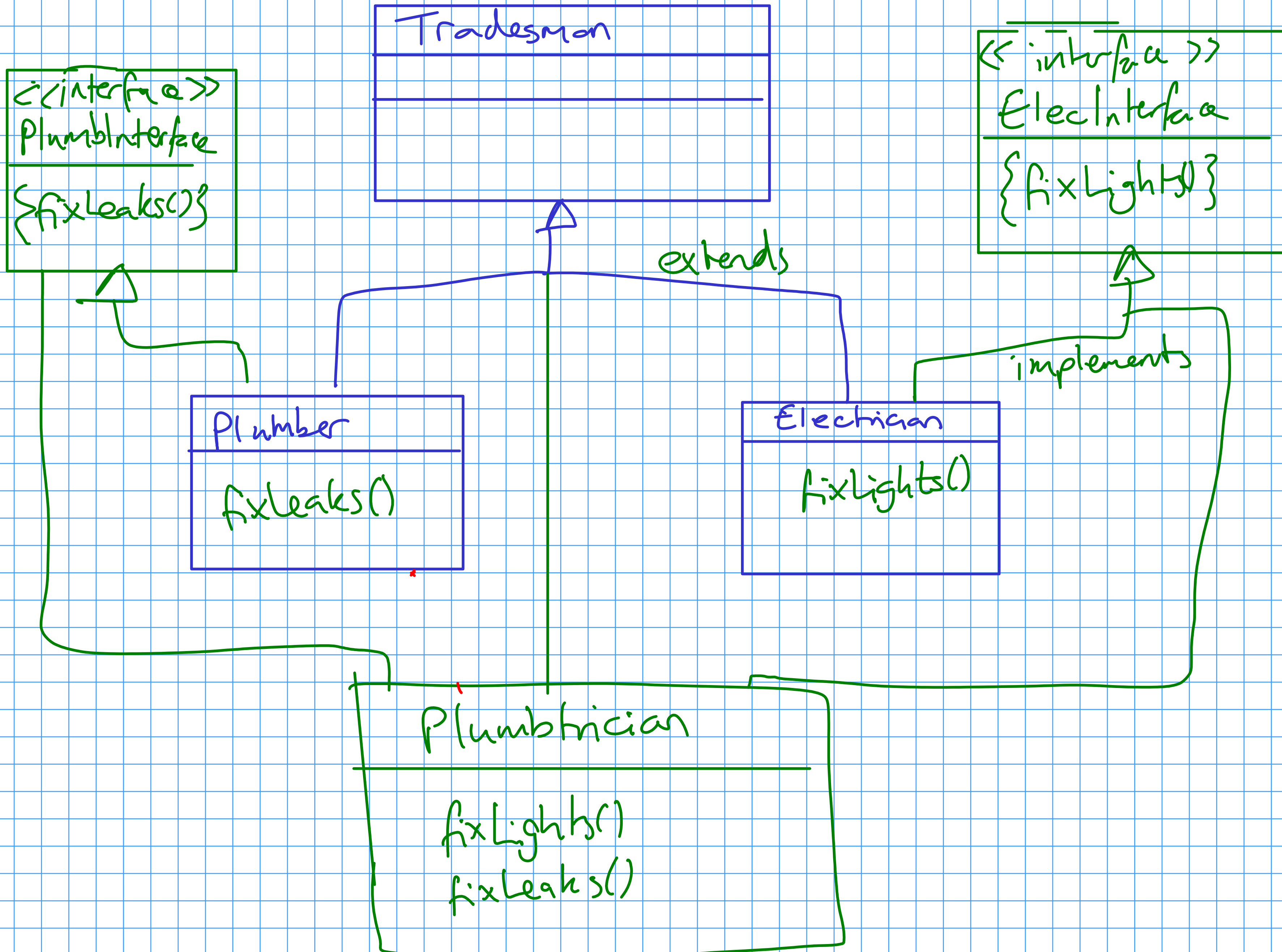
The Dreaded Diamond



Fixing with Abstraction



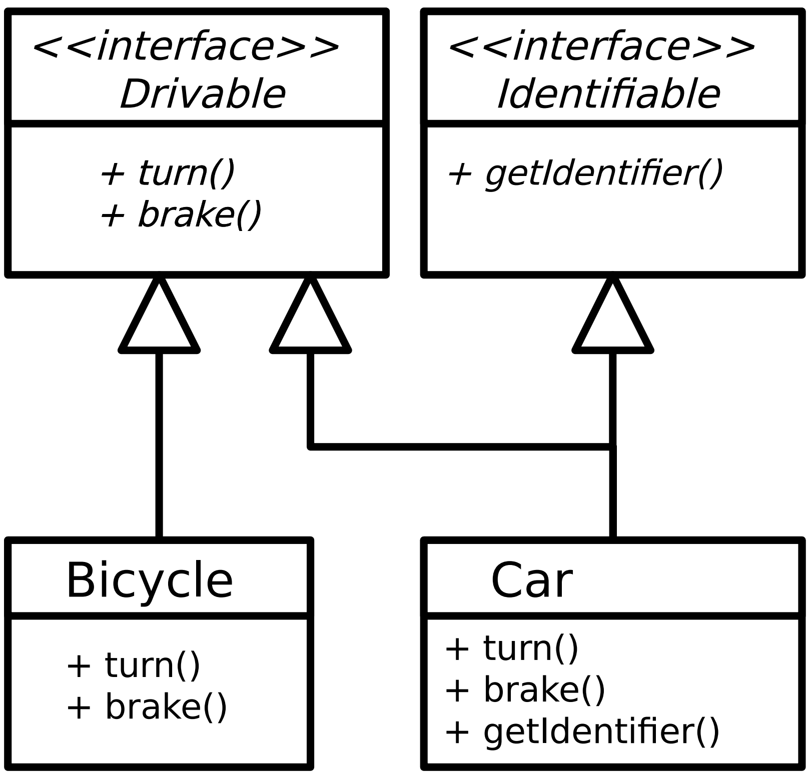
- Actually, this problem goes away if one or more of the conflicting methods is abstract


Tradesman
<<interface>>
PlumbInterface
{fixLeaks()}
<< interface >>
ElecInterface
{fixLights()}
extends
implements
Plumber
fixLeaks()
Electrician
fixLights()
Plumbtrician
fixLights()
fixLeaks()

Interfaces (Java only)

Java's Take on it: Interfaces

- Classes can have at most **one** parent. Period.
- But special 'classes' that are totally abstract can do multiple inheritance – call these **interfaces**




```
interface Drivable {
   public void turn();
   public void brake();
}

interface Identifiable {
   public void getIdentifier();
}

class Bicycle implements Drivable {
   public void turn() {...}
   public void brake() {... }
}

class Car implements Drivable, Identifiable {
   public void turn() {...}
   public void brake() {... }
   public void getIdentifier() {...}
}
```

Interfaces

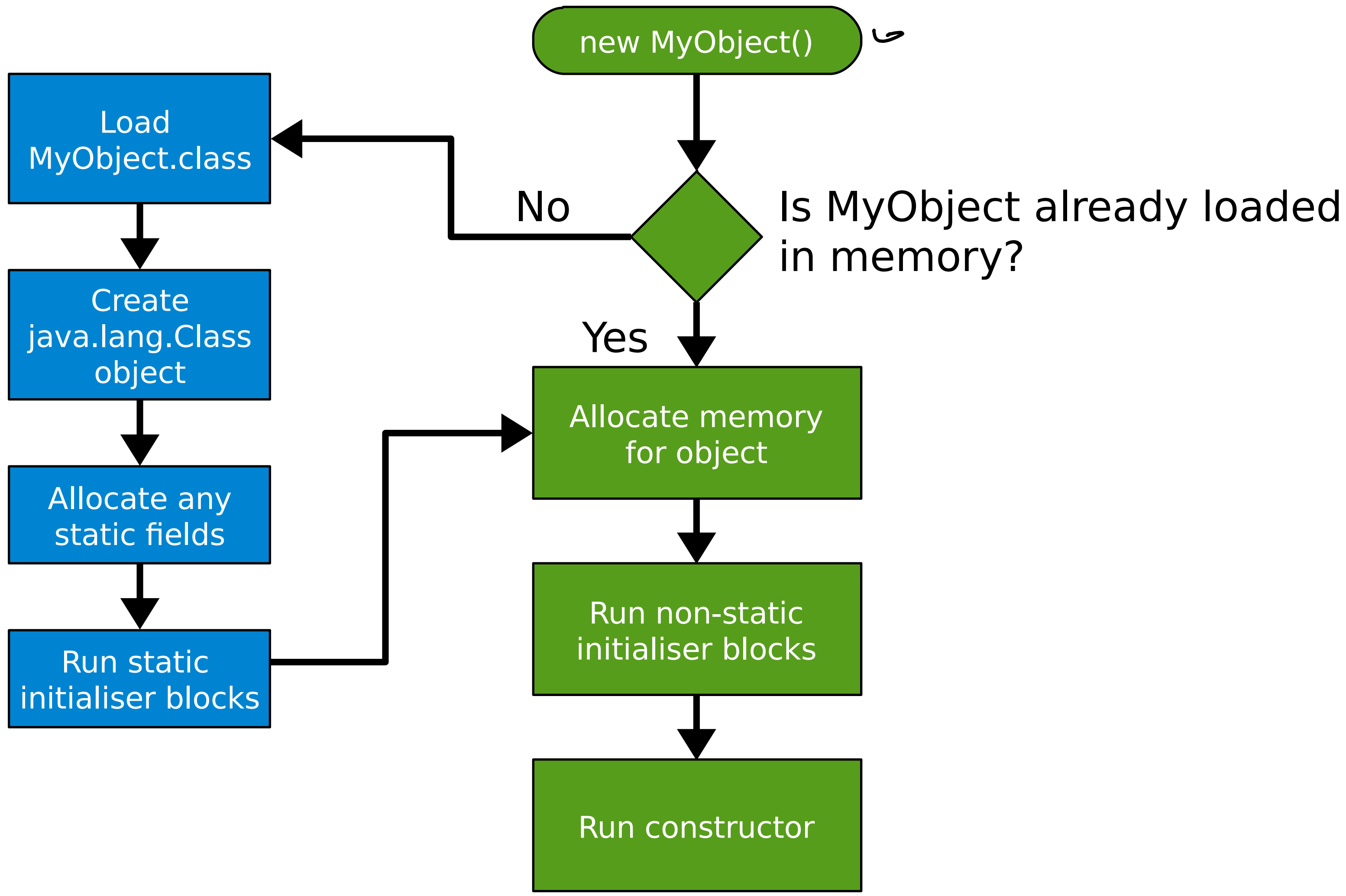
for interfaces

"extends" for classes

'multiple inheritance'

Lecture 6:
Lifecycle of an Object

Creating Objects in Java



Initialisation Example

```
public class Blah {
   private int mX = 7;
   public static int sX = 9;

   {
      mX=5;
    }

    static {
      sX=3;
    }

   public Blah() {
      mX=1;
      sX=9;
   }
}


Blah b = new Blah();
Blah b2 = new Blah();
```

1. Blah loaded
2. sX created
3. sX set to 9
4. sX set to 3
5. Blah object allocated
6. mX set to 7
7. mX set to 5
8. Constructor runs (mX=1, sX=9)
9. b set to point to object

first

10. Blah object allocated
11. mX set to 7
12. mX set to 5
13. Constructor runs (mX=1, sX=9)
14. b2 set to point to object

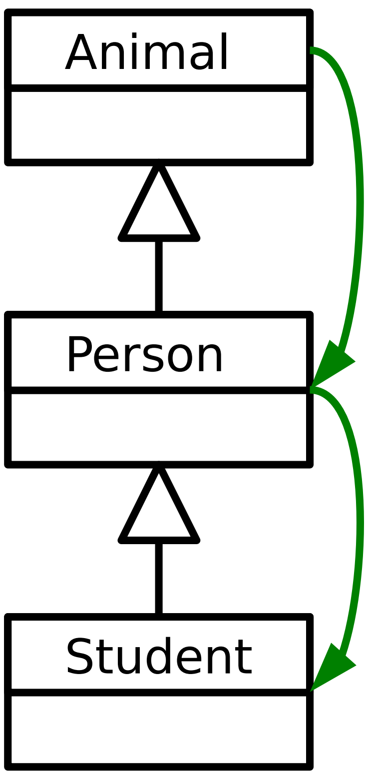
second

Constructor Chaining

- When you construct an object of a type with parent classes, we call the constructors of all of the parents in sequence

```
Student s = new Student();
```

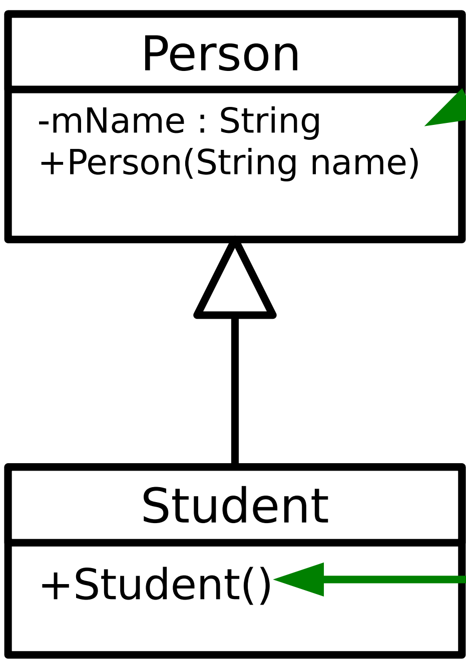
obj | Person | Lecturer



1. Call Animal()
2. Call Person()
3. Call Student()

Chaining without Default Constructors

- What if your classes have explicit constructors that take arguments? You need to explicitly chain
- Use **super** in Java:




```
public Person (String name) {
    mName=name;
}
```

```
public Student () {
    super("Bob");
}
```

Deterministic Destruction

- Objects are created, used and (eventually) destroyed. Destruction is very language-specific
- Deterministic destuction is what you would expect
  - Objects are deleted at predictable times
  - Perhaps manually deleted (C++):

```
void UseRawPointer()
{
    MyClass *mc = new MyClass();
    // ...use mc...
    delete mc;
}
```

  - Or auto-deleted when out of scope (C++):

```
void UseSmartPointer()
{
    unique_ptr<MyClass> *mc = new MyClass();
    // ...use mc...
}   // mc deleted here
```

Destructors

- Most OO languages have a notion of a destructor too
  - Gets run when the object is destroyed
  - Allows us to release any resources (open files, etc) or memory that we might have created especially for the object

C++

```
class FileReader {
  public:

    // Constructor
    FileReader() {
      f = fopen("myfile","r");
    }

    // Destructor
    ~FileReader() {
      fclose(f);
    }

  private :
    FILE *file;
}
```

```
int main(int argc, char ** argv) {

  // Construct a FileReader Object
  FileReader *f = new FileReader();

  // Use object here
  ...

  // Destruct the object
  delete f;

}
```

Non-Deterministic Destruction

- Deterministic destruction is easy to understand and seems simple enough. But it turns out we humans are rubbish of keeping track of what needs deleting when
- We either forget to delete ($\rightarrow$ memory leak) or we delete multiple times ($\rightarrow$ crash)
- **We can instead leave it to the system to figure out when to delete**
  - **"Garbage Collection"**
  - The system someohow figures out when to delete and does it for us
  - In reality it needs to be cautious and sure it can delete. This leads to us not being able to predict exactly when something will be deleted!!
- **This is the Java approach!!**

What about Destructors?

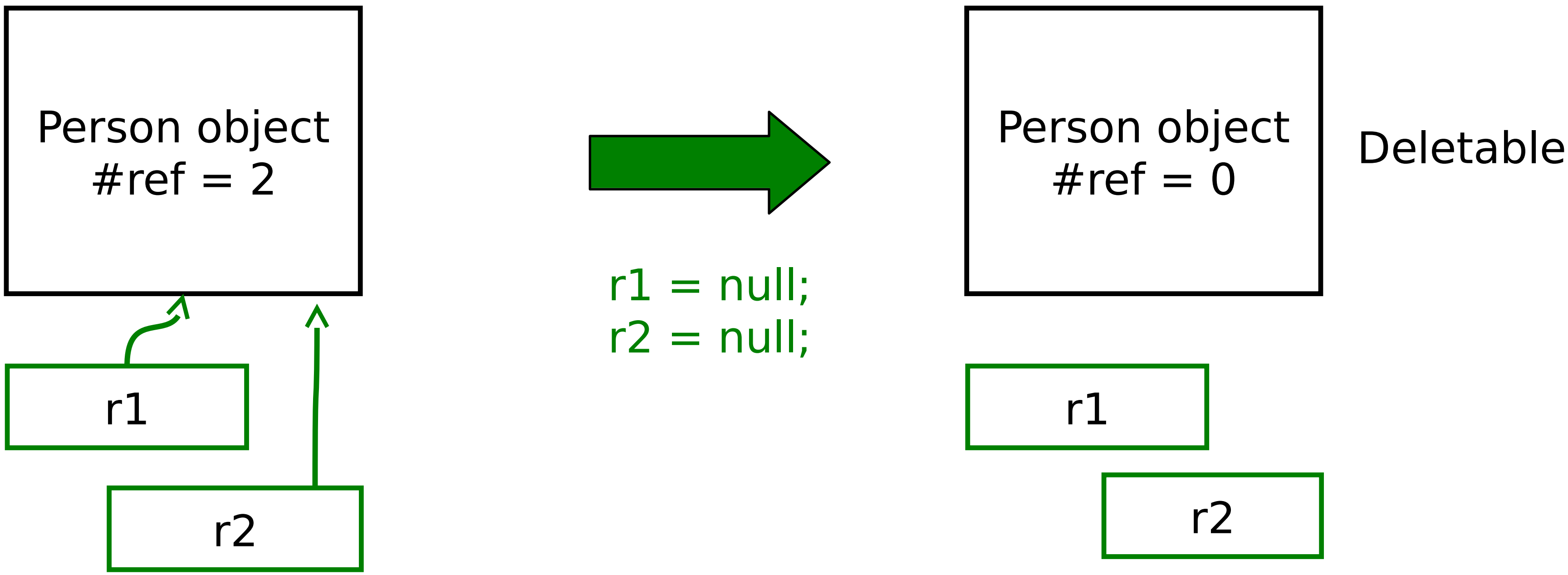
- Conventional destructors don't make sense in non-deterministic systems
  - When will they run?
  - Will they run at all??
- Instead we have finalisers: same concept but they only run when the system deletes the object (which may be never!)

Garbage Collection

- So how exactly does garbage collection work? How can a system know that something can be deleted?
- The garbage collector is a separate process that is constantly monitoring your program, looking for things to delete
- Running the garbage collector is obviously not free. If your program creates a lot of short-term objects, you will soon notice the collector running
    - Can give noticeable pauses to your program!
    - But minimises memory leaks (it does not prevent them...)
- There are various algorithms: we'll look at two that can be found in Java
    - Reference counting
    - Tracing

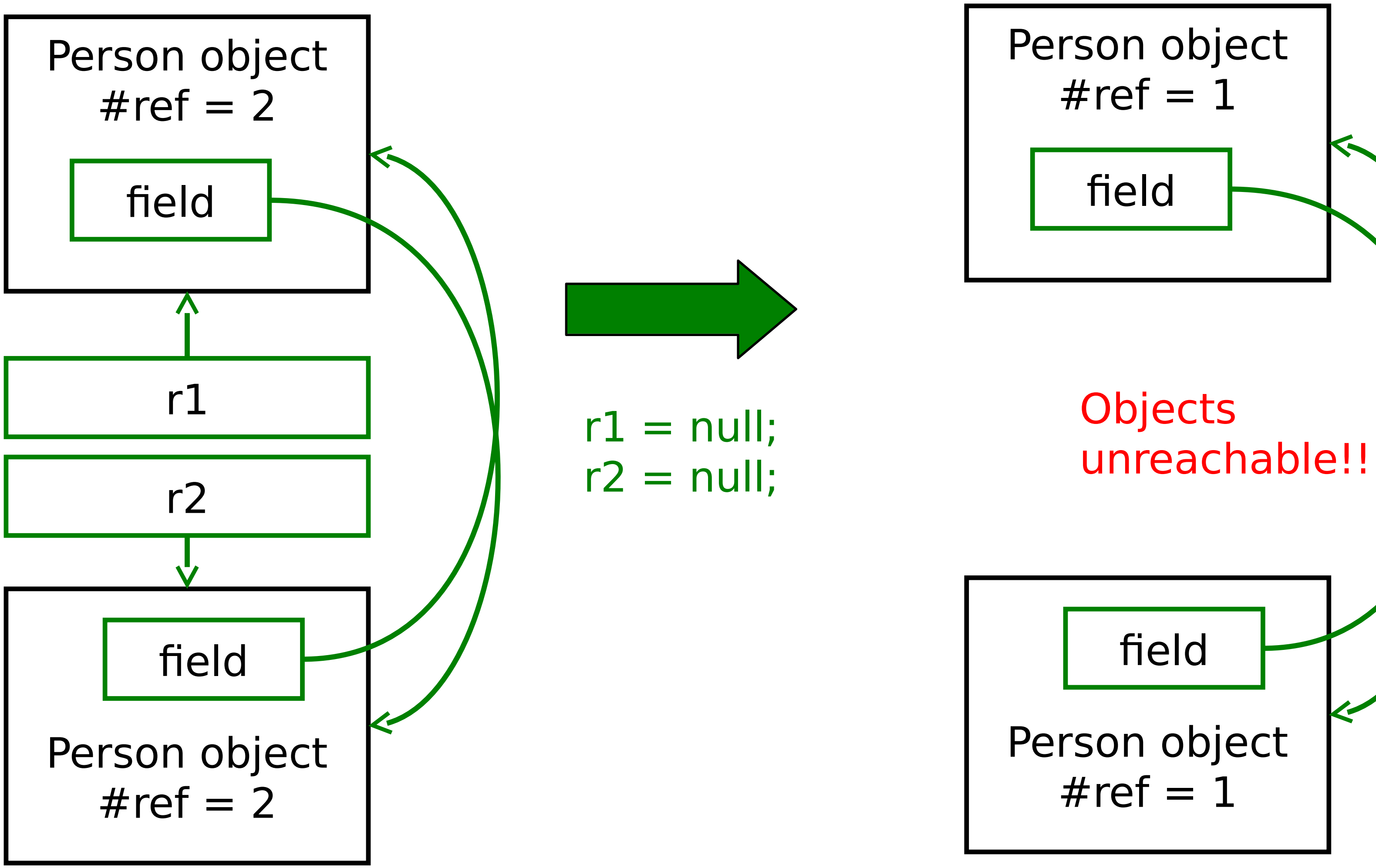
Reference Counting

- Java's original GC. It keeps track of how many references point to a given object. If there are none, the programmer can't access that object ever again so it can be deleted



Reference Counting Gotcha

- Circular references are a pain



Tracing

- Start with a list of all references you can get to
- Follow all refrences recursively, marking each object
- Delete all objects that were not marked