Object Oriented Programming
Dr Robert Harle

IA CST, PPS (CS) and NST (CS)
Lent 2012/13

The OOP Course

- Last term you studied **functional** programming (ML)
- This term you are looking at **imperative** programming (Java primarily).
  - You already have a few weeks of Java experience
  - This course is hopefully going to let you separate the fundamental software design principles from Java's quirks and specifics

- Four Parts
  - From Functional to Imperative
  - Object-Oriented Concepts
  - The Java Platform
  - Design Patterns and OOP design examples

Java Practicals

- This course is meant to *complement* your practicals in Java
    - 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.

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

1213

http://www.cl.cam.ac.uk/teaching/1~~112~~/OOProg/

Section: From Functional to Imperative Programming

Explicit Start Points

Magic start method

Array of Strings

Java: public static void main(String args[])

Array of array

C/C++: int main(int argc, char **argv)

No of arguments

python:

```
def main():
    # main code here

if __name__ == "__main__":
    main()
```

Immutable to Mutable Data

ML

Assign

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

Comparison

Java

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

int x=9;
```

Assign

Assign

will not compile

Types and Variables

- We write code like:

```
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

$N \quad (-2^{N-1}) \to (2^{N-1}-1)$

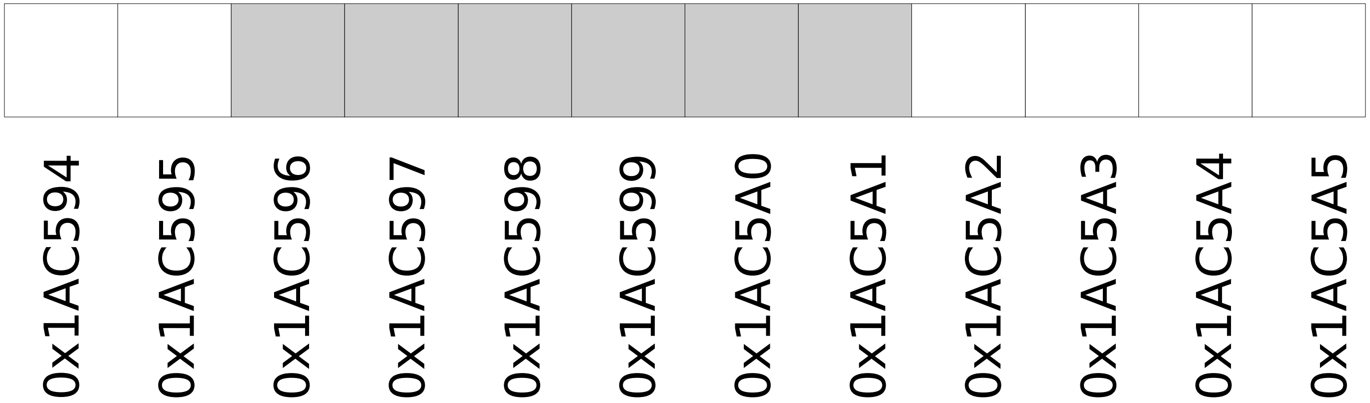
See Workbook 1

Arrays

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

int x, y[], z[][];

int[] x[]; ⟷ int[][] x ⟷ int x[][];



Functions to Procedures

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

ML:

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

Java:

```
public int m(int x, int y) = x*y;
```

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

Side effect

Interpreter to Virtual Machine

- *Java* was born in an era of internet connectivity. SUN wanted to distribute programs to internet machines
  - But many architectures were attached to the internet – how do you write one program for them all?
  - And how do you keep the size of the program small (for quick download)?

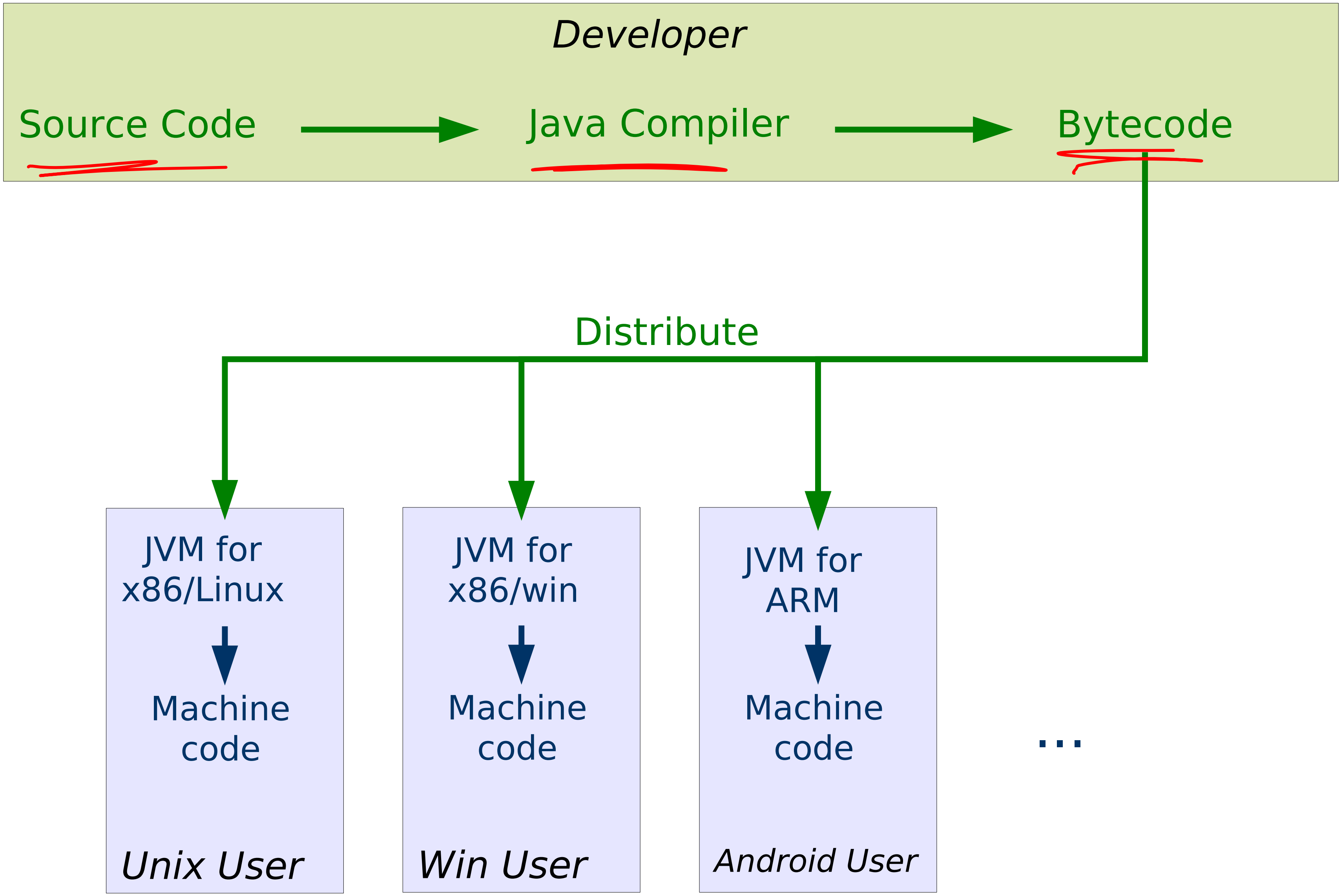
- Could use an interpreter (→ Javascript). But:
  - High level languages not very space-efficient
  - The source code would implicitly be there for anyone to see, which hinders commercial viability.

- Went for a clever hybrid interpreter/compiler

Java Bytecode I

- SUN envisaged a hypothetical Java Virtual Machine (JVM). Java is compiled into machine code (called bytecode) for that (imaginary) machine. The bytecode is then distributed.
- To use the bytecode, the user must have a JVM that has been specially compiled for their architecture.
- The JVM takes in bytecode and spits out the correct machine code for the local computer. i.e. is a bytecode interpreter

Java Bytecode II



Developer

Source Code → Java Compiler → Bytecode

Distribute

JVM for x86/Linux → Machine code

Unix User

JVM for x86/win → Machine code

Win User

JVM for ARM → Machine code

Android User

...

Java Bytecode III

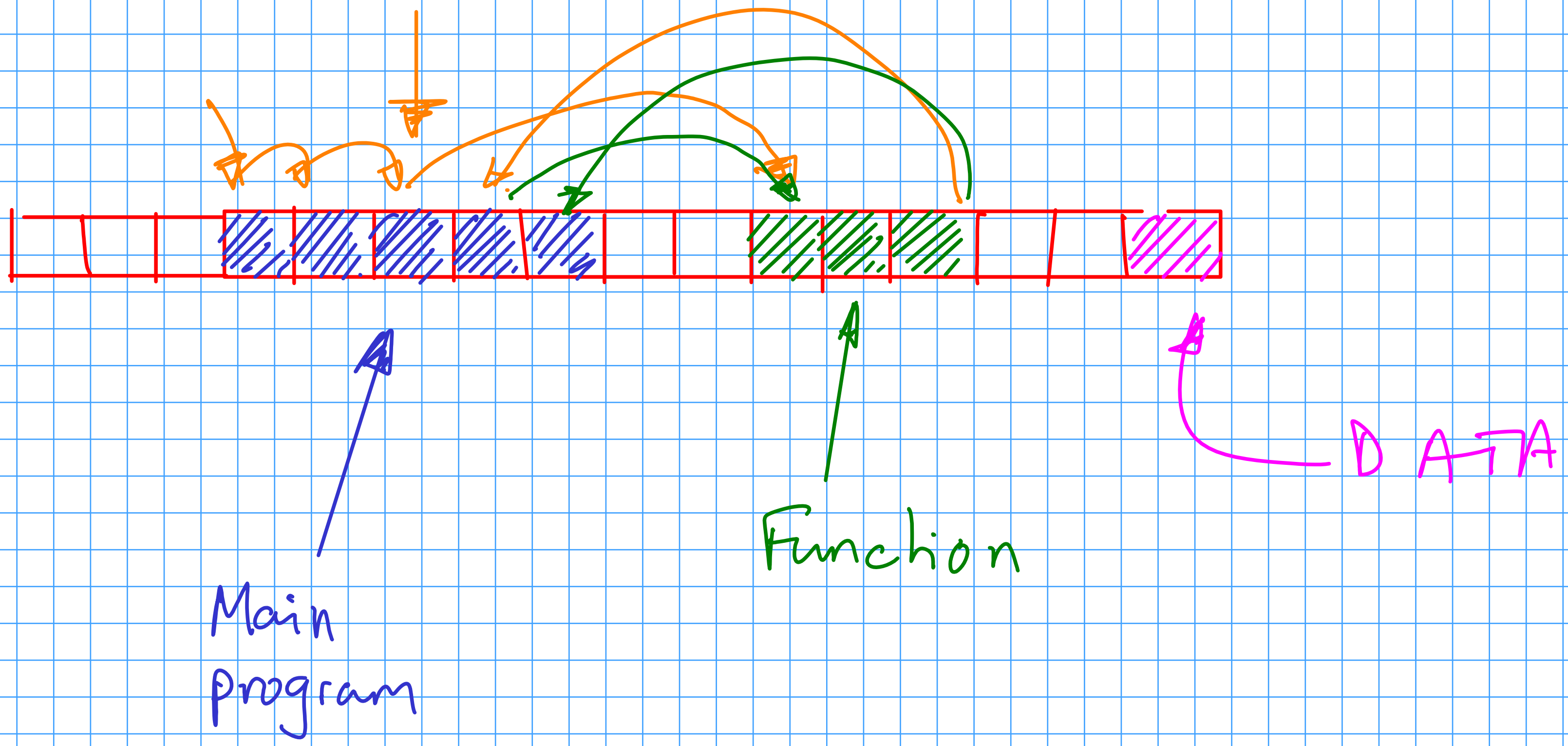
+ Bytecode is compiled so not easy to reverse engineer

+ The JVM ships with tons of libraries which makes the bytecode you distribute small

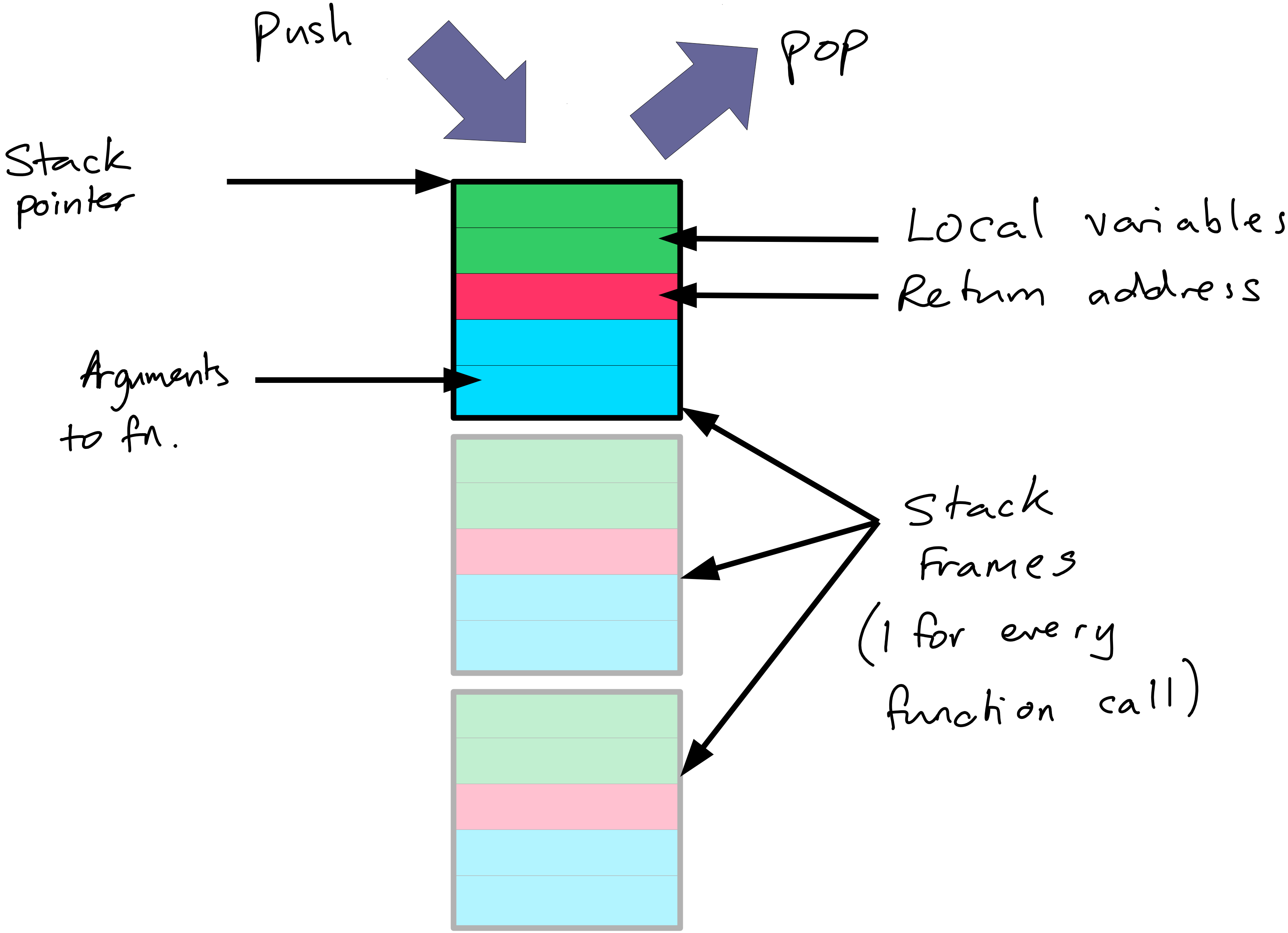
+ The toughest part of the compile (from human-readable to computer readable) is done by the compiler, leaving the computer-readable bytecode to be translated by the JVM (→ easier job → faster job)

- Still a performance hit compared to fully compiled (“native”) code

Lecture 2:
Memory Manipulation

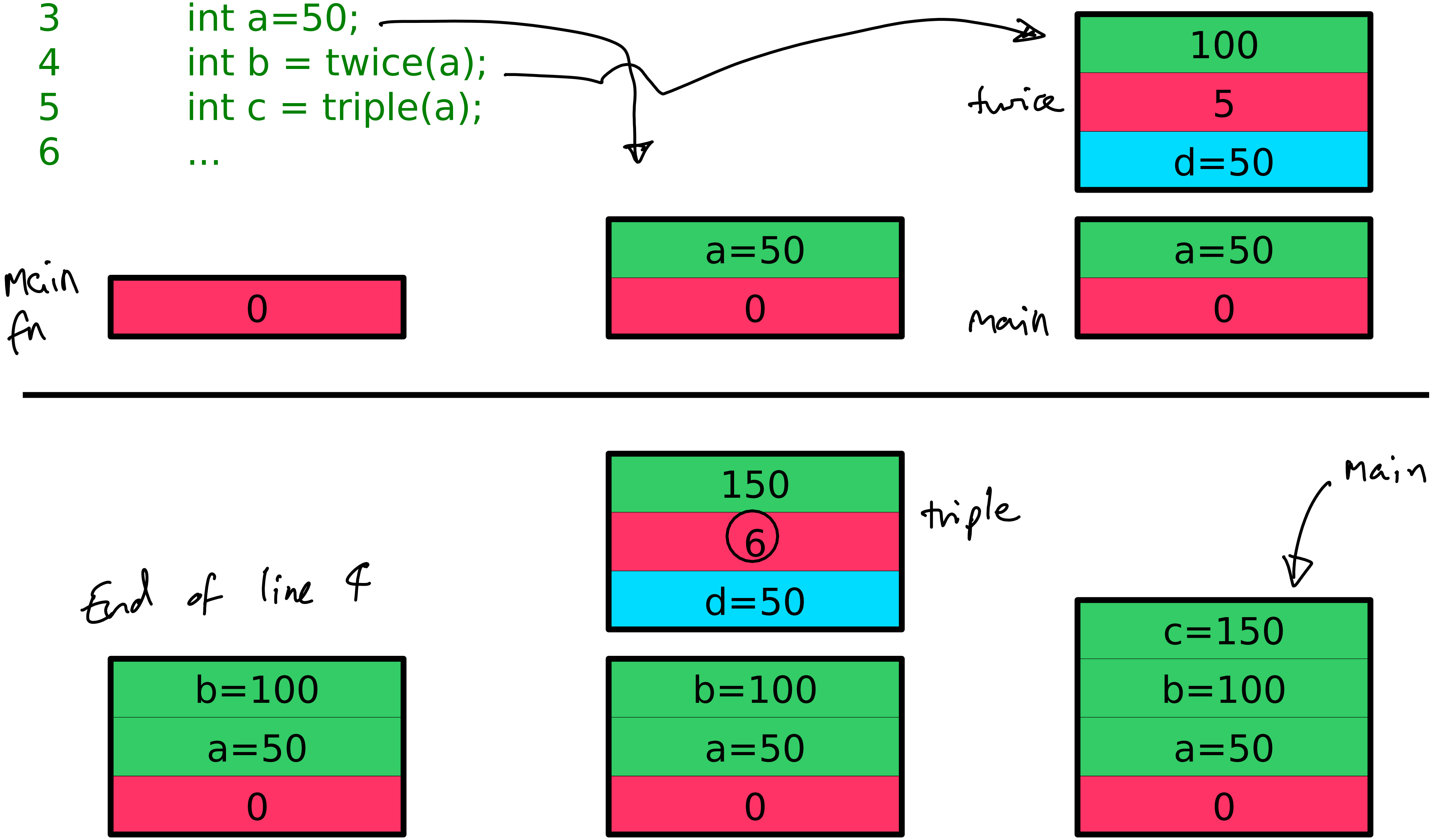

Main program
Function
DATA

The Call Stack



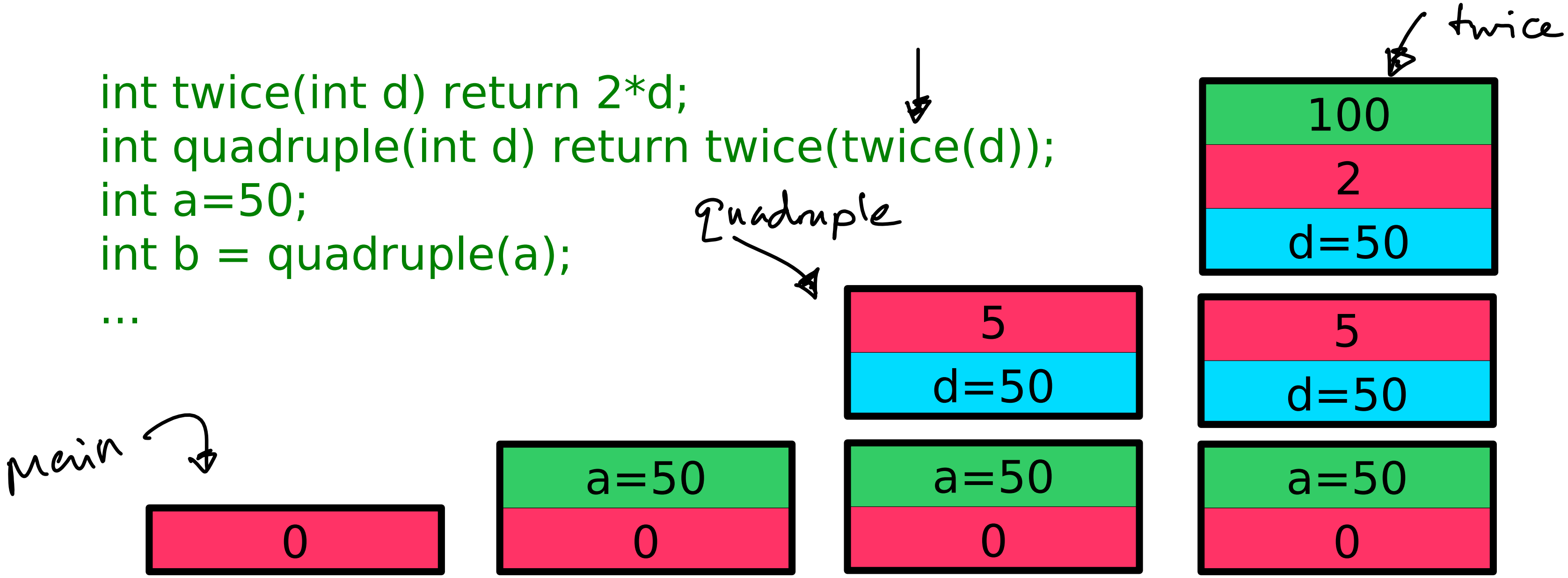
The Call Stack: Example

```
1        int twice(int d) return 2*d;
2        int triple(int d) return 3*d;
3        int a=50;
4        int b = twice(a);
5        int c = triple(a);
6        ...
```



Nested Functions

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



main

0

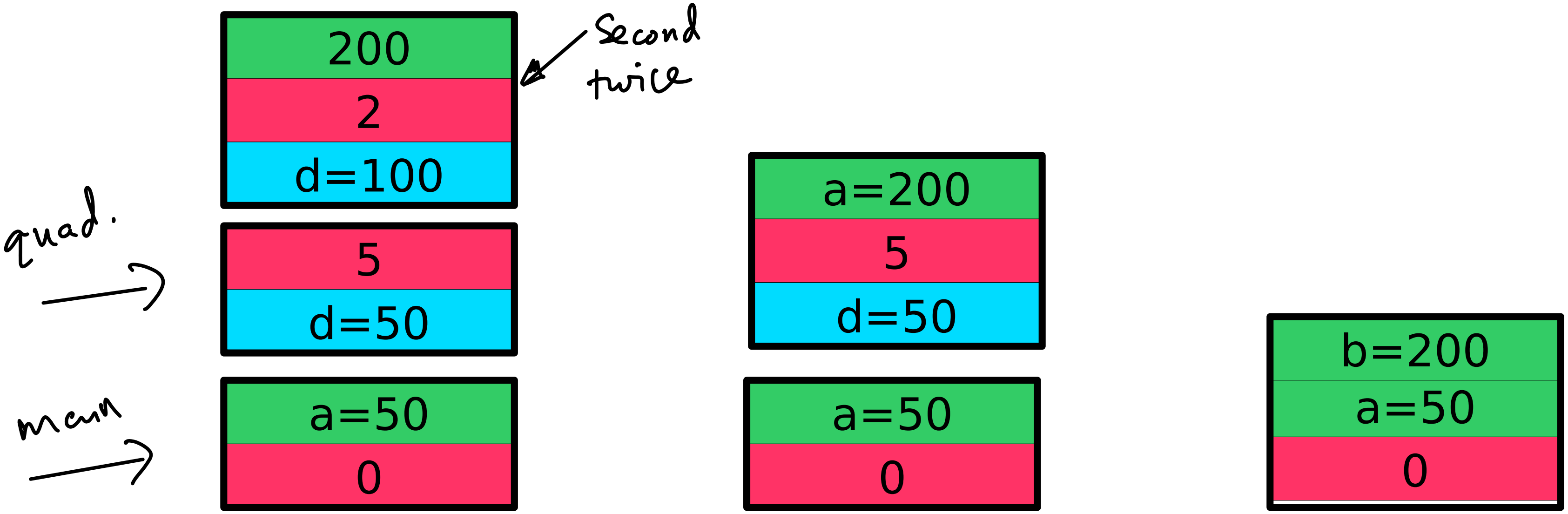
a=50
0

quadruple

5
d=50
a=50
0

twice

100
2
d=50
5
d=50
a=50
0



Second twice

200
2
d=100
5
d=50
a=50
0

quad.

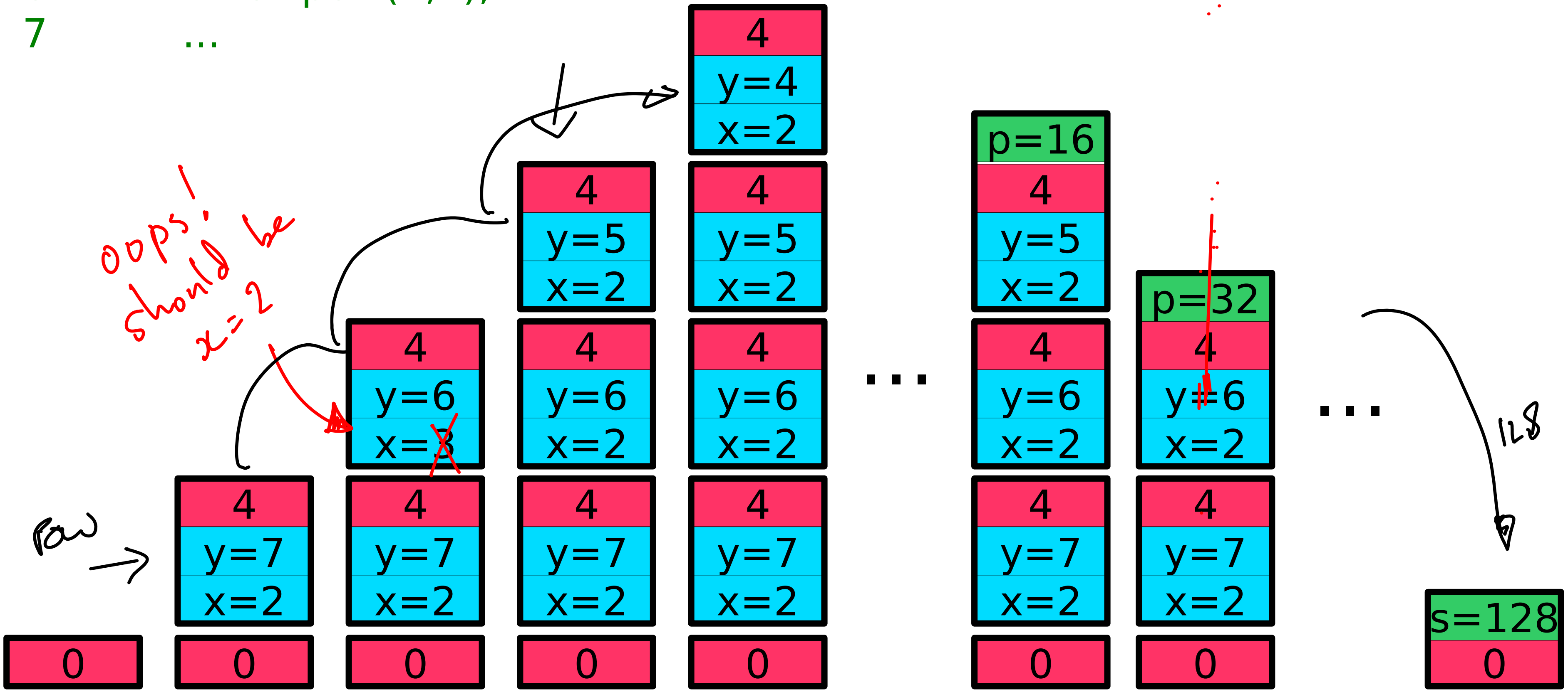
main

a=200
5
d=50
a=50
0

b=200
a=50
0

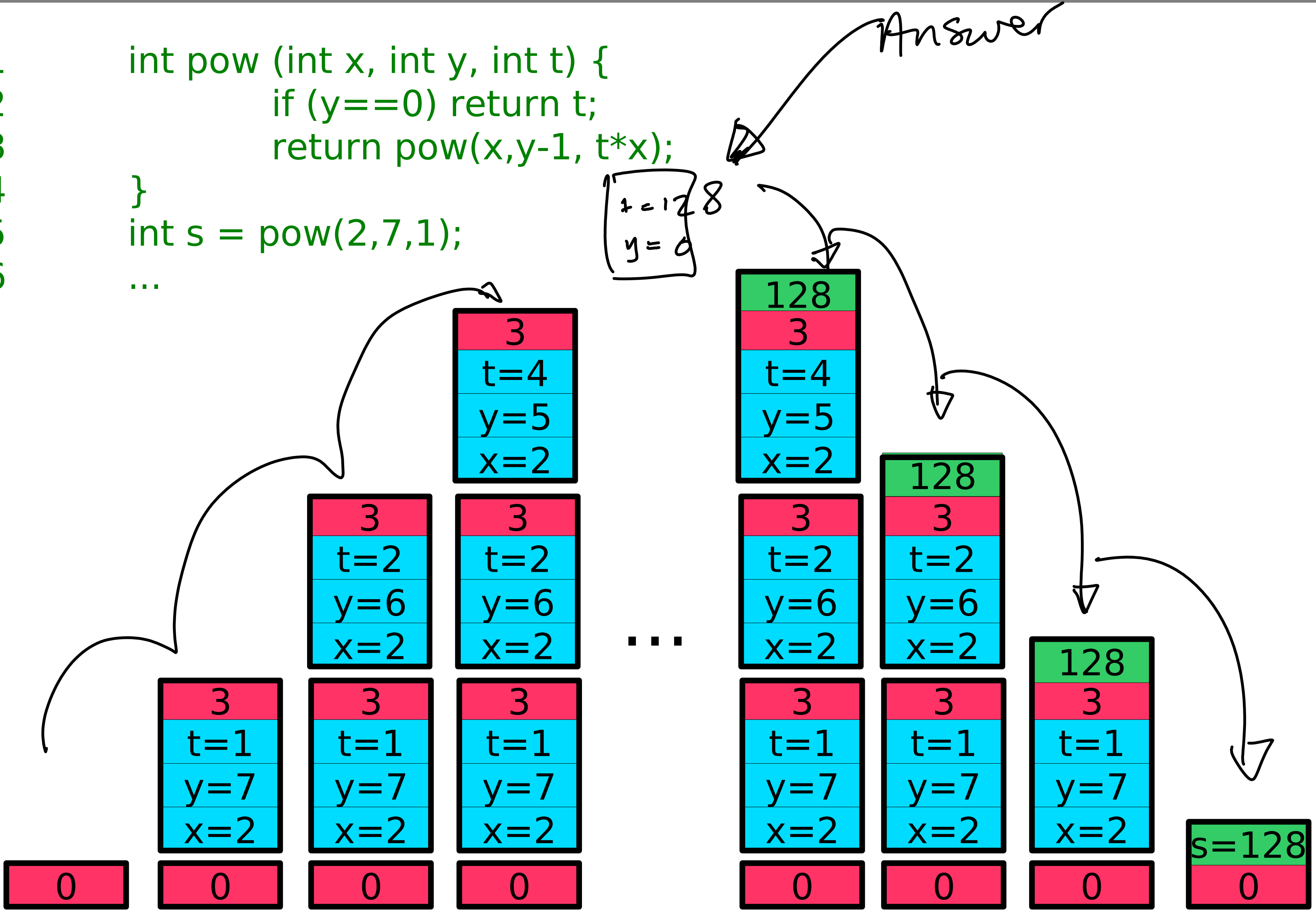
Recursive Functions

```
1        int pow (int x, int y) {
2                if (y==0) return 1;
3                int p = pow(x,y-1);
4                return x*p;
5        }
6        int s=pow(2,7);
7        ...
```



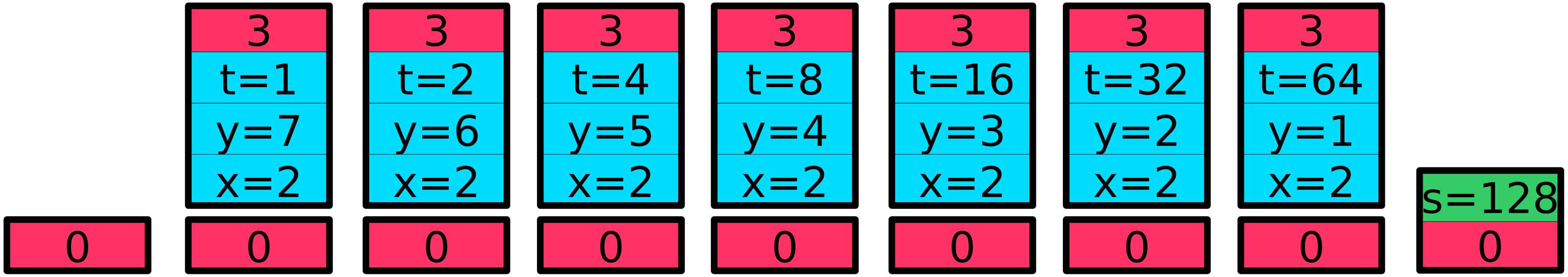
Tail-Recursive Functions I

```
1        int pow (int x, int y, int t) {
2                if (y==0) return t;
3                return pow(x,y-1, t*x);
4        }
5        int s = pow(2,7,1);
6        ...
```



Tail-Recursive Functions II

```
1        int pow (int x, int y, int t) {
2                 if (y==0) return t;
3                 return pow(x,y-1, t*x);
4        }
5        int s = pow(2,7,1);
6        …
```



Control Flow: for and while

for(*init; boolean_expression; step* **)**

```
for (int i=0; i<8; i++) ...

int j=0;  for(; j<8; j++) ...

for(int k=7;k>=0; j--) ...
```

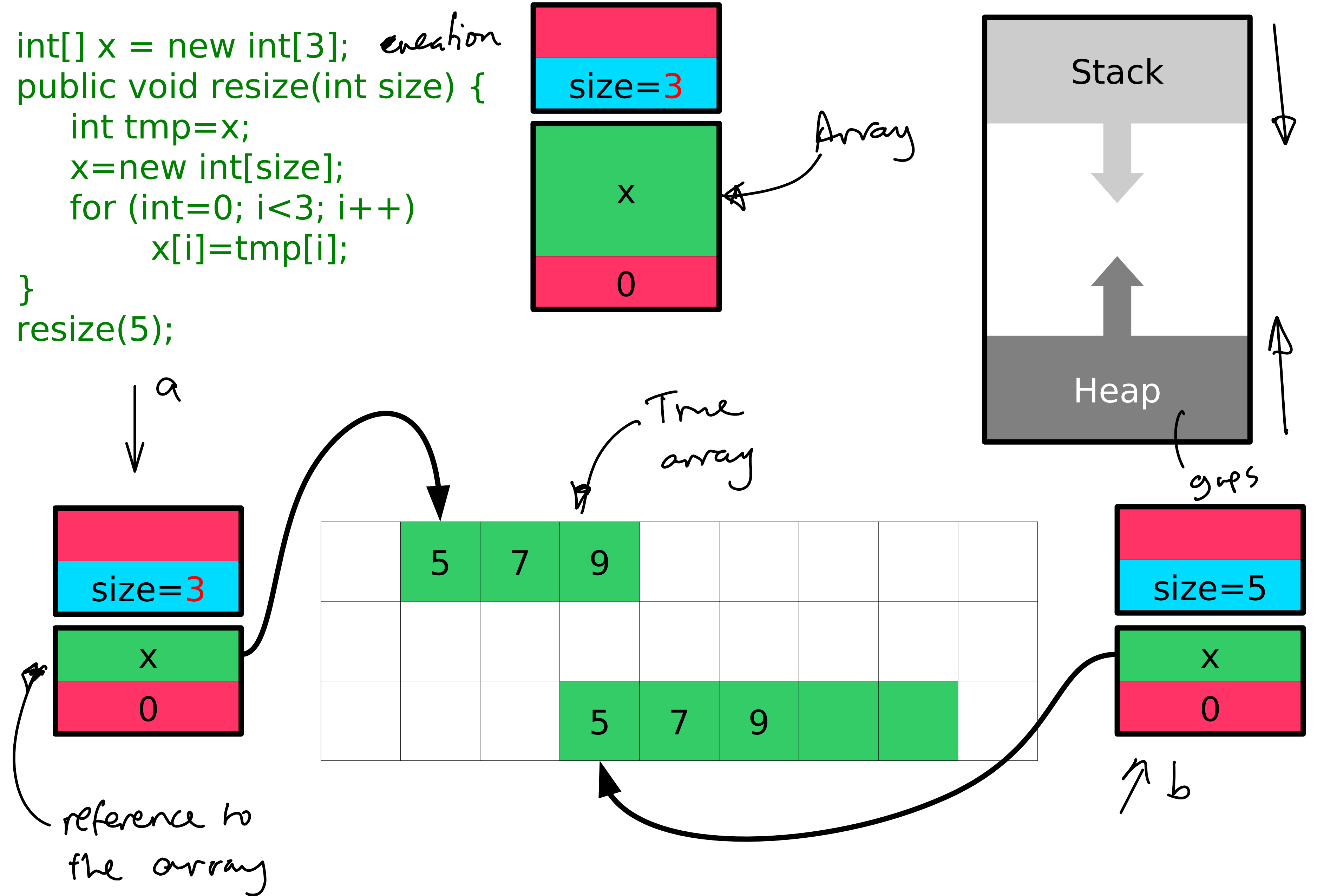
while(*boolean_expression* **)**

```
int i=0;  while (i<8) { i++; ...}

int j=7; while (j>=0) { j--; ...}
```

8 x

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);
```



References

- Pointers are useful but dangerous
- References can be thought of as restricted pointers
  - Still just a memory address
  - But the compiler limits what we can do to it
- C, C++: pointers *and* references
- Java: references *only*
- ML: references *only*

References vs Pointers

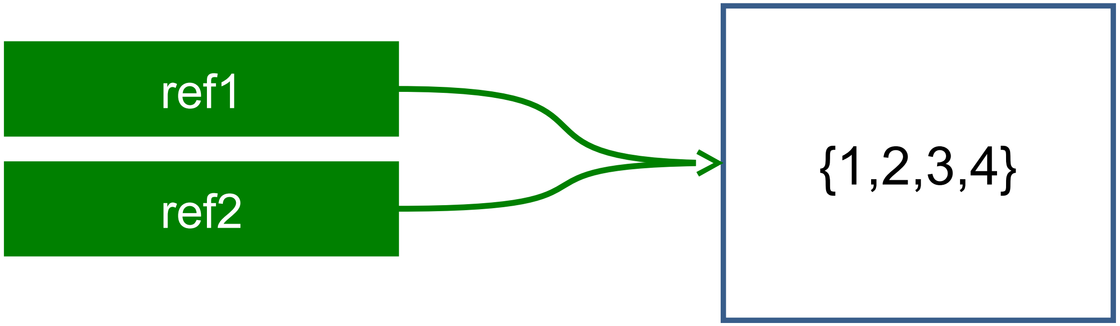
	Pointers	References
Represents a memory address	Yes	Yes
Can be arbitrarily assigned	Yes	**No**
Can be assigned to established object	Yes	Yes
Can be tested for validity	**No**	Yes
Can perform "arithmetic"	Yes	No

References Example (Java)

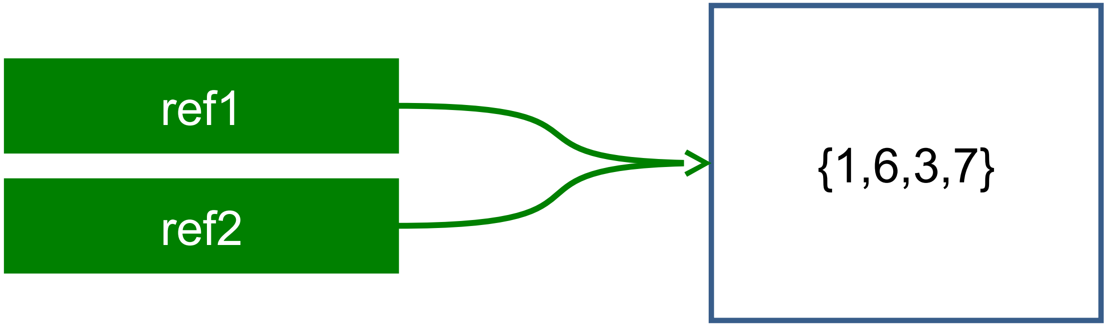
Not an array ⇒ a reference

```
int[] ref1 = null;
ref1 = new int[]{1,2,3,4};
int[] ref2 = ref1;
```

Creates on the heap and returns a reference



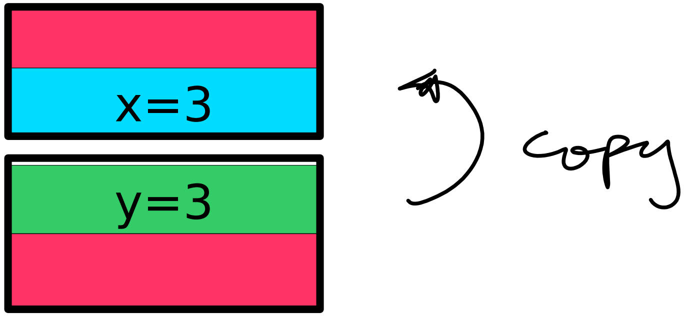
```
ref1[3]=7;
ref2[1]=6;
```



Argument Passing

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

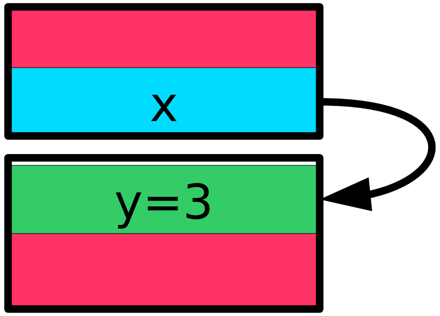
```
void test(int x) {...}
int y=3;
test(y);
```



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

C++

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



Passing Procedure Arguments In Java

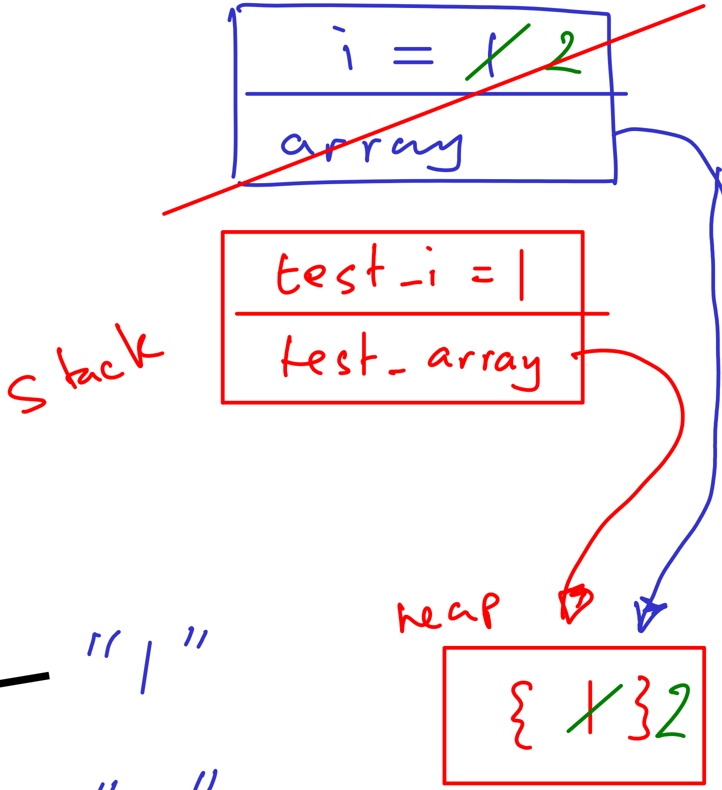
```
class Reference {

   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]);
   }

}
```

primitive

ref

"1"

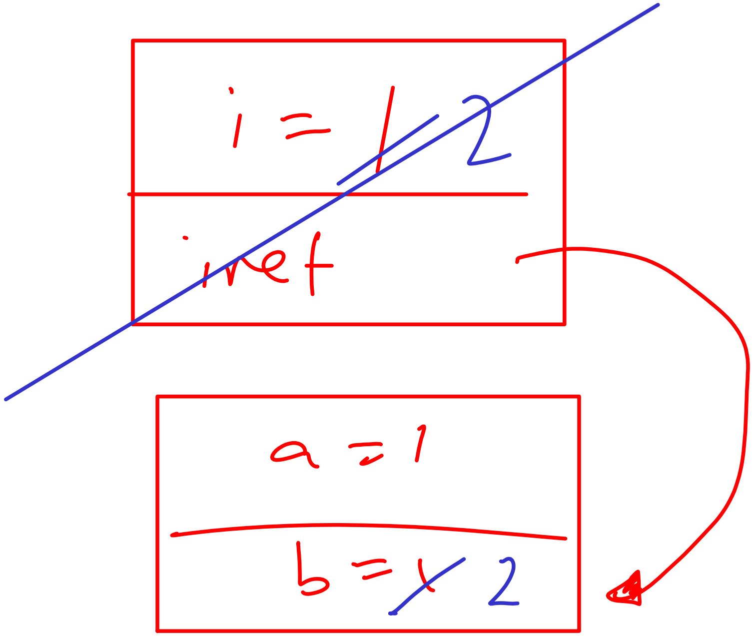
"2"



Passing Procedure Arguments In C++

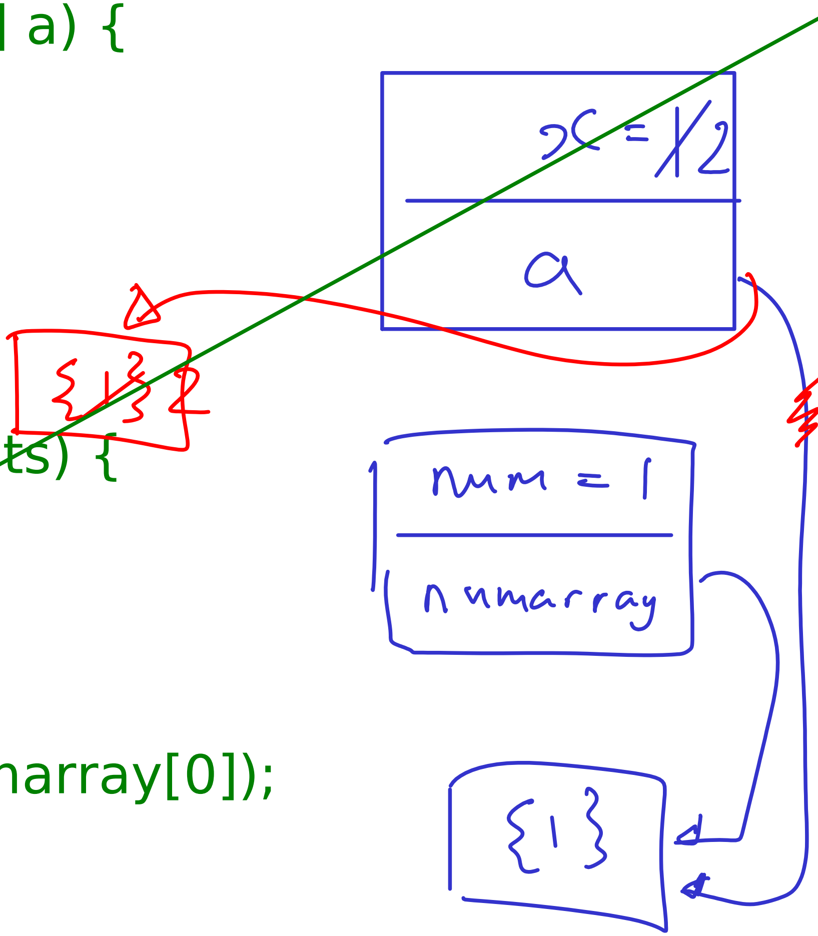
pass by reference

```
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);
}
```



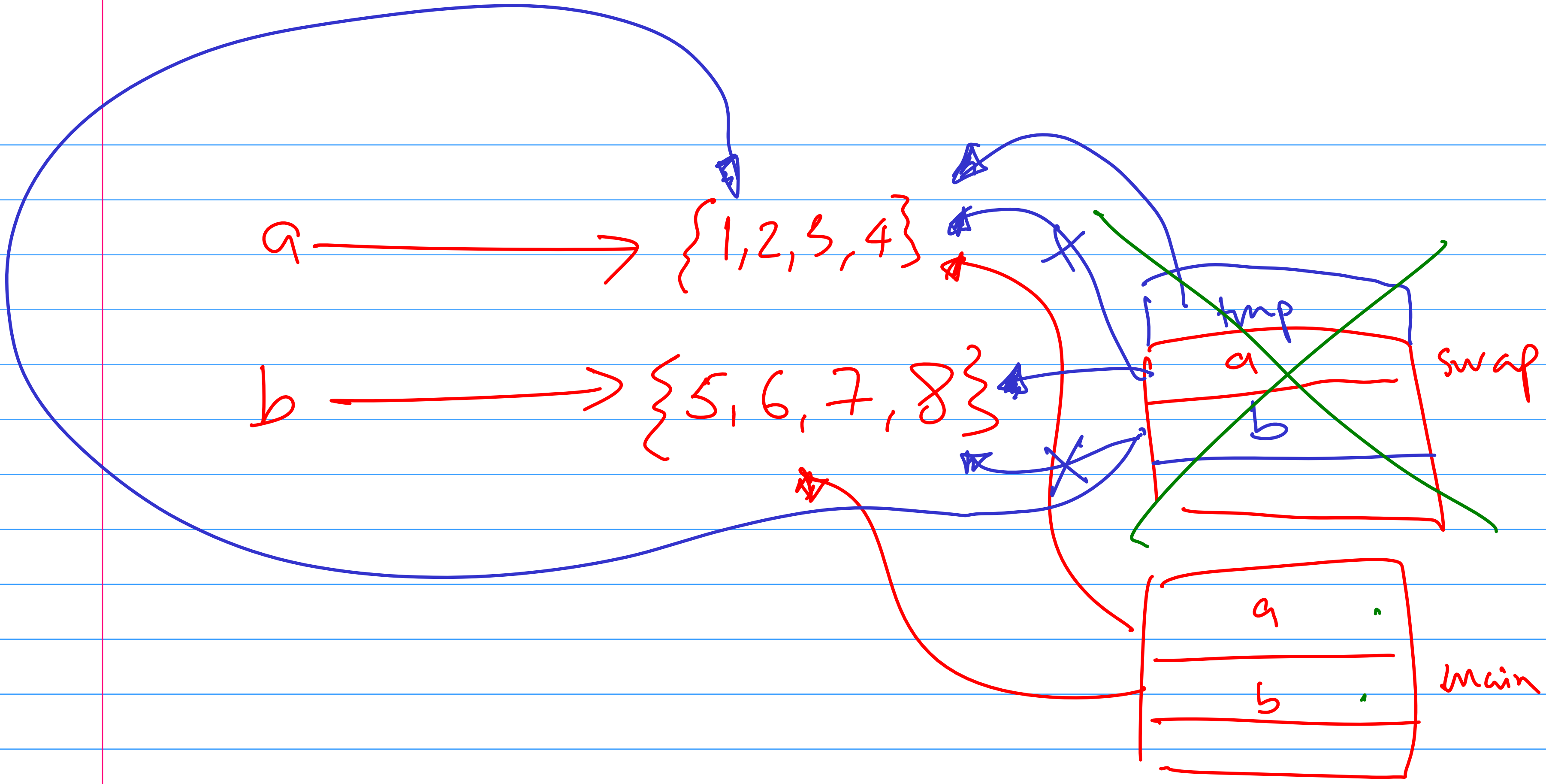
Check...

```
public static void 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 3:
OOP and Classes


a
{1,2,3,4}
b
{5,6,7,8}
tmp
a
b
swap
a
b
main

Custom Types

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

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

- In OOP we go further
  - We include both state *and* procedures in our type definition
  - The idea is that each type groups together *related* state and procedures, providing a complete implementation of a single *concept*
  - We call such types **classes**

See Workbook 3

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**

Loose Terminology (again!)

State

Fields
Instance Variables
Properties
Variables
Members

Attributes

Behaviour

Functions
Methods
Procedures

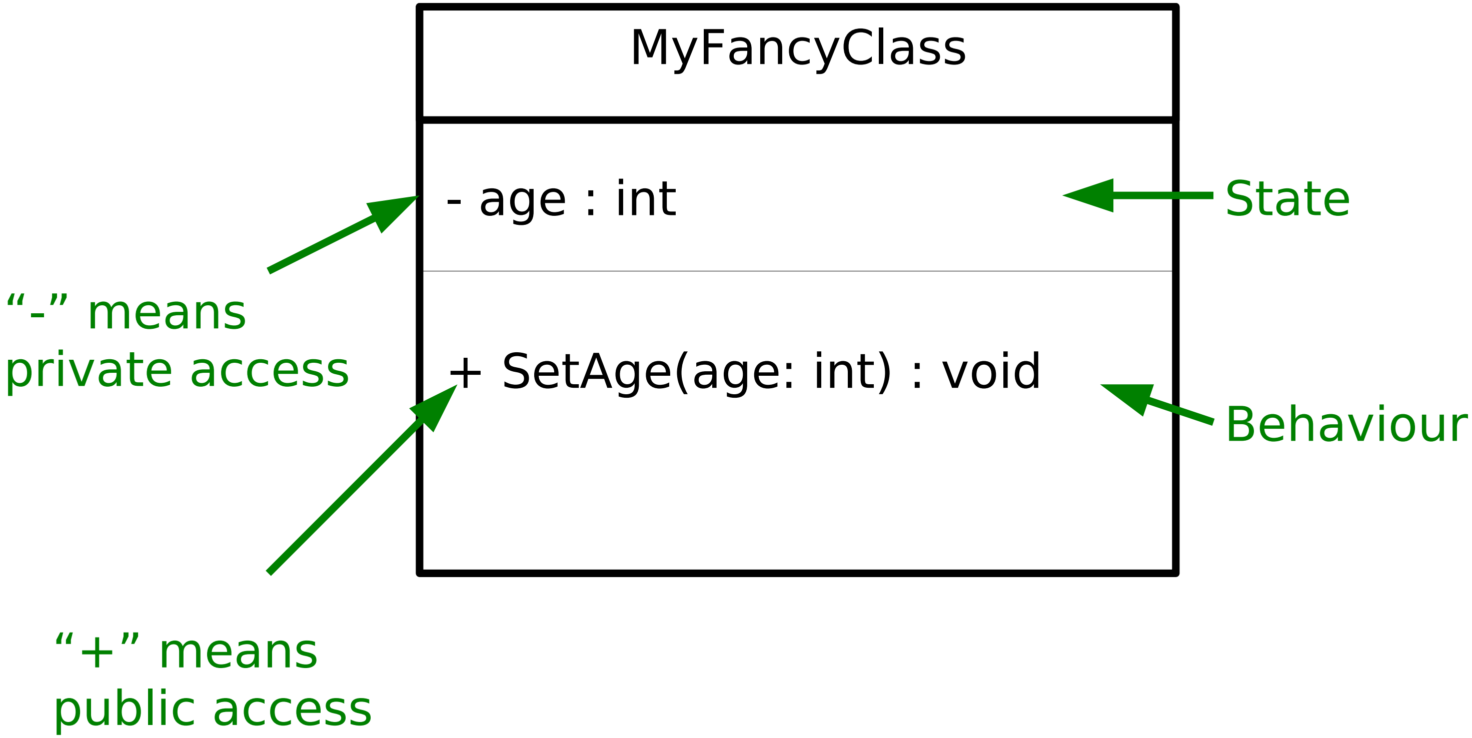
Actions

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 simulation of the Earth's orbit around the Sun"

UML: Representing a Class Graphically



MyFancyClass

- age : int

+ SetAge(age: int) : void

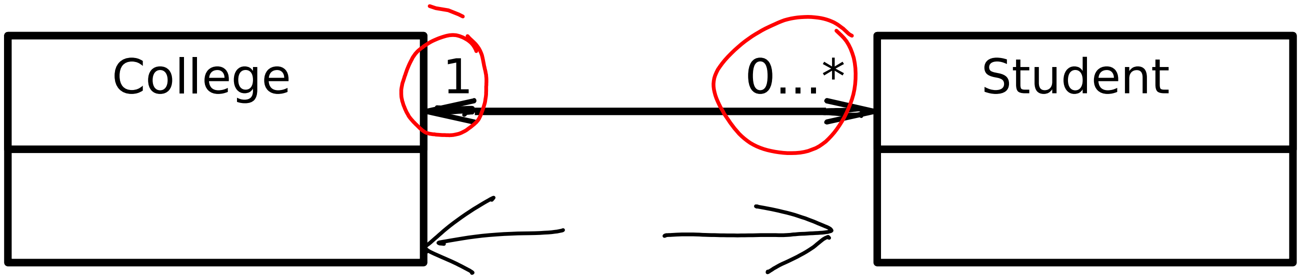
State

Behaviour

“-” means
private access

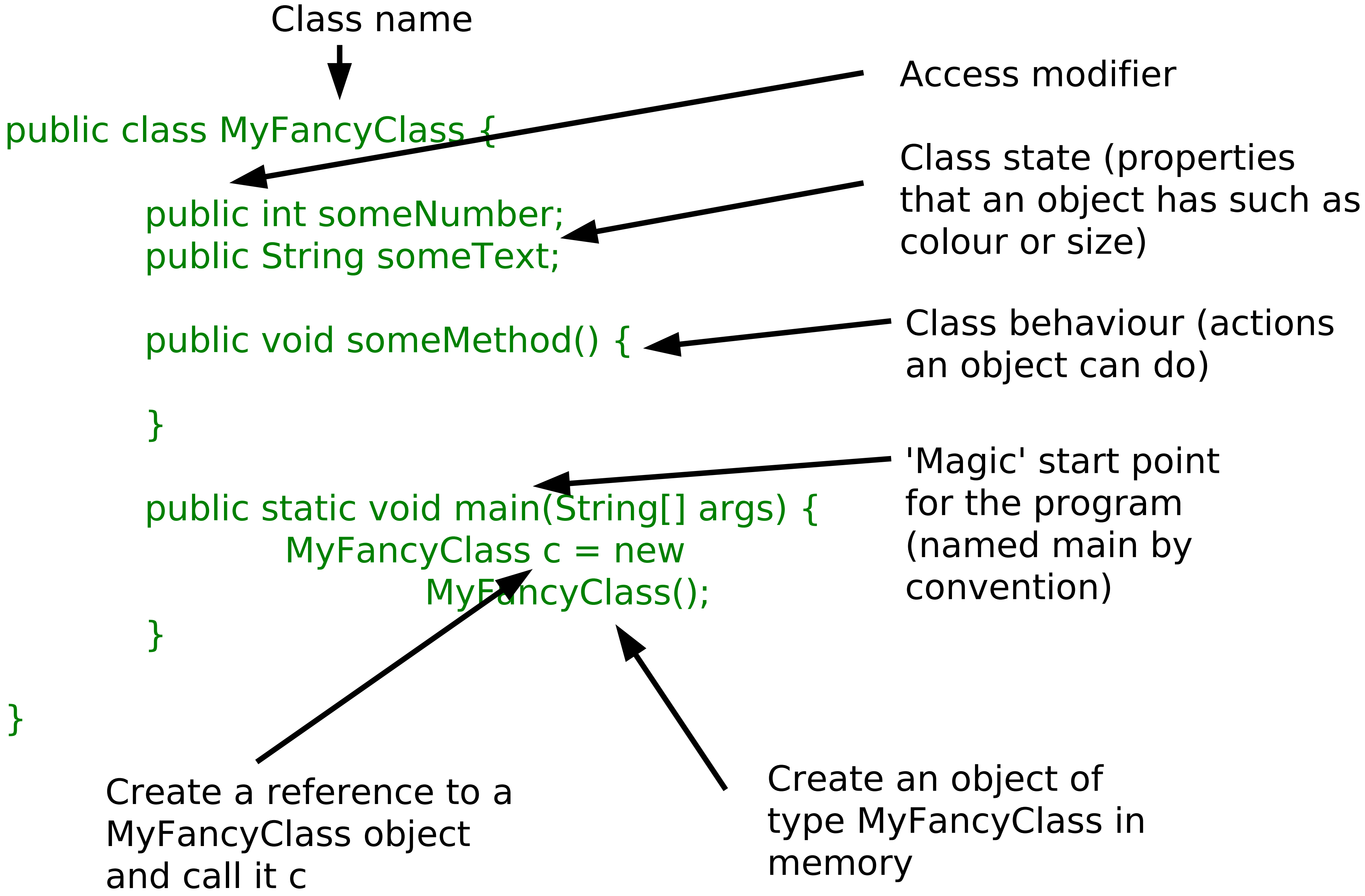
“+” means
public access

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)

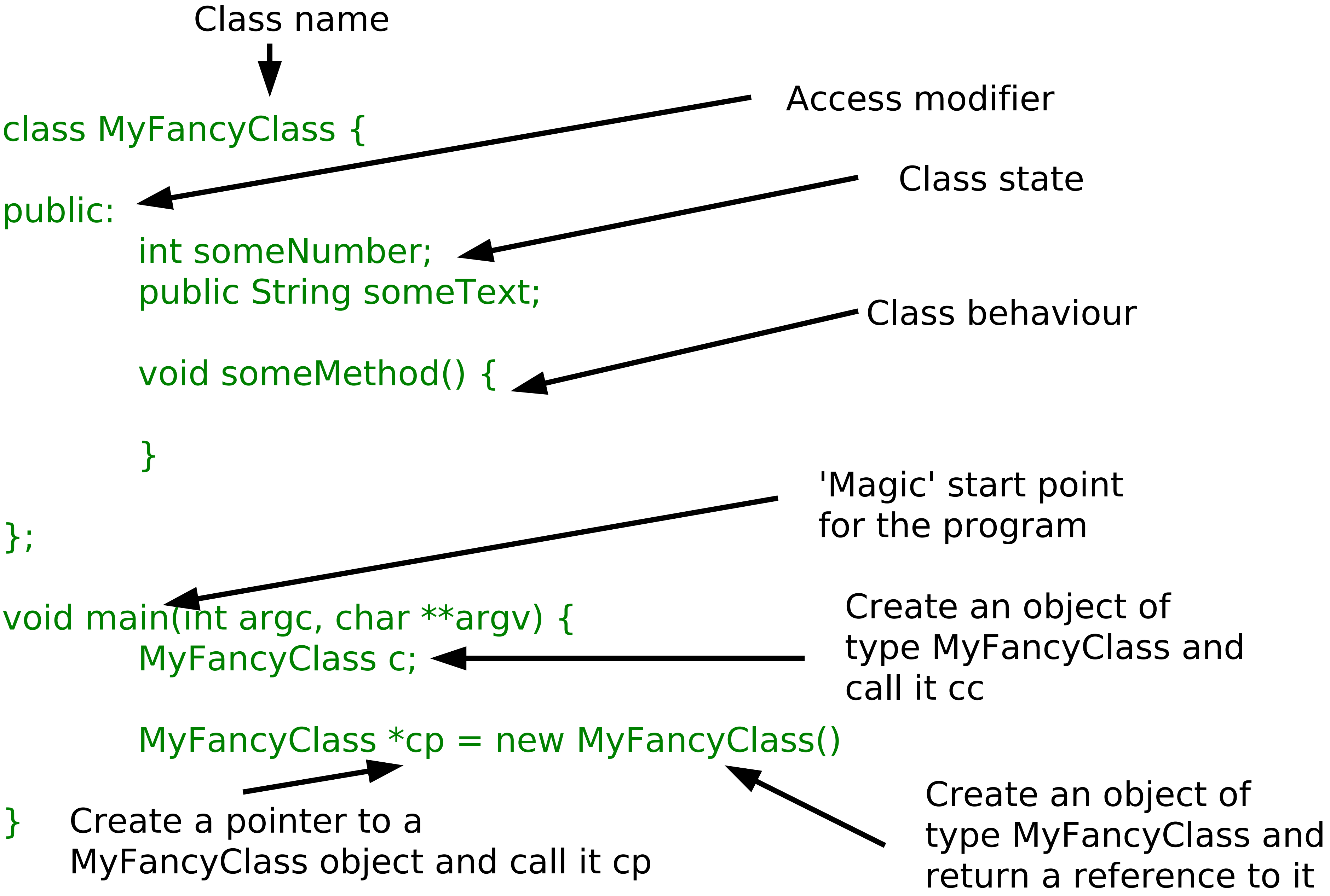
Class behaviour (actions an object can do)

'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

Access modifier

Class state

Class behaviour

```
class MyFancyClass {

public:
        int someNumber;
        public String someText;

        void someMethod() {

        }

};

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

        MyFancyClass *cp = new MyFancyClass()

}
```

'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);
}
```

Access Modifier

Bounds checking

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);}
}
```

Why Encapsulate?

i) Sanity check or consistency check variables.

ii) Change underlying representations

iii) Change variable names.

iv) Control mutability

v) Encourage good programming practice

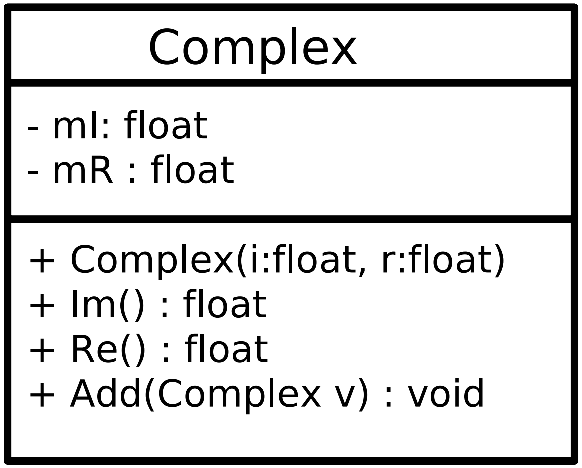
Access Modifiers

	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

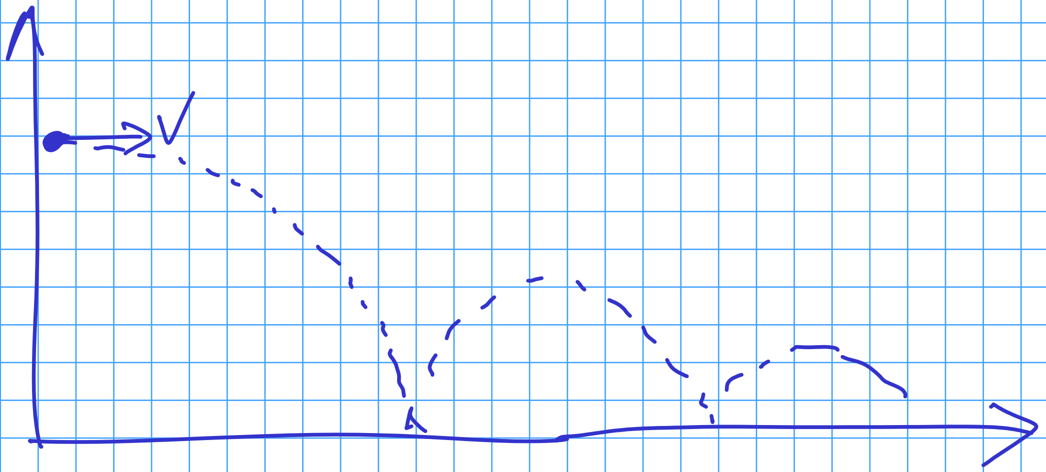
Complex Example

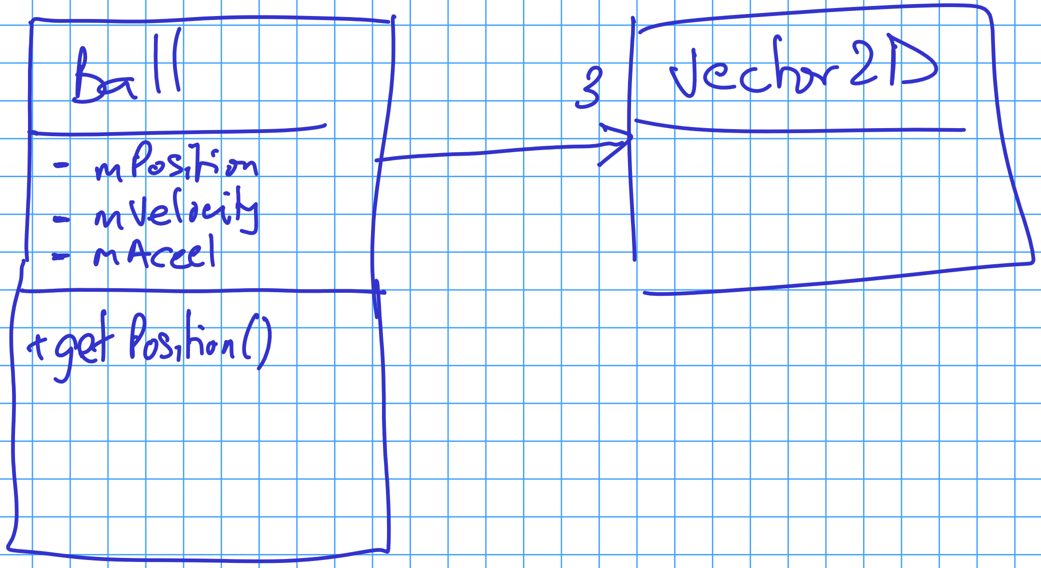


Complex

- mI: float
- mR : float

+ Complex(i:float, r:float)
+ Im() : float
+ Re() : float
+ Add(Complex v) : void


V


Ball
- mPosition
- mVelocity
- mAccel
+ getPosition()
3
Vector2D

Lecture 4:
Inheritance and Polymorphism

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

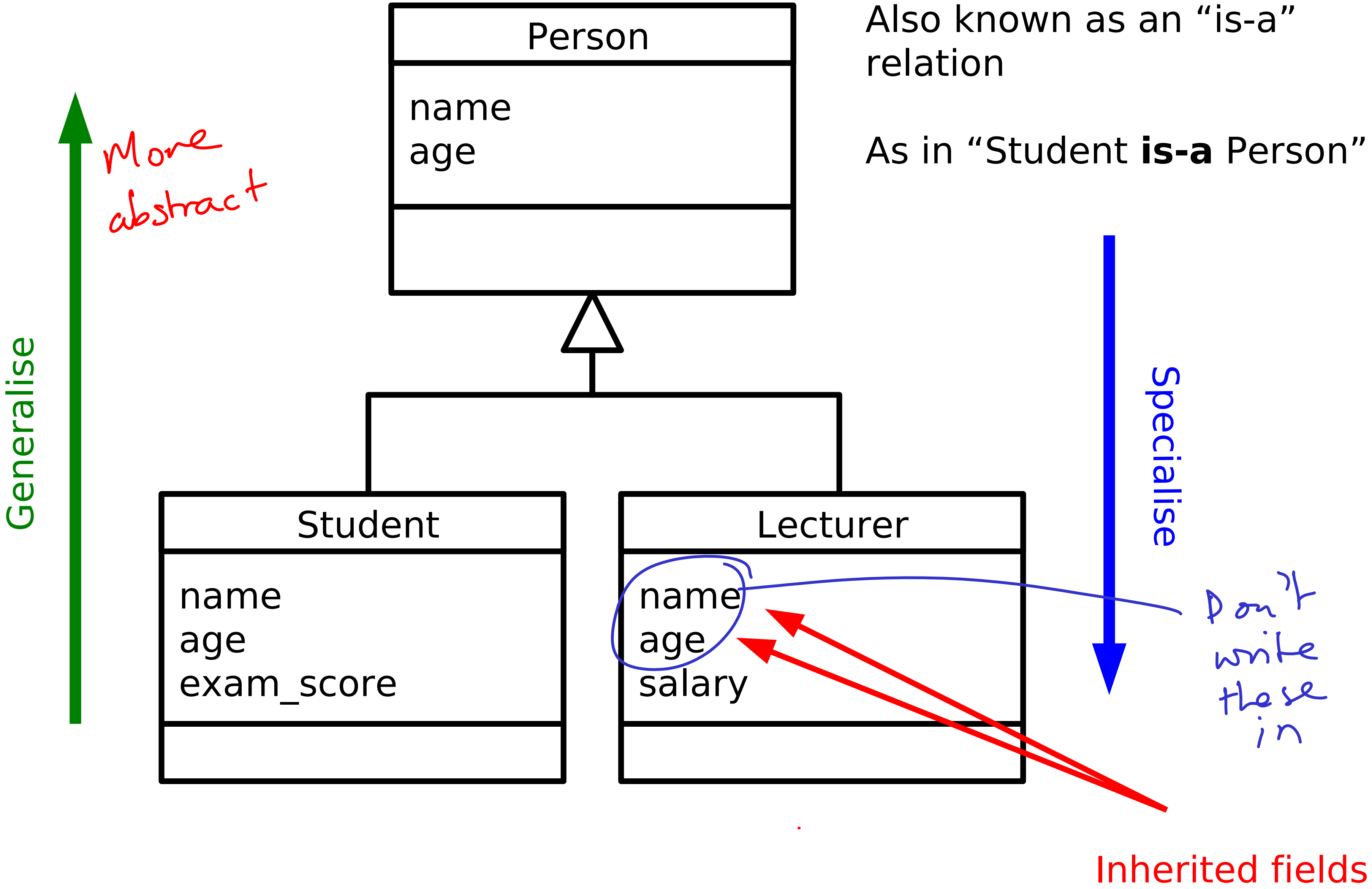
```
class Person {
   public int age;
   Public String name;
}

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”



Person

name
age

Student

name
age
exam_score

Lecturer

name
age
salary

Generalise

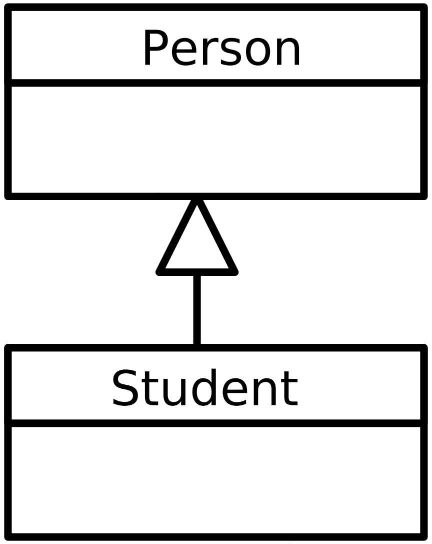
More abstract

Specialise

Don't write these in

Inherited fields

Widening Conversions



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

```
int x = 7
long y = (long)x;
```

```
Student s = new Student();

Person p = (Person) s;
```

✓

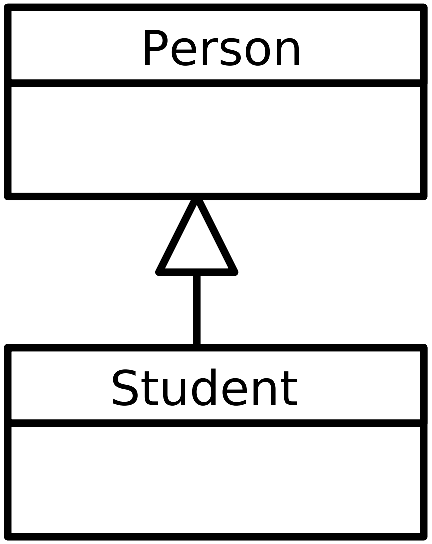
"Casting"

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

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

✓

Implicit cast

Narrowing Conversions



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

```
int x = 1000;
byte b = (byte) x;
```

```
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) ~~p;~~ s;
Students s2 = (Student) p;

OK because underlying object
Is a Student

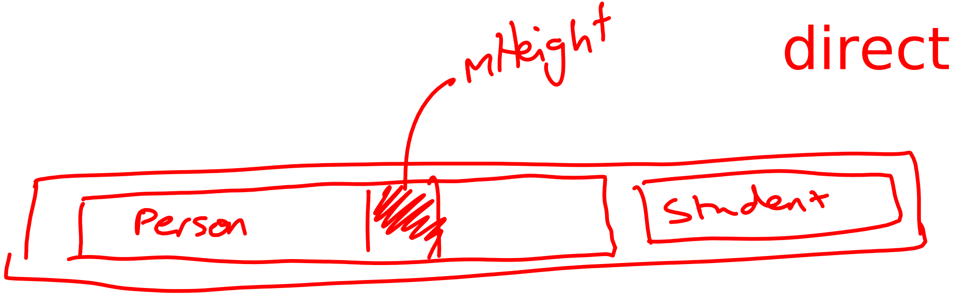
Fields and Inheritance

```
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; x
  }

}
```

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

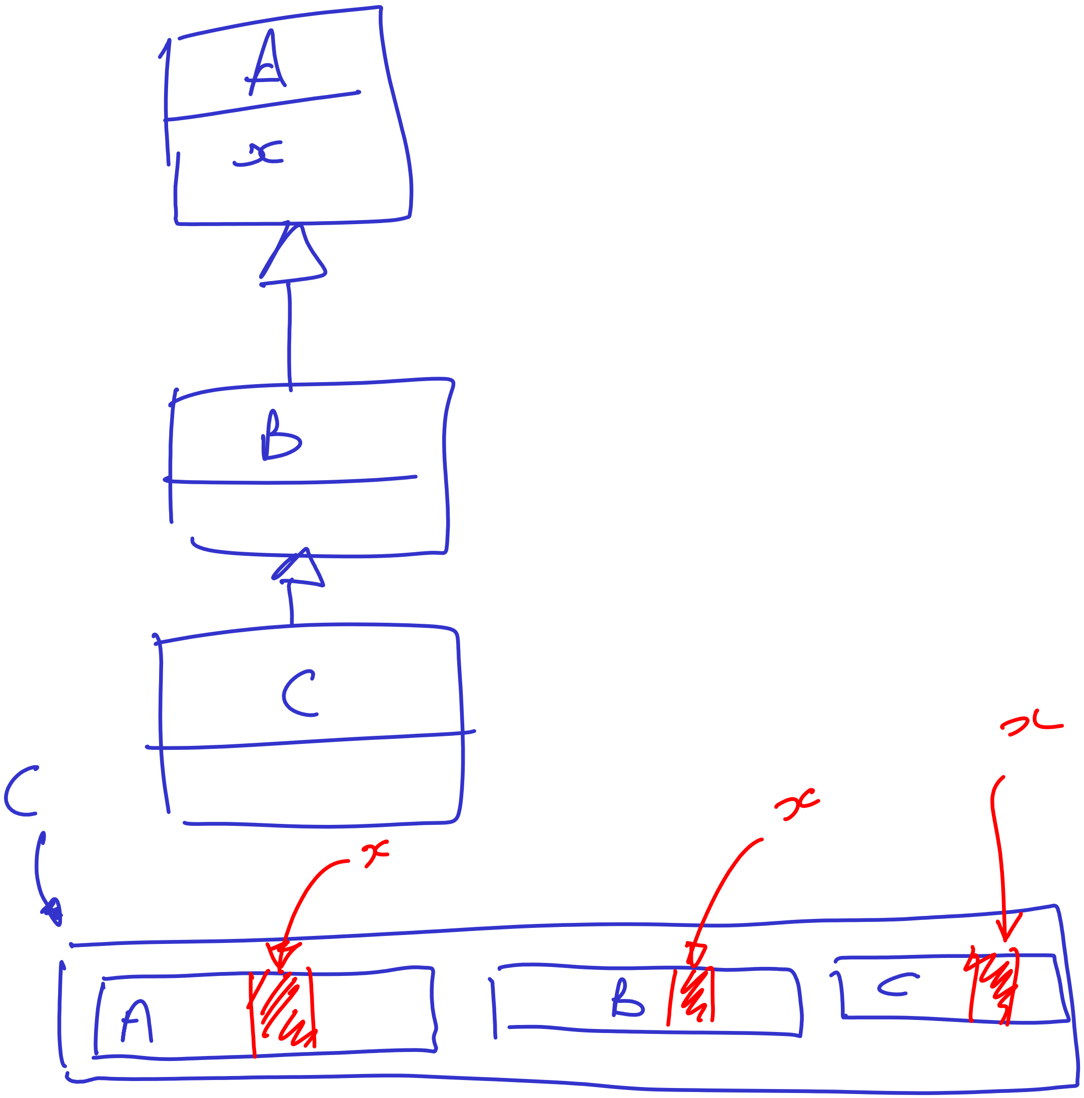
Student inherits this as a protected variable and so can access it

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



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

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

Polymorphic Concepts II

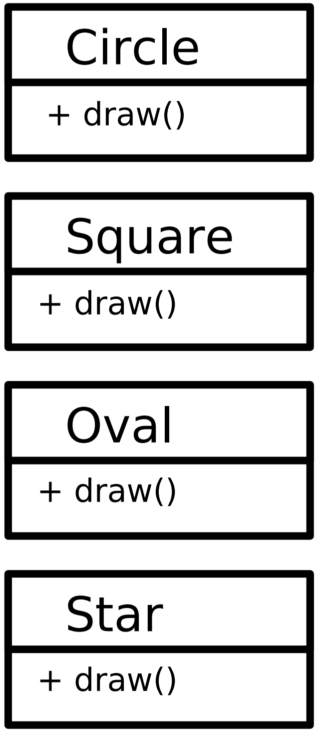
- **Dynamic** polymorphism
  - Run the method in the child
  - Must be done at run-time since that's when we know the child's type
  - Type errors cause run-time faults (crashes!)

```
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?



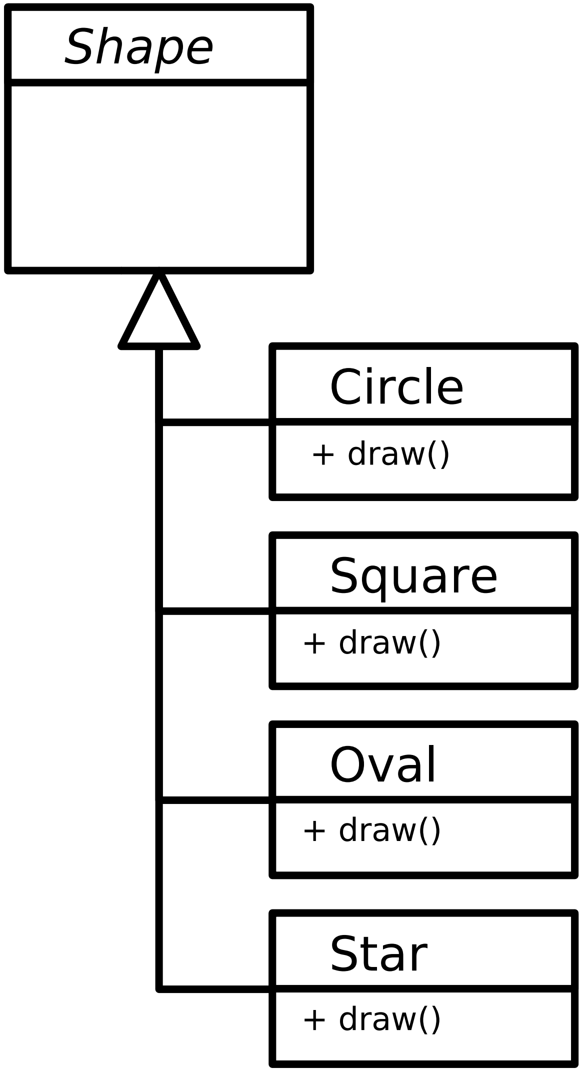
Circle
+ draw()

Square
+ draw()

Oval
+ draw()

Star
+ draw()

The Canonical Example II

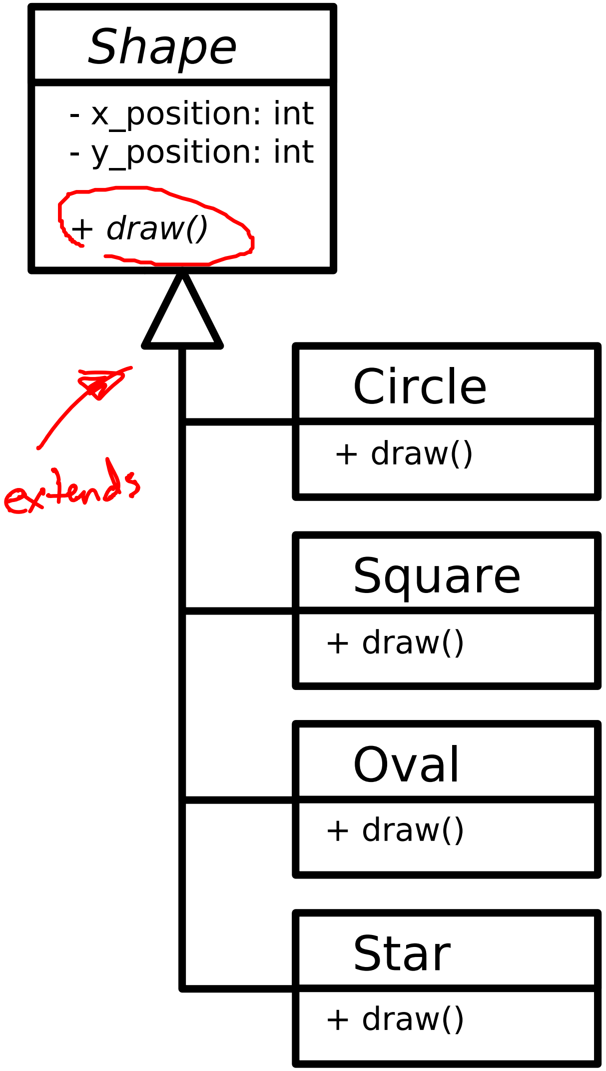


- **Option 2**
  - Keep a single list of Shape references
  - Figure out what each object really is, narrow the reference and then draw()

```
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...
```

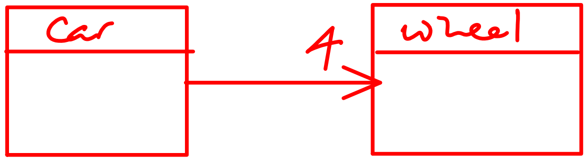
  - What if we want to add a new shape?

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

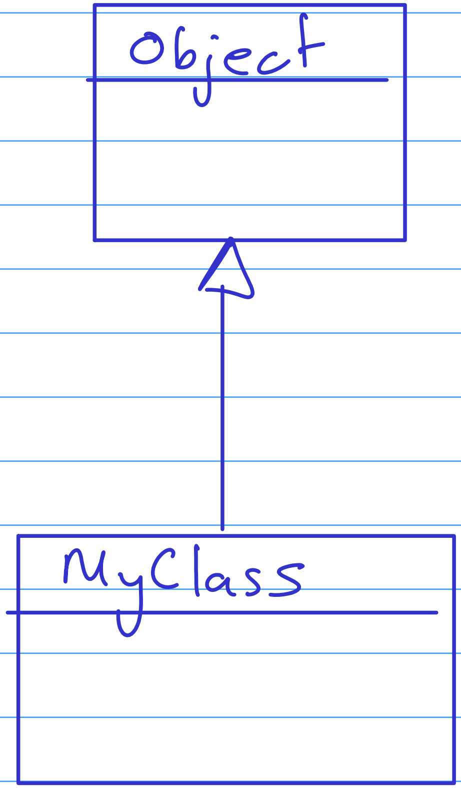
- Java
  - All methods are dynamic polymorphic.
- Python
  - All methods are dynamic polymorphic.
- C++
  - Only functions marked *virtual* are dynamic polymorphic

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

What polymorphism does not apply to

1. static methods
2. private methods
3. Variables (shadows)

[4. Generics]



Every class
extends Object

Lecture 5:
Static Data, Abstract Classes and Interfaces

Class-Level Data and Functionality I

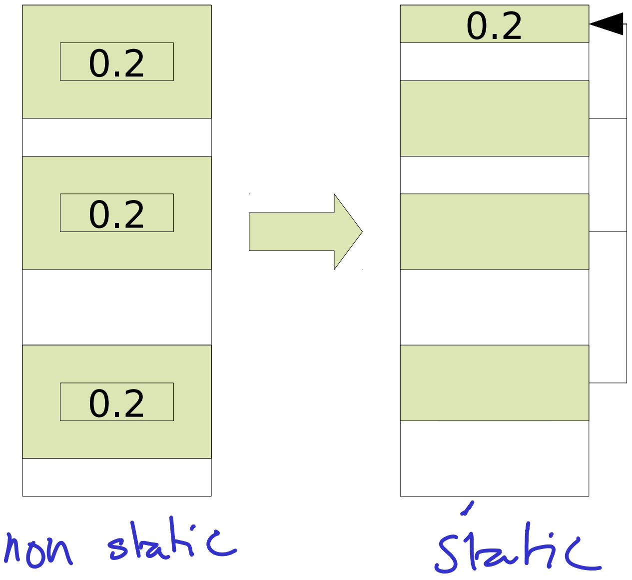
- A **static** field is created only once in the program's execution, despite being declared as part of a class

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

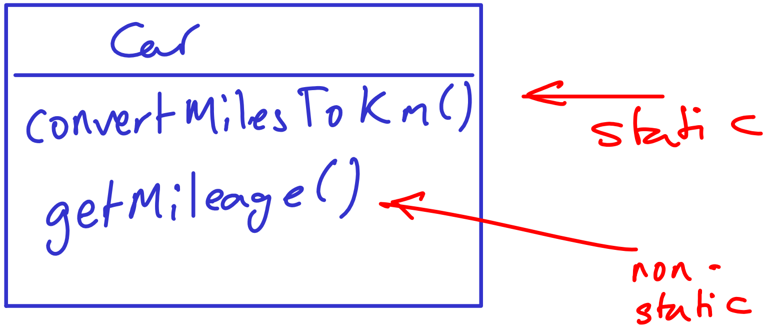
One of these created every time a new ShopItem is instantiated. Nothing keeps them all in sync.

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) {
     ...
  }
}
```

"True" functions
"pure" should
be static

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);
…
```

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();
   }
}
```

No def^{n}

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();
}
```

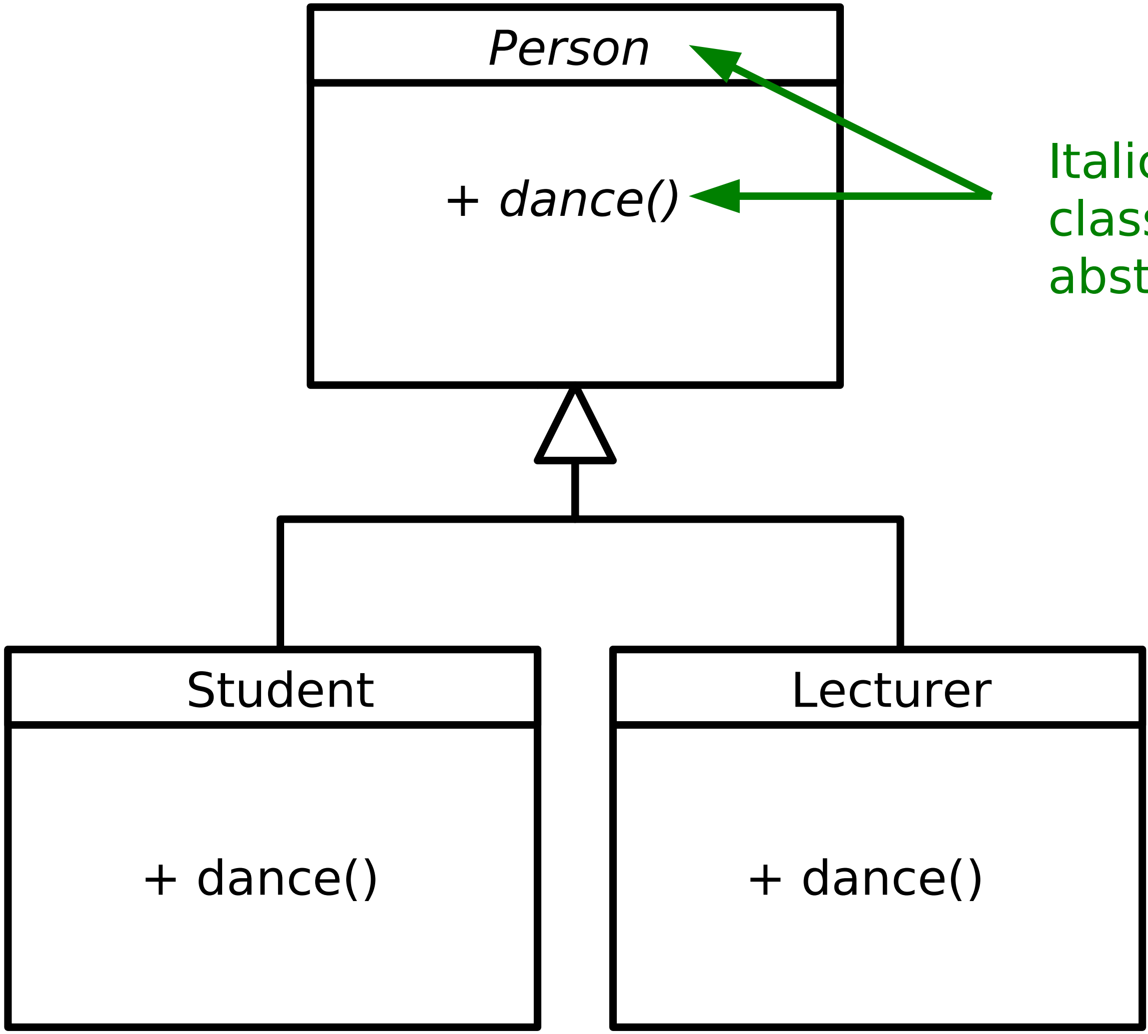
Java

```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



Person

\+ *dance()*

Student

\+ dance()

Lecturer

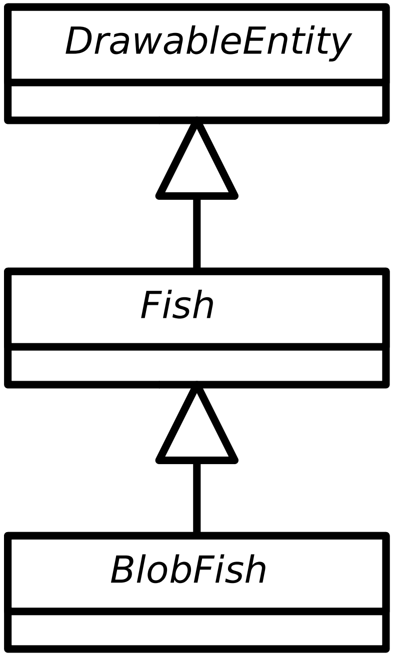
\+ dance()

Italics indicate the class or method is abstract

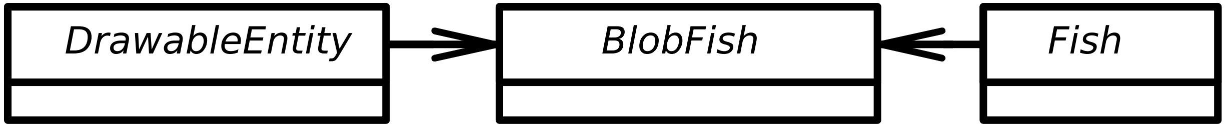
{ dance() }

Harder Problems

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

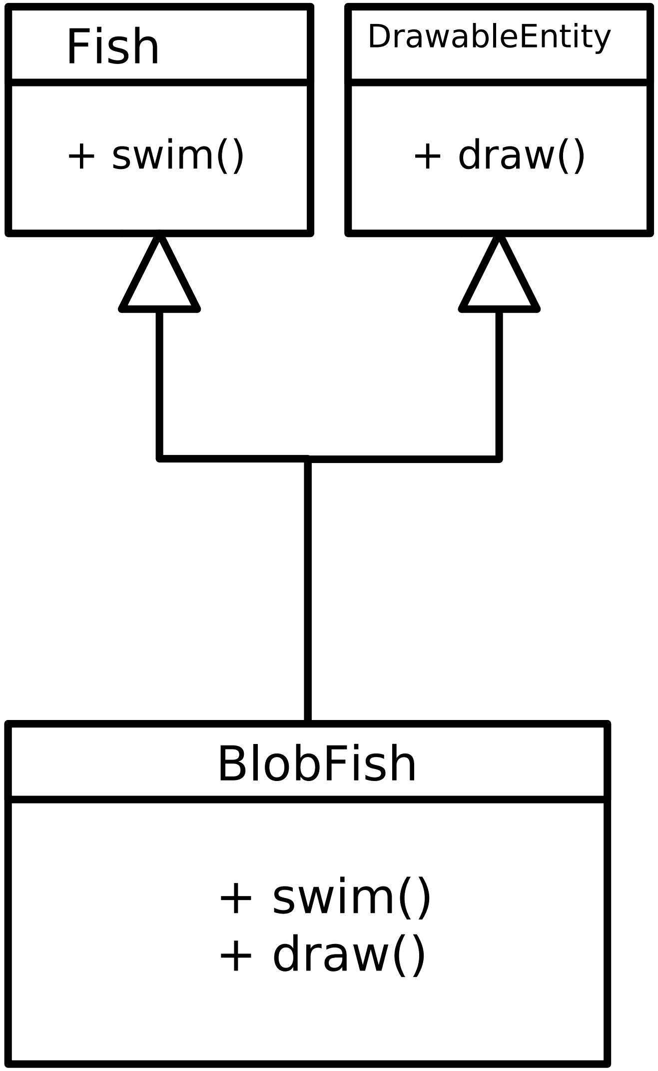


X Dependency between two independent concepts



X Conceptually wrong

Multiple Inheritance



Fish

+ swim()

DrawableEntity

+ draw()

BlobFish

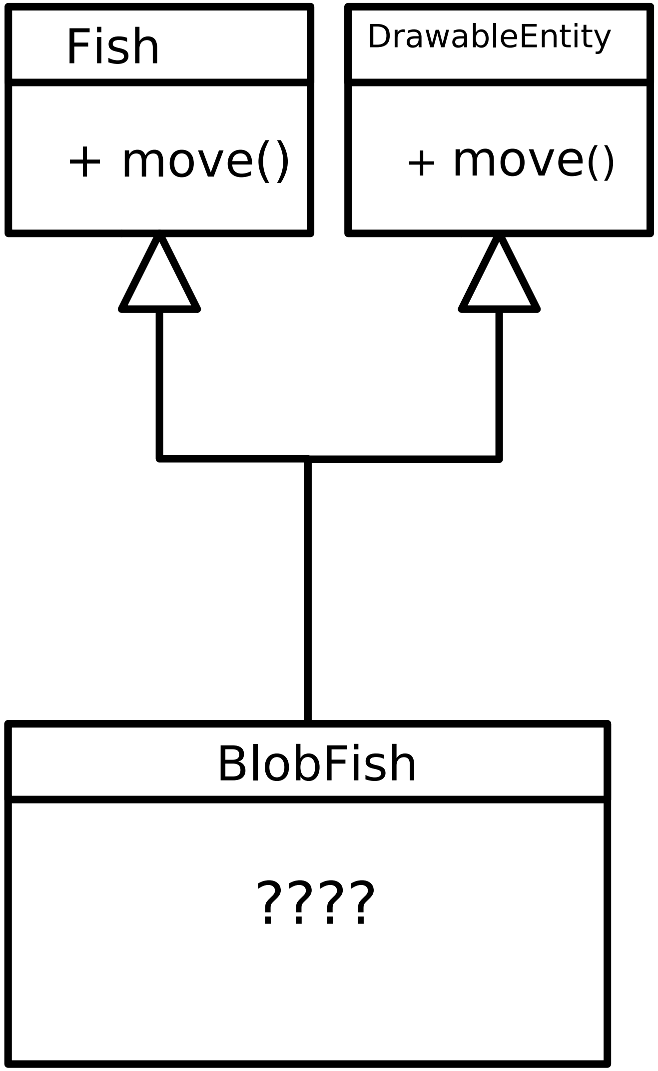
+ swim()
+ draw()

- 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…

Multiple Inheritance Problems

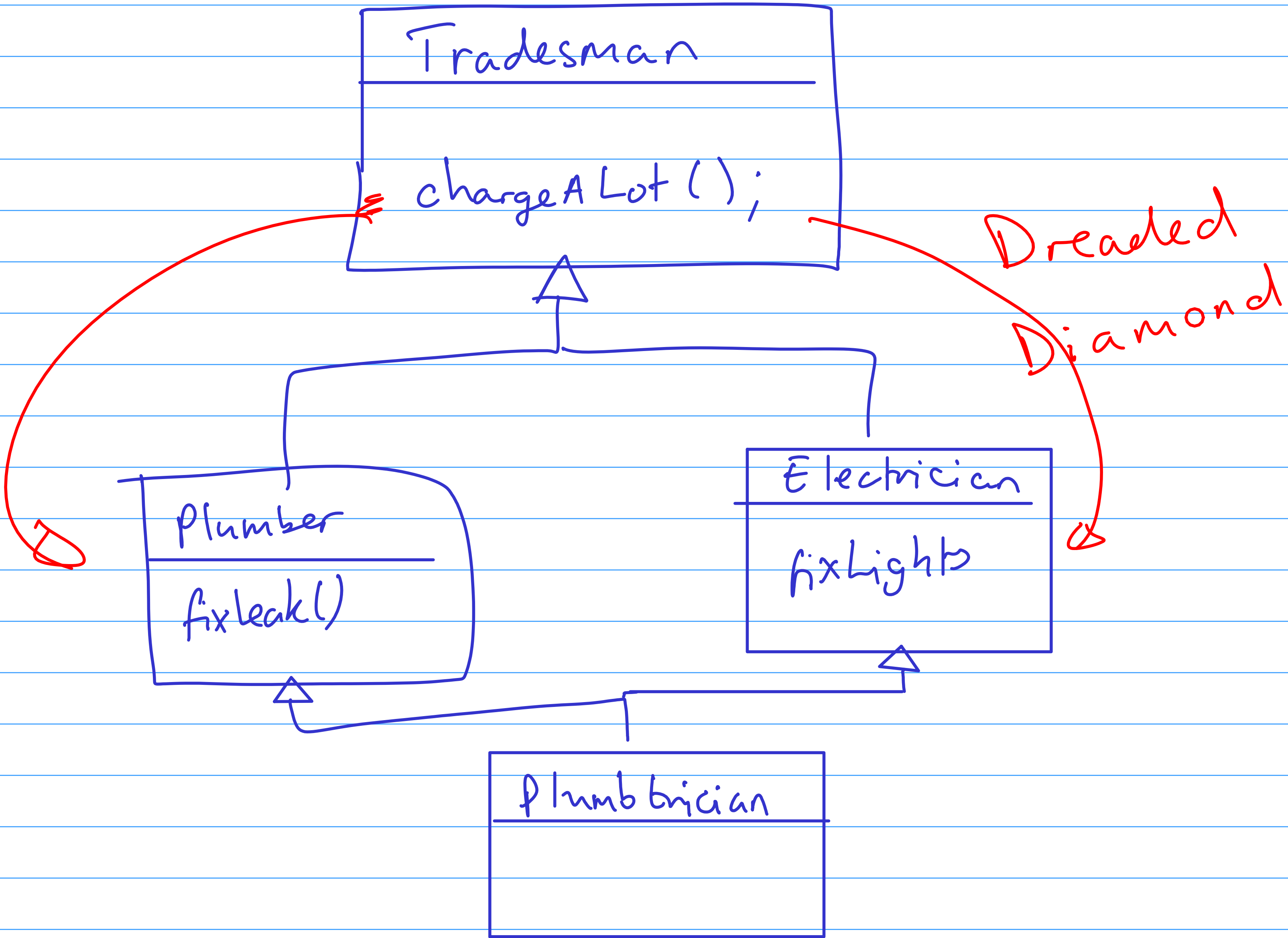


- What happens here? Which of the move() methods is inherited?
- Have to add some grammar to make it explicit
- C++:

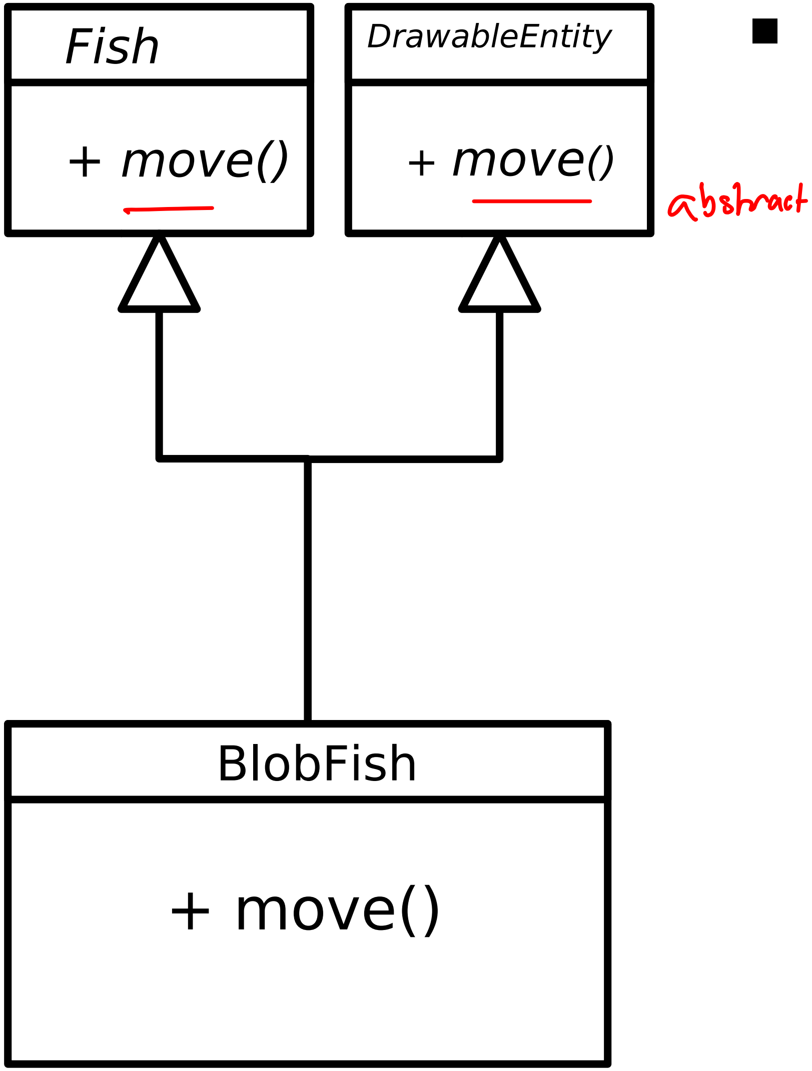
```
BlobFish *bf = new BlobFish();
bf->Fish::move();
bf->DrawableEntity::move();
```

- Yuk.

Tradesman Example

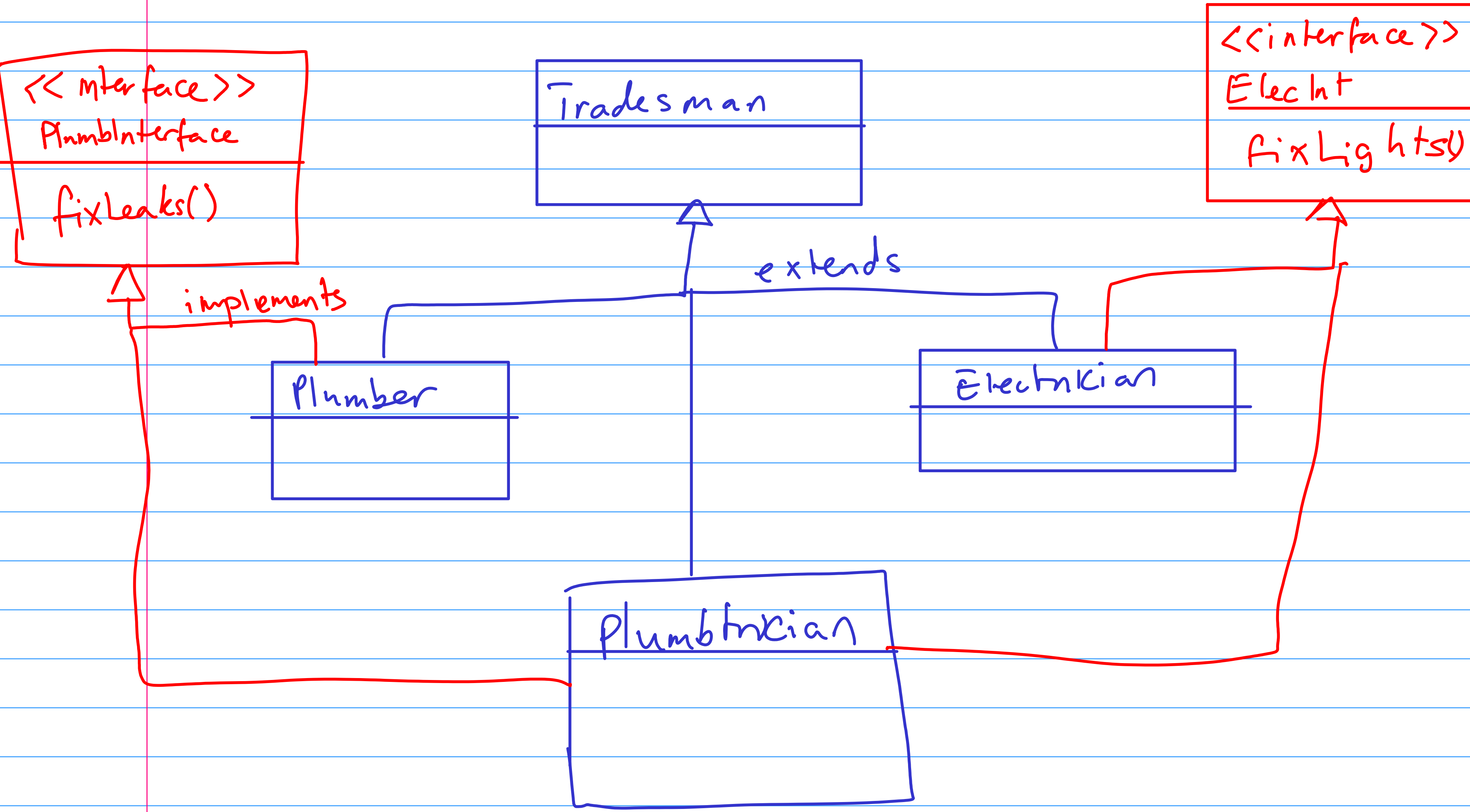


Fixing with Abstraction



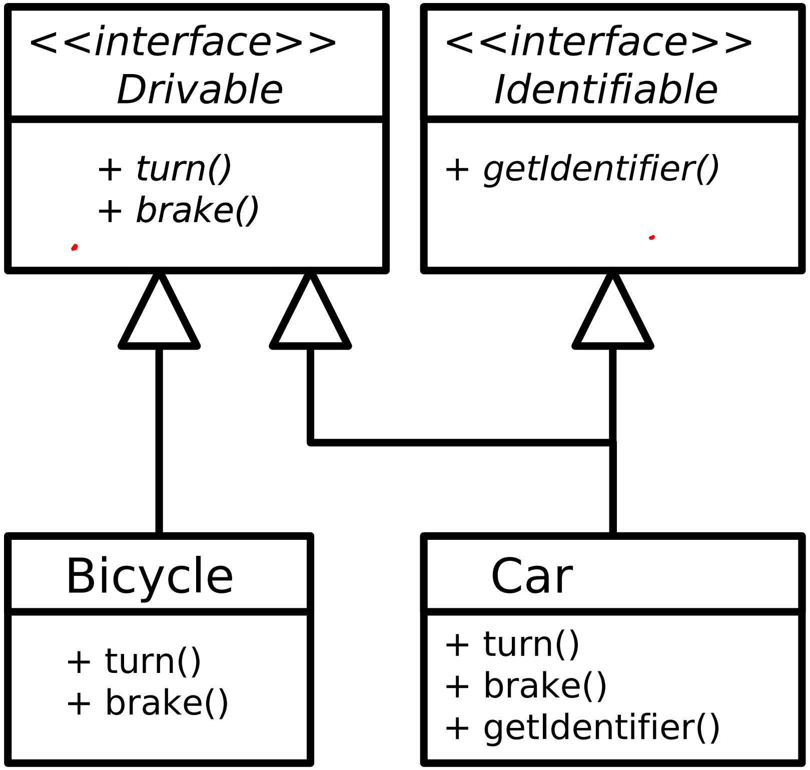
- Actually, this problem goes away if one or more of the conflicting methods is abstract

Tradesman Example with Interfaces



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() {…}
}
```

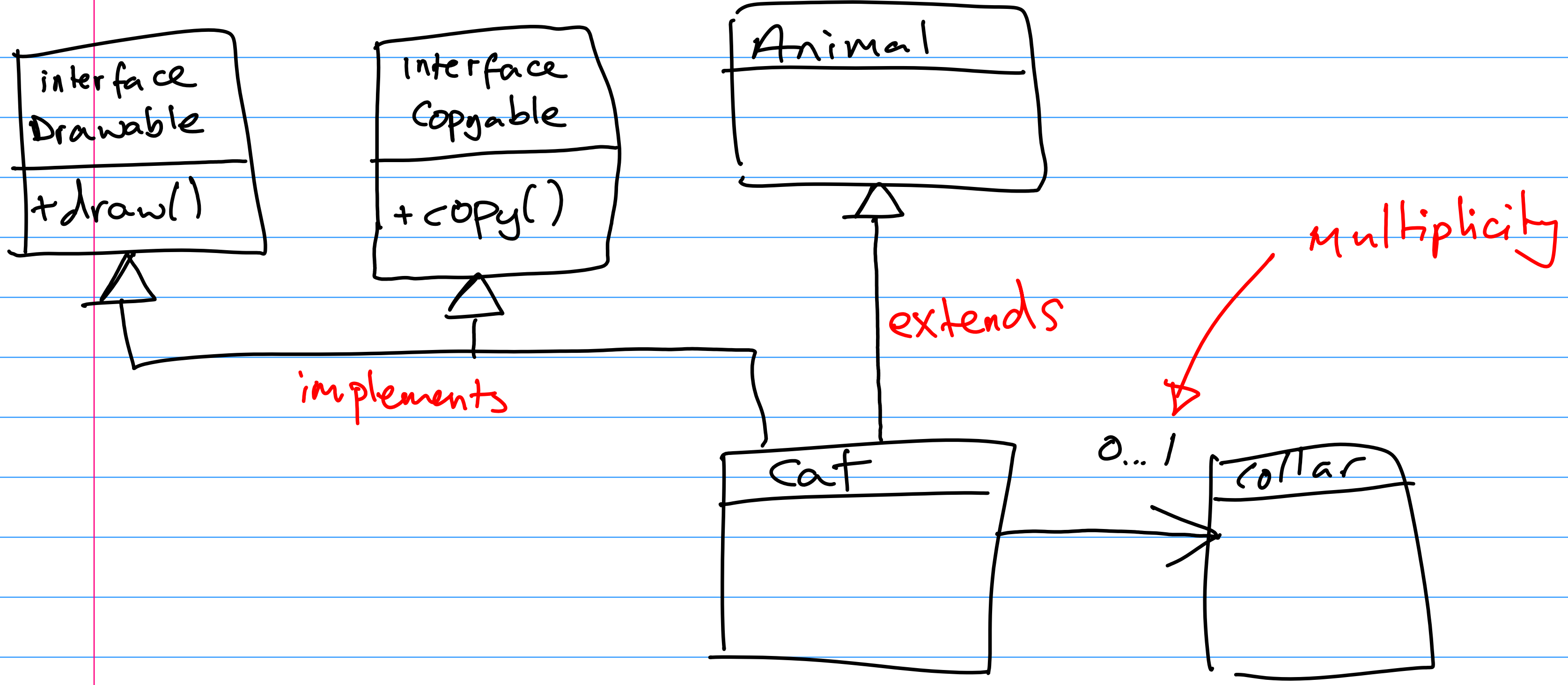
abstract

i/f

interface

extends Vehicle

Recap


interface Drawable
+draw()
interface Copyable
+copy()
Animal
implements
extends
multiplicity
Cat
0...1
collar

Final

state - only assign once

method - can't override

class - can't extend

Lecture 6:

Construction, Destruction and Error Handling

Constructors

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

type ref creates on heap Constructor

- 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

Constructor Examples

Java

```
public class Person {
   private String mName;

   // Constructor
   public Person(String name) {
       mName=name;
   }

   public static void main(
          String[] args) {
     Person p =
         new Person(“Bob”);
   }

}
```

No return type

C++

```
class Person {
   private:
      std::string mName;

   public:
      Person(std::string &name){
         mName=name;
      }
};


int main (int argc,
              char ** argv) {
   Person p (“Bob”);
}
```

Default Constructor

```
public Person() {
}
```

```
public class Person {
   private String mName;

   public static void main(String[] args) {
     Person p = new Person();
   }

}
```

- If you specify no constructor at all, Java fills in an empty one for you
- Here it creates Person() for us
- The default constructor takes no arguments (since it wouldn't know what to do with them!)

Multiple Constructors

```
public class Student {
    private String mName;
    private int mScore;

    public Student(String s) {
       mName=s;
       mScore=0;
    }

    public Student(String s, int sc) {
        mName=s;
        mScore=sc;
    }

    public static void main(String[] args) {
      Student s1 = new Student("Bob");
      Student s2 = new Student("Bob",55);
    }
  }
```

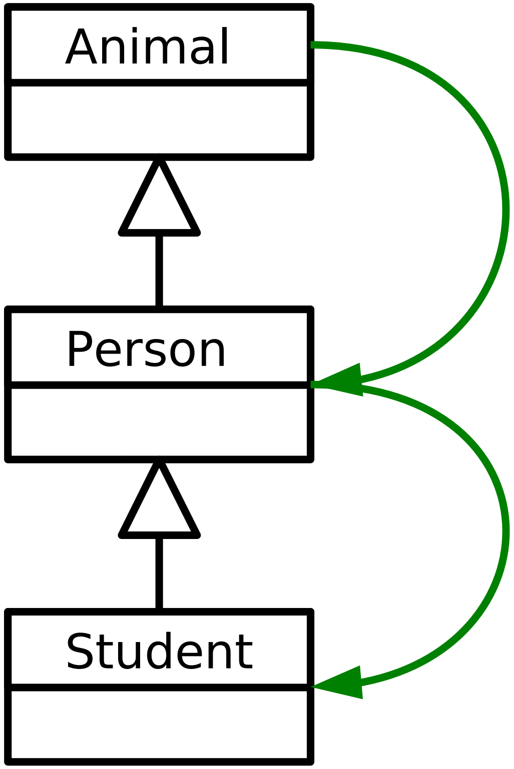
- You can specify as many constructors as you like.
- Each constructor must have a different signature (argument list)

method overloading

Constructor Chaining

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

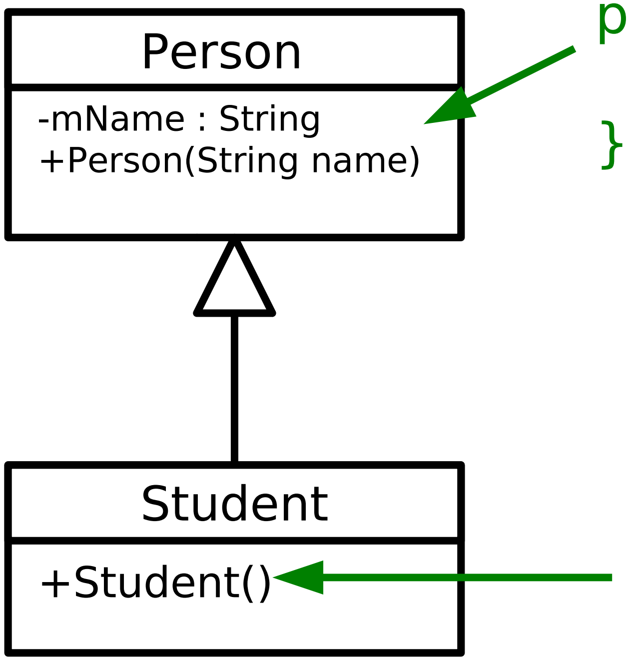
`Student s = new Student();`



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”);
}
```

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;

}
```

Cleaning Up

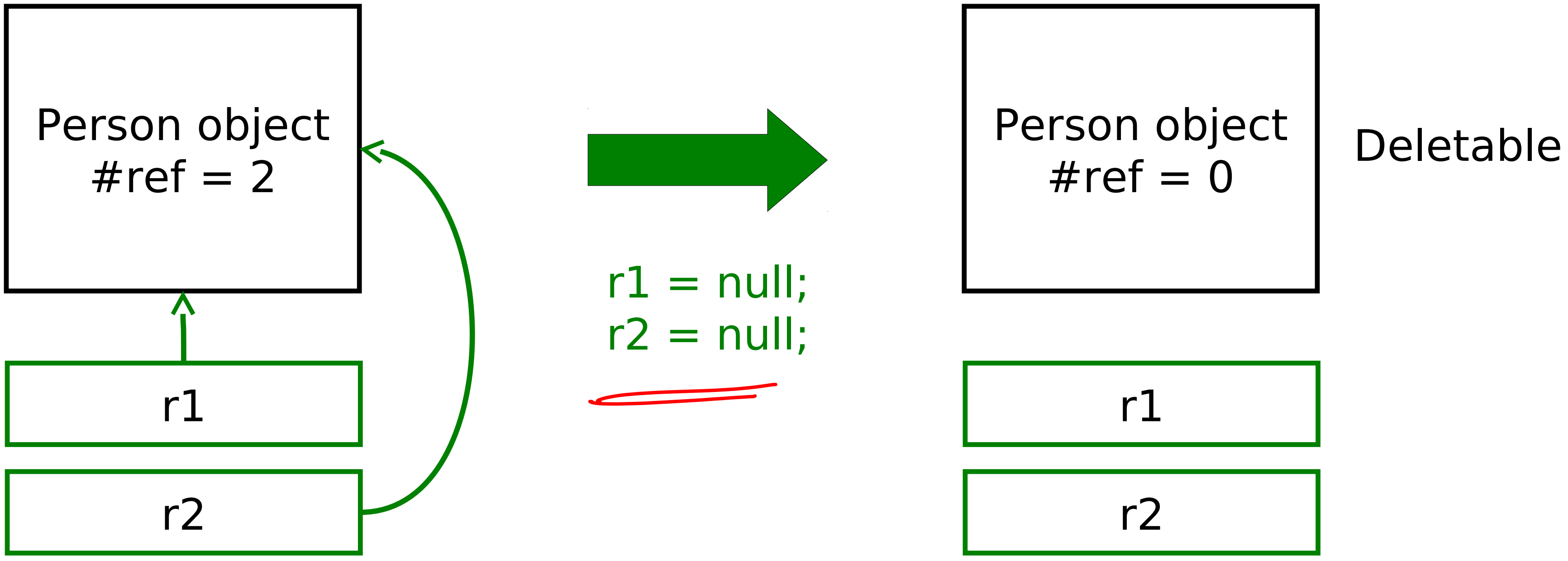
- A typical program creates lots of objects, not all of which need to stick around all the time

- **Approach 1:**
  - Allow the programmer to specify when objects should be deleted from memory
  - Lots of control, but what if they forget to delete an object?
    - A "memory leak"

- **Approach 2:**
  - Delete the objects automatically (**Garbage collection**)
  - But how do you know when an object will never be used again and can be deleted??

Cleaning Up (Java) I

- Java *reference counts*. i.e. 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



Person object
#ref = 2

r1

r2

r1 = null;
r2 = null;

Person object
#ref = 0

Deletable

r1

r2

Cleaning Up (Java) II

- Actual deletion occurs through a **garbage collector**
  - A separate process that periodically scans the objects in memory for any with a reference count of zero, which it then deletes.
  - 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
    - Gives noticeable pauses to your application while it runs.
    - But minimises memory leaks (it does not prevent them...)

Cleaning Up (Java) III

- One problem with GC is we have no idea *when* an object will actually be deleted. The GC may even decide to defer the deletion until a future run.
- This causes issues for destructors – it might be ages before a resource is closed and available again!
- Therefore Java doesn't have destructors
- It does have **finalizers** that gets run when the GC deletes an object
  - BUT there's no guarantee an object will ever get garbage collected in Java...
  - Garbage Collection != Destruction

Error Handling

- The traditional imperative way to handle errors is to return a value that indicates success/failure/error

```
public int divide(double a, double b) {
   if (b==0.0) return -1; // error
   double result = a/b;
   return 0; // success
}

...

if ( divide(x,y)<0) System.out.println(“Failure!!”);
```

- Problems:
  - Could ignore the return value
  - Have to keep checking what the return values are meant to signify, etc.
  - The actual result often can't be returned in the same way

Exceptions I

- An exception is an object that can be *thrown* or *raised* by a method when an error occurs and *caught* or *handled* by the calling code

```
public double divide(double a, double b)
                        throws DivideByZeroException {
   if (b==0) throw DivideByZeroException();
   else return a/b
}

...

try {
   double z = divide(x,y);
}
catch(DivideByZeroException d) {
   // Handle error here
}
```

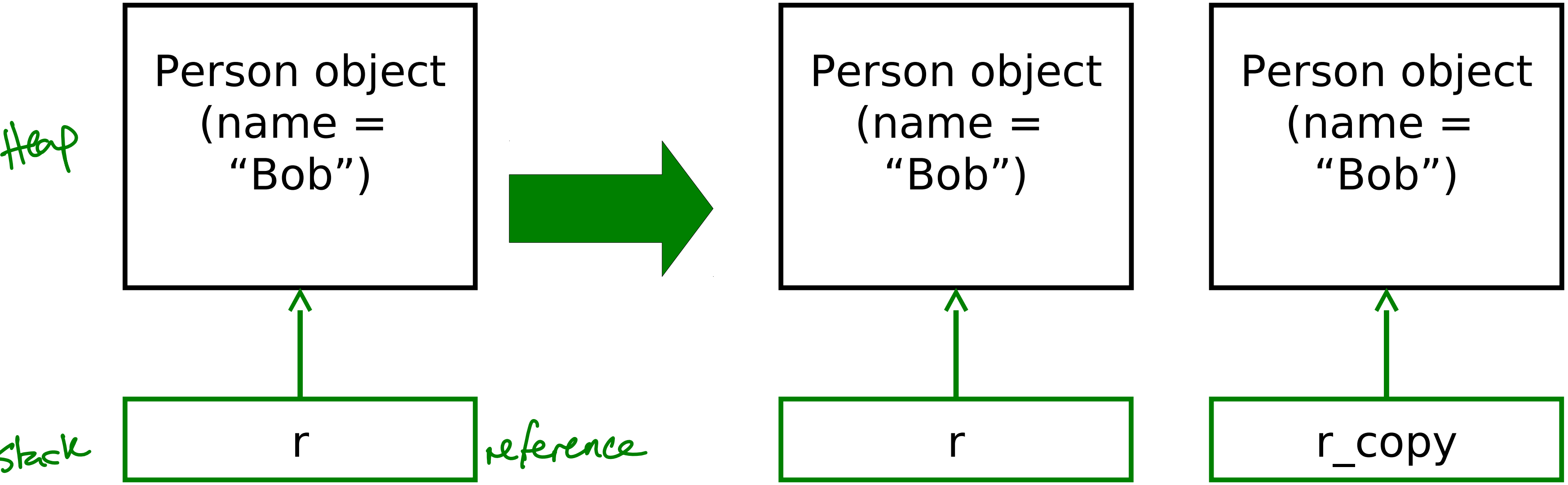
Exceptions II

- Advantages:
  - Class name can be descriptive (no need to look up error codes)
  - Doesn't interrupt the natural flow of the code by requiring constant tests
  - The exception object itself can contain state that gives lots of detail on the error that caused the exception
  - Can't be ignored, only handled

Lecture 7:
Copying and Cloning

Cloning I

- Sometimes we really do want to copy an object



Person object (name = “Bob”)

r

Person object (name = “Bob”)

r

Person object (name = “Bob”)

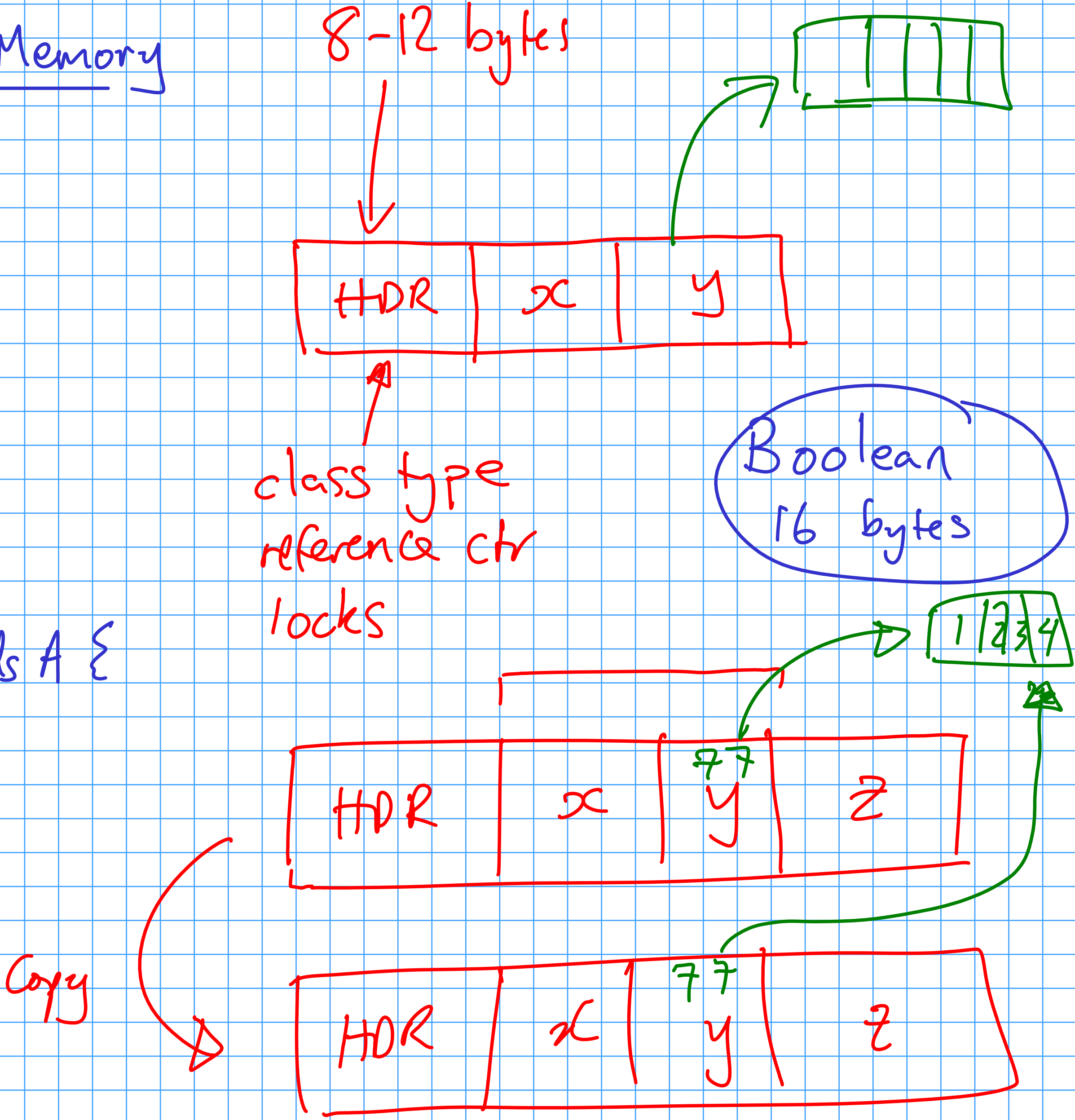
r_copy

- Java calls this ***cloning***
- We need special support for it

r_copy = r;

Java Objects in Memory

```
public class A {
    int x;
    int[] y;
}

public class B extends A {
    int z;

}
```

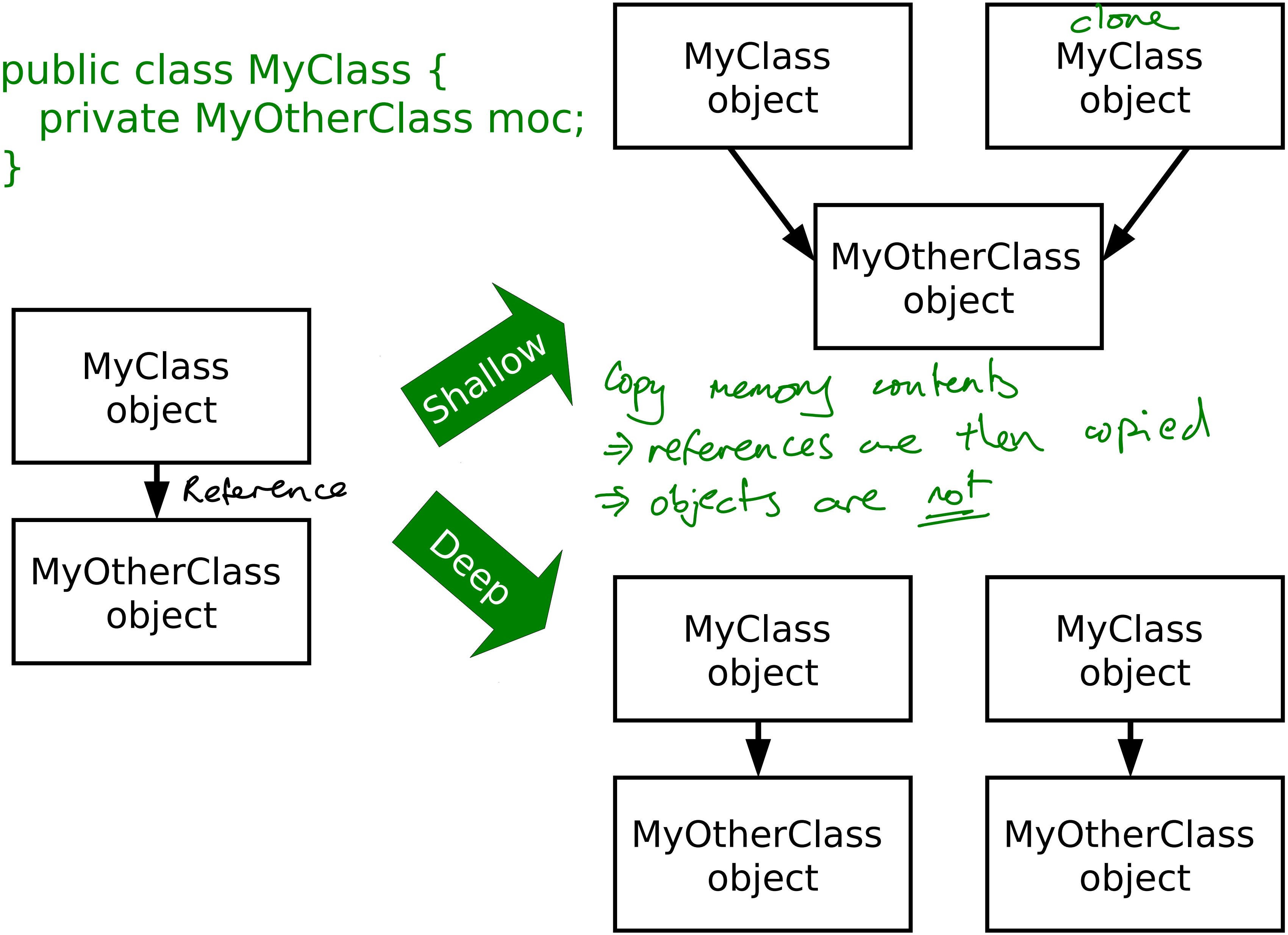


Cloning II

- Every class in Java ultimately inherits from the **Object** class
  - This class contains a clone() method so we just call this to clone an object, right?
  - This can go horribly wrong if our object contains reference types (objects, arrays, etc)

Shallow and Deep Copies

```
public class MyClass {
   private MyOtherClass moc;
}
```



MyClass object

Reference

MyOtherClass object

Shallow

MyClass object

clone

MyClass object

MyOtherClass object

Copy memory contents

⇒ references are then copied

⇒ objects are not

Deep

MyClass object

MyOtherClass object

MyClass object

MyOtherClass object

Java Cloning

- So do you want shallow or deep?
    - The default implementation of clone() performs a **shallow** copy
    - But Java developers were worried that this might not be appropriate: they decided they wanted to know for sure that we'd thought about whether this was appropriate

- Java has a **Cloneable** interface
    - If you call clone on anything that doesn't extend this interface, it fails

Recipe to clone

1. implement Cloneable
2. Make clone public [optional]
3. Start your clone with super.clone()
4. (Recursively/deep) clone any object in your class

Clone Example I

```
public class Velocity {
  public float vx;
  public float vy;
  public Velocity(float x, float y) {
      vx=x;
      vy=y;
  }
 };
```

evil but simple

```
 public class Vehicle {
  private int age;
  private Velocity vel;
  public Vehicle(int a, float vx, float vy) {
      age=a;
      vel = new Velocity(vx,vy);
  }
 };
```

primitive

reference type

Clone Example II

1

```
public class Vehicle implements Cloneable {
  private int age;
  private Velocity vel;
  public Vehicle(int a, float vx, float vy) {
     age=a;
     vel = new Velocity(vx,vy);
  }

  public Object clone() {
     return super.clone();
  }

};
```

2

throws cloneNotSupported Exception

Shallow copy

3

Clone Example III

```
public class Velocity implement Cloneable {
      ....
     public Object clone() {
         return super.clone();
     }
};
```

1

2

3

Shallow but OK.

```
public class Vehicle implements Cloneable {
  private int age;
  private Velocity v;
  public Student(int a, float vx, float vy) {
      age=a;
      vel = new Velocity(vx,vy);
  }

  public Object clone() {
      Vehicle cloned = (Vehicle) super.clone();
      cloned.vel = (Velocity)vel.clone();
      return cloned;
  }
};
```

shallow

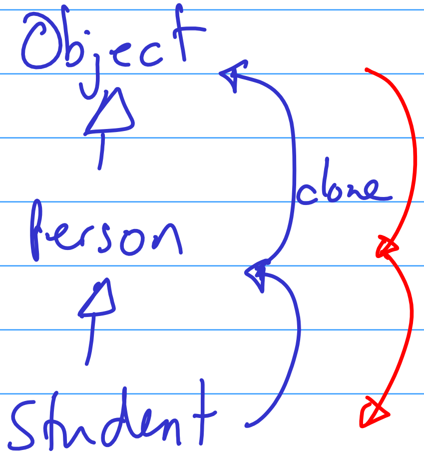
2

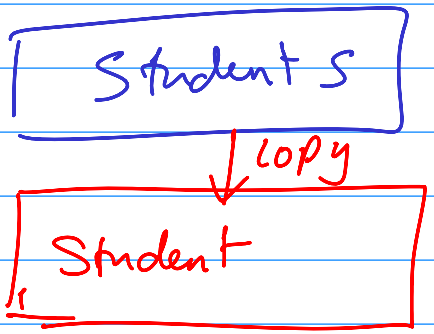
4 (deep)

Super.clone() Weirdness

Person

`Student cloned = (Student) super.clone()`





Marker Interfaces

- If you look at what's in the Cloneable interface, you'll find it's empty!! What's going on?
- Well, the clone() method is already inherited from Object so it doesn't need to specify it
- This is an example of a **Marker Interface**
  - A marker interface is an empty interface that is used to label classes
  - This approach is found occasionally in the Java libraries

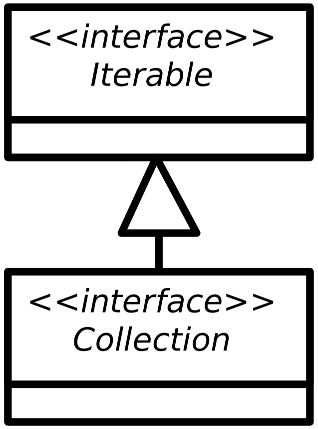
Lecture 8:
Collections and Generics

Java Class Library

- Java the platform contains around 4,000 classes/interfaces
    - Data Structures
    - Networking, Files
    - Graphical User Interfaces
    - Security and Encryption
    - Image Processing
    - Multimedia authoring/playback
    - And more…

- All neatly(ish) arranged into packages (see API docs)

Java's Collections Framework

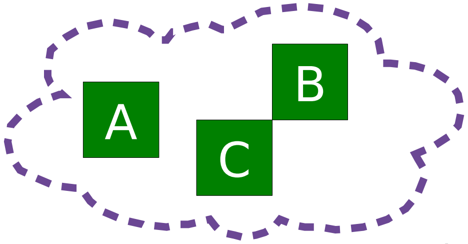


- Important chunk of the class library
- A collection is some sort of grouping of things (objects)
- Usually when we have some grouping we want to go through it (“***iterate*** over it”)

- The Collections framework has two main interfaces: Iterable and Collections. They define a set of operations that all classes in the Collections framework support
- add(Object o), clear(), isEmpty(), etc.

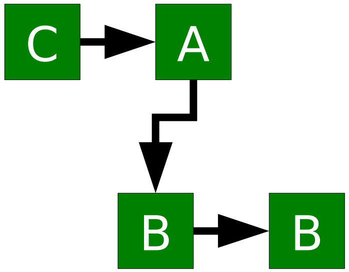
Major Collections Interfaces I

- <<interface>> Set
  - Like a mathematical set in DM 1
  - A collection of elements with no duplicates
  - Various concrete classes like TreeSet (which keeps the set elements sorted)

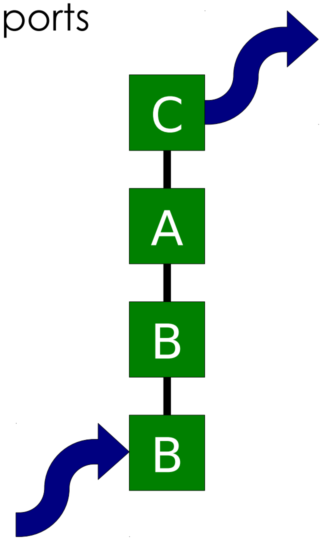


Sorted

- <<interface>> List
  - An ordered collection of elements that may contain duplicates
  - ArrayList, Vector, LinkedList, etc.



- <<interface>> Queue
  - An ordered collection of elements that may contain duplicates and supports removal of elements from the head of the queue
  - PriorityQueue, LinkedLIst, etc.

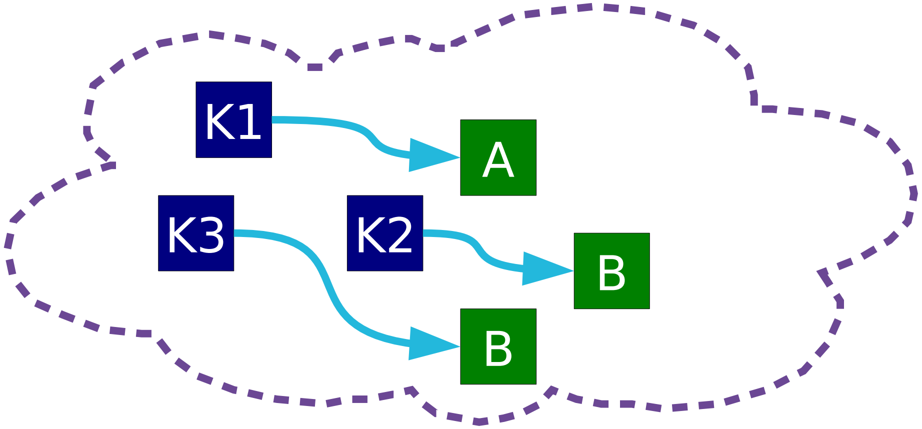


Prefix Tree ⇒ Sorted

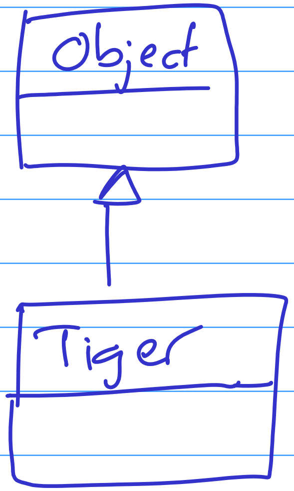
Hash ⇒ v. fast access (but ~random order)

Major Collections Interfaces II

- <<interface>> Map
  - Like relations in DM 1, or dictionaries in ML
  - Maps key objects to value objects
  - Keys must be unique
  - Values can be duplicated and (sometimes) null.



OOP for Collections



Write for an "Object"

⇒ Work for all else

Iteration

- for loop

```
LinkedList list = new LinkedList();
...
for (int i=0; i<list.size(); i++) {
    Object next = list.get(i);
}
```

- foreach loop (Java 5.0+)

```
LinkedList list = new LinkedList();
...
for (Object o : list) {

}
```

Iterators

- What if our loop changes the structure?

```
for (int i=0; i<list.size(); i++) {
    If (i==3) list.remove(i);
}
```

- Java introduced the Iterator class

```
Iterator it = list.iterator();

while(it.hasNext()) {Object o = it.next();}

for (; it.hasNext(); ) {Object o = it.next();}
```

- Safe to modify structure

```
while(it.hasNext()) {
    it.remove();
}
```

Collections and Types I

```
// Make a TreeSet object
TreeSet ts = new TreeSet();

// Add integers to it
ts.add(new Integer(3));

// Loop through
iterator it = ts.iterator();
while(it.hasNext()) {
    Object o = it.next();
    Integer i = (Integer)o;
}
```

- The original Collections framework just dealt with collections of Objects
  - Everything in Java “is-a” Object so that way our collections framework will apply to any class
  - But this leads to:
    - Constant casting of the result (ugly)
    - The need to know what the return type is
    - Accidental mixing of types in the collection

Collections and Types II

```
// Make a TreeSet object
TreeSet ts = new TreeSet();

// Add integers to it
ts.add(new Integer(3));
ts.add(new Person(“Bob”));

// Loop through
iterator it = ts.iterator();
while(it.hasNext()) {
    Object o = it.next();
    Integer i = (Integer)o;
}
```

Going to fail for the second element! (But it will compile: the error will be at runtime)

Java Generics

- To help solve this sort of problem, Java introduced *Generics* in JDK 1.5
- Basically, this allows us to tell the compiler what is supposed to go in the Collection
- So it can generate an error at compile-time, not run-time

```
// Make a TreeSet of Integers
TreeSet<Integer> ts = new TreeSet<Integer>();

// Add integers to it
ts.add(new Integer(3));
ts.add(new Person(“Bob”));

// Loop through
Iterator<Integer> it = ts.iterator();
while(it.hasNext()) {
     Integer i = it.next();
}
```

Won't even compile

No need to cast :-)

Generics Declaration and Use

```
public class Coordinate <T> {
    private T mX;
    private T mY;

    public Coordinate(T x, T y) {
        mX=x; mY=y;
    }

    public T getX() { return mX; }
    public T getY() { return mY; }
}

Coordinate<Double> c =
    New Coordinate<Double>(1.0,1.0);

Double d = c.getX();
```

Placeholder

"Parameter type"

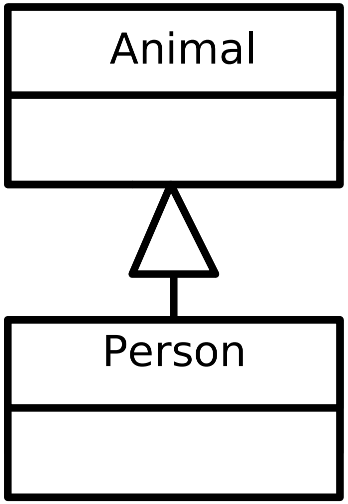
Generics Implementation : Type Erasure

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

↓ Type erasure (Remembers l ⇒ Integers)

```
LinkedList l = new LinkedList();
```

Generics and SubTyping



```
// Object casting
Person p = new Person();
Animal o = (Animal) p;

// List casting
List<Person> plist = new LinkedList<Person>();
List<Animal> alist = (List<Animal>)plist;
```

So a list of **Persons** is a list of **Animals**, yes?

```
alist.add(new Giraffe());
```

⇒ Can't be allowed

- Can't subtype
- Can't create arrays in a generic type

Lecture 9:
Comparing Objects

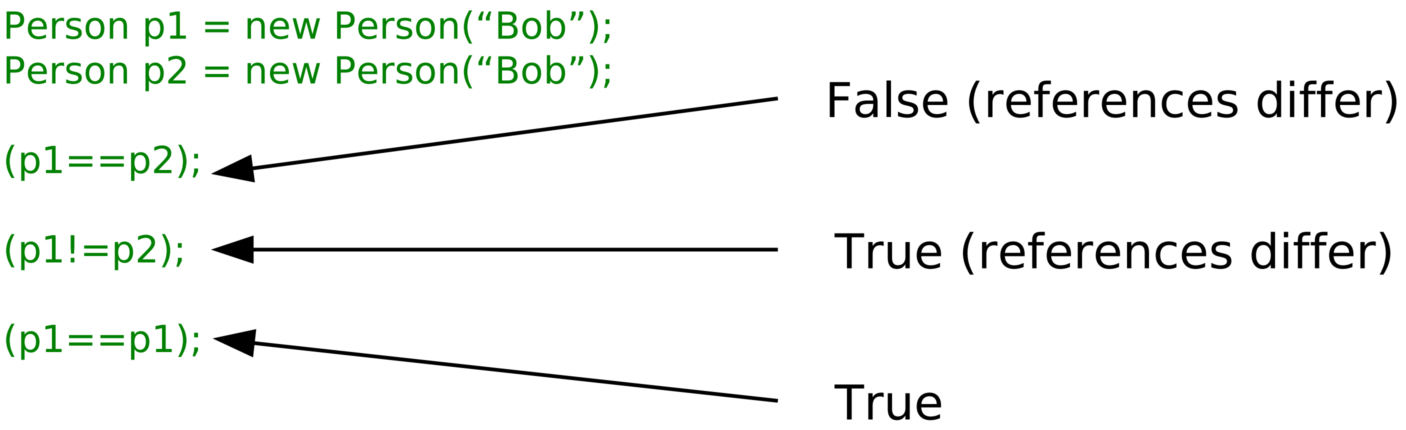
Comparing Primitives

>	Greater Than
>=	Greater than or equal to
==	Equal to
!=	Not equal to
<	Less than
<=	Less than or equal to

- Clearly compare the value of a primitive
- But what does (ref1==ref2) do??
  - Test whether they point to the same object?
  - Test whether the objects they point to have the same state?

Option 1: a==b, a!=b

- These compare the *references directly*



```
Person p1 = new Person(“Bob”);
Person p2 = new Person(“Bob”);

(p1==p2);

(p1!=p2);

(p1==p1);
```

False (references differ)

True (references differ)

True

Literally comparing memory addresses

Option 2: The equals() Method

- Object defines an equals() method. By default, this method just does the same as ==.
    - Returns boolean, so can only test equality
    - Override it if you want it to do something different
    - Most (all?) of the core Java classes have properly implemented equals() methods

```
public EqualsTest {
    public int x = 8;

    public boolean equals(Object o) {
        EqualsTest e = (EqualsTest)o;
        return (this.x==e.x);
    }

    public static void main(String args[]) {
        EqualsTest t1 = new EqualsTest();
        EqualsTest t2 = new EqualsTest();
        System.out.println(t1==t2);
        System.out.println(t1.equals(t2));
    }
}
```

Option 3: Comparable<T> Interface I

```
int compareTo(T obj);
```

- Part of the Collections Framework
- Doesn't just tell us true or false, but smaller, same, or larger: useful for sorting.
- Returns an integer, r:

r<0	This object is less than obj
r==0	This object is equal to obj
r>0	This object is greater than obj

Giving a "natural ordering"

Option 3: Comparable<T> Interface II

```
public class Point  implements Comparable<Point> {
    private final int mX;
    private final int mY;
    public Point (int, int y) { mX=x; mY=y; }

    // sort by y, then x
    public int compareTo(Point p) {
        if ( mY>p.mY) return 1;
        else if (mY<p.mY) return -1;
        else {
            if (mX>p.mX) return 1;
            else if (mX<p.mX) return -1;
            else return 0.
        }
    }
}
```

requires

simple concept

```
// This will be sorted automatically by y, then x
Set<Point> list = new TreeSet<Point>();
```

use

Option 4: Comparator<T> Interface

int compare~~To~~(T obj1, T obj2)

- Also part of the Collections framework and allows us to specify a particular comparator for a particular job
- E.g. a Person might have a compareTo() method that sorts by surname. We might wish to create a class AgeComparator that sorts Person objects by age. We could then feed that to a Collections object.

Lecture 10:
Design Patterns

Design Patterns

- A Design Pattern is a general reusable solution to a commonly occurring problem in software design
- Coined by Erich Gamma in his 1991 Ph.D. thesis
- Originally 23 patterns, now many more. Useful to look at because they illustrate some of the power of OOP (and also some of the pitfalls)
- We will only consider a subset

The Open-Closed Principle

Classes should be open for extension but closed for modification

Extend (inherit) to add functionality

Not modify an established class

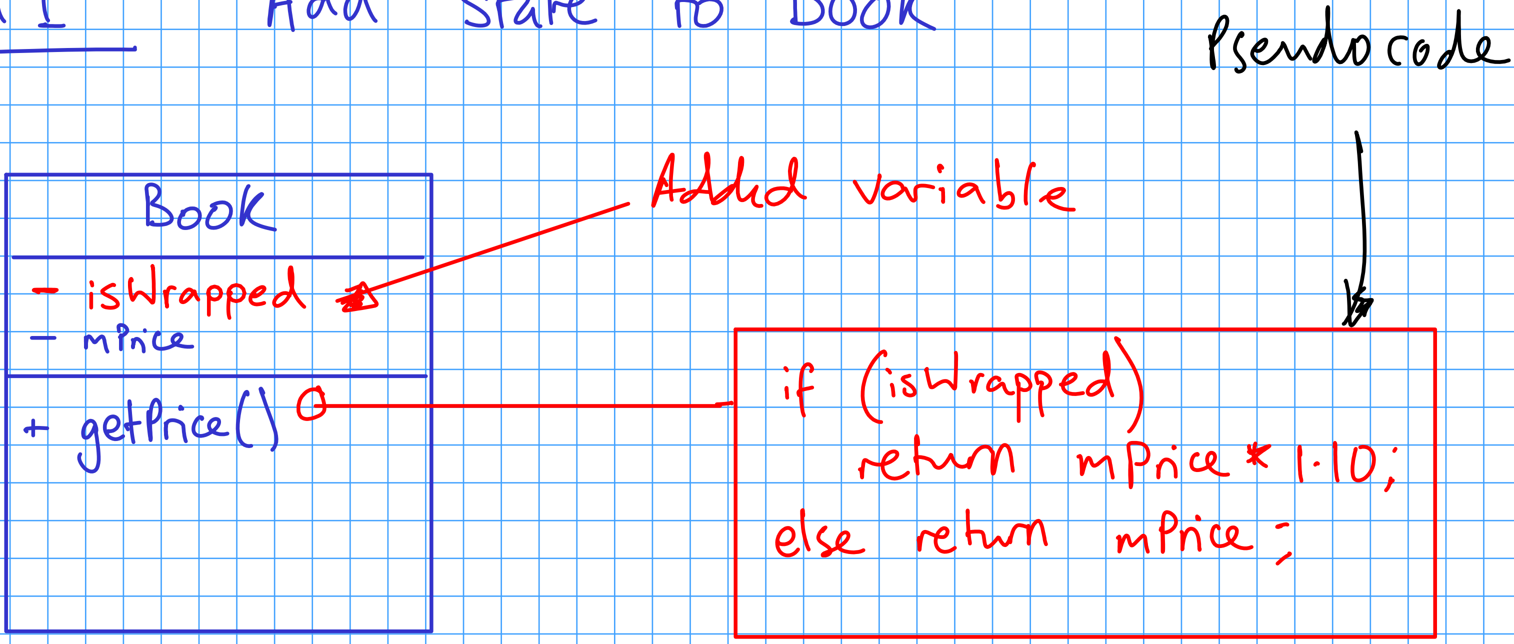
- i.e. we would like to be able to modify the behaviour without touching its source code
- This rule-of-thumb leads to more reliable large software and will help us to evaluate the various design patterns

Decorator

Abstract problem: How can we add state or methods at runtime?

Example problem: How can we efficiently support gift-wrapped books in an online bookstore?

Solution 1 Add state to Book



```
if (isWrapped)
    return mPrice * 1.10;
else return mPrice;
```

✓ works

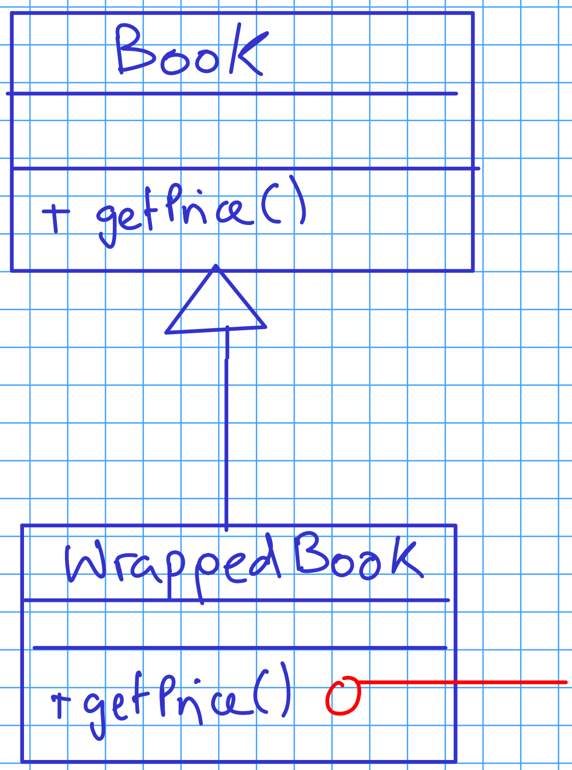
✗ Clutter Book class

→ waste storage

→ Harder to maintain

(violates open-closed principle)

Solution 2 Extend Book

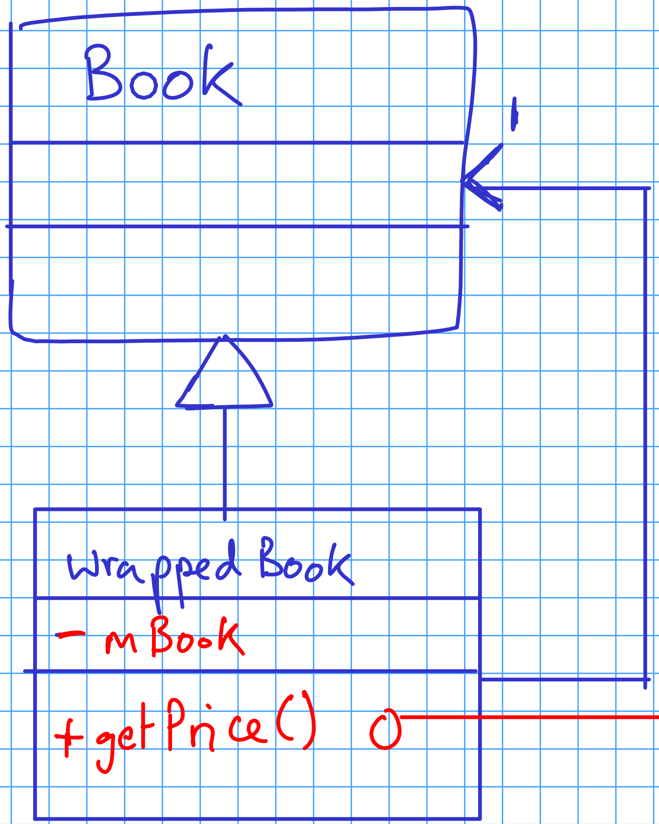


- ✓ More storage efficient
- ✗ Double no. of classes
- ✗ Remember to create wrapped version!
- ✗ Can't unwrap

```
return
 super.getPrice() * 1.10;
```

10% surcharge

Solution 3 Decorator pattern



return mBook.getPrice() * 1.10

✓ Everything wrappable

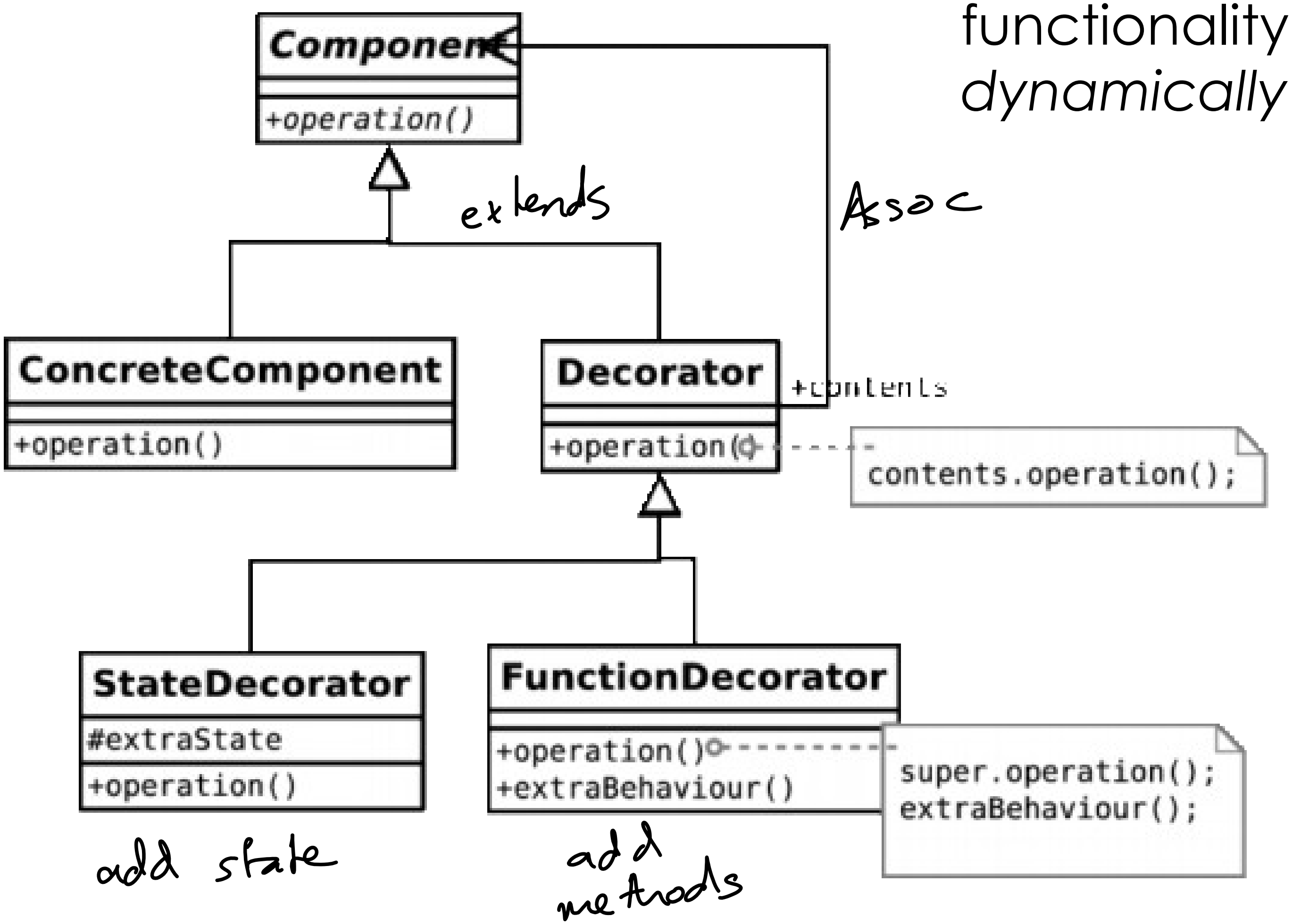
✓ Easy conversion

× Wrap a wrapped Book

Association AND inheritance.

× Long chains of small objects

Decorator in General



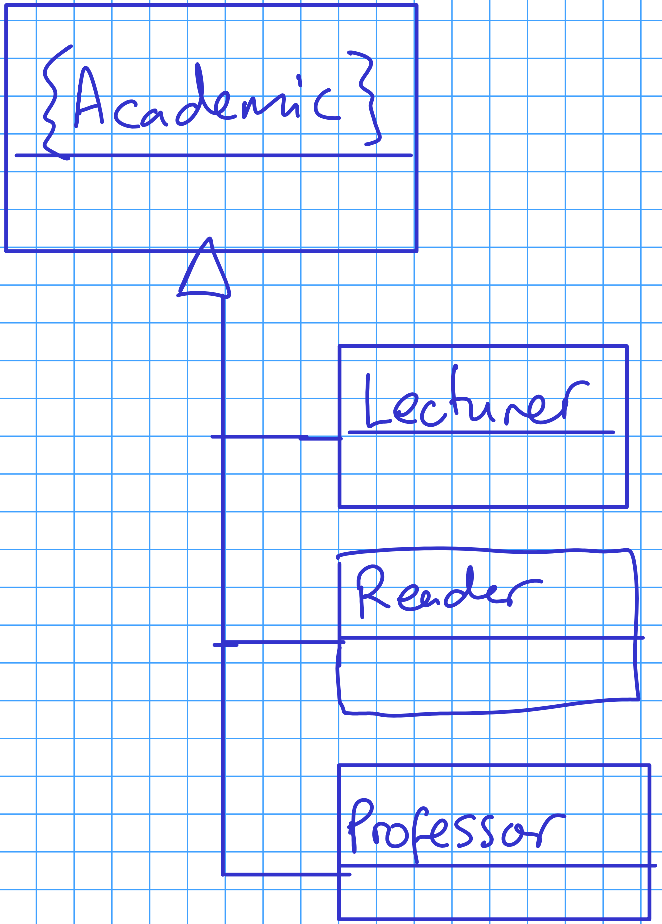
- The decorator pattern adds state and/or functionality to an object *dynamically*

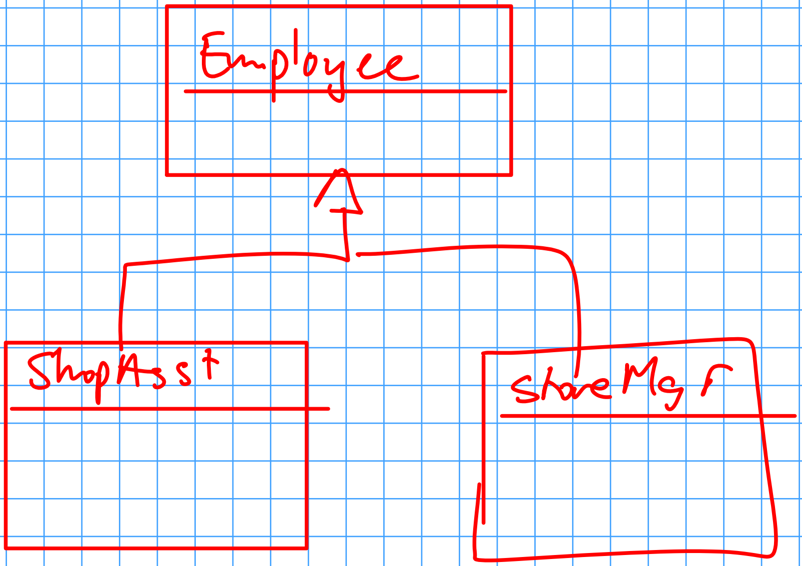
State

Abstract problem: How can we let an object alter its behaviour when its internal state changes?

Example problem: Representing academics as they progress through the rank

Solution 1   Base class



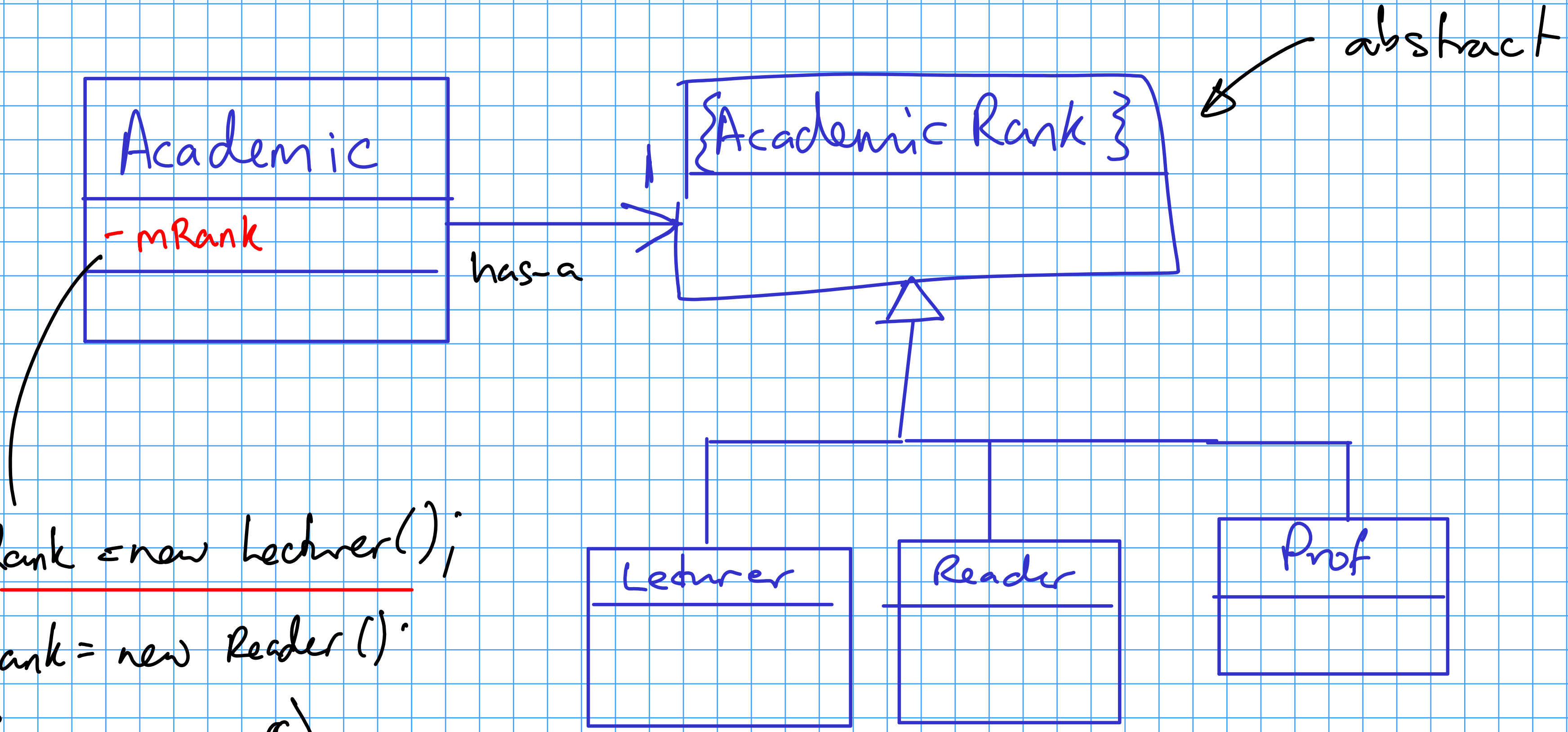


```
Academic a = new Lecturer();
a = new Reader();
[copy old a → new a]
```

X Can't promote

Solution 2

State Pattern

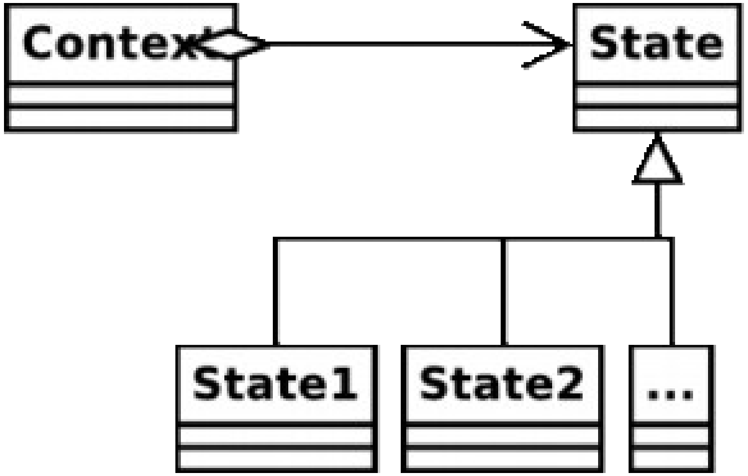


mRank = new Lecturer();

mRank = new Reader();

(no copy ref.)

State in General



- The state pattern allows an object to cleanly alter its behaviour when internal state changes

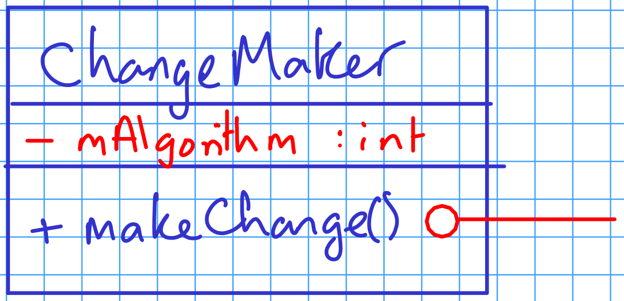
Strategy

Abstract problem: How can we select an algorithm implementation at runtime?

Example problem: We have many possible change-making implementations. How do we cleanly change between them?

Solution 1

Control flow



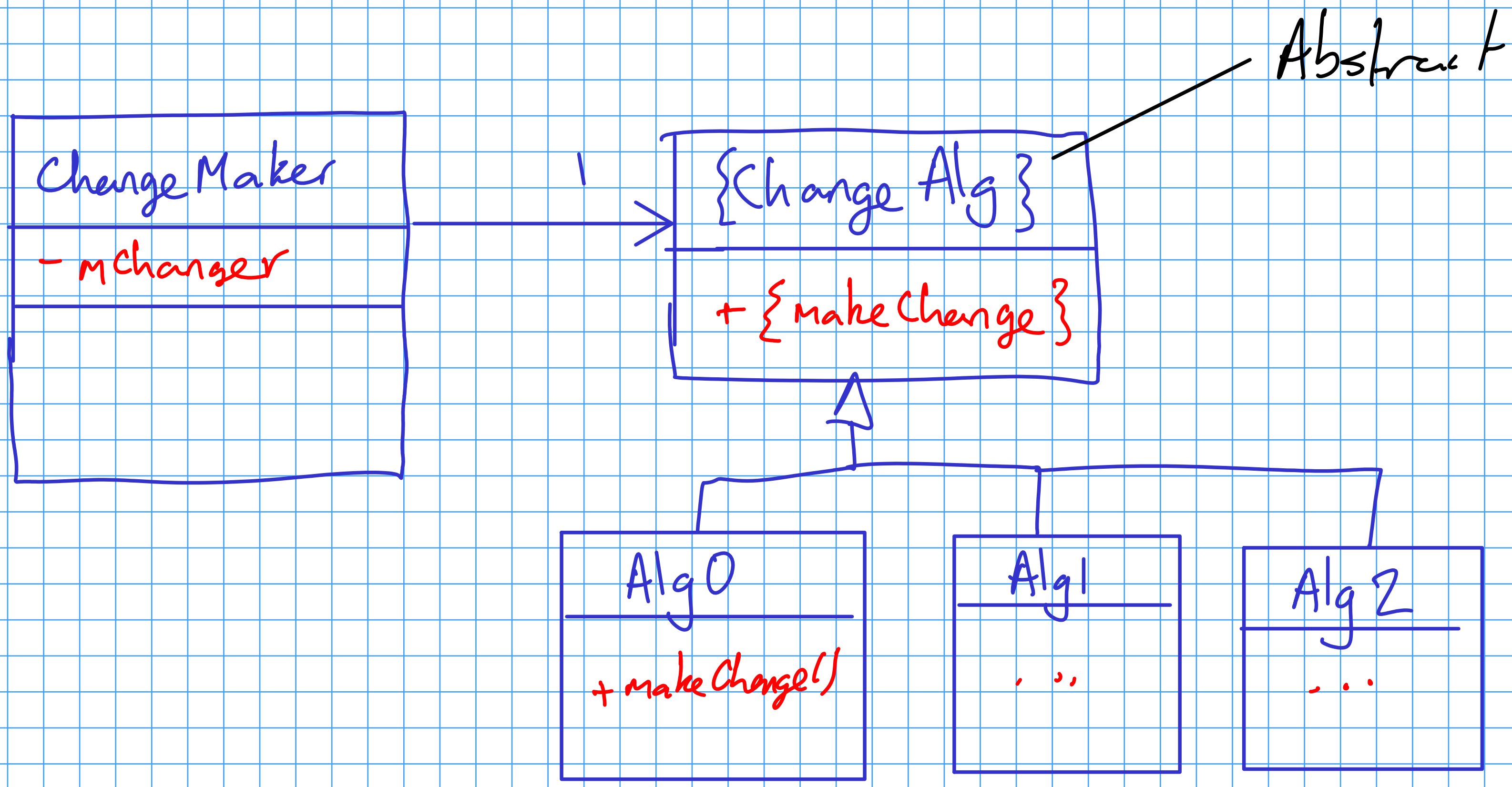
```
if (mAlgorithm == 0)
        makeChangeAlg0();
else if (mAlgorithm == 1)
        makeChangeAlg1();
else
        makeChange2();
```

✓ works.

✗ Not readable

✗ New options ⇒ edit the class (open/closed violated)

Solution 2 : Strategy pattern



✓ Encapsulates each alg completely

✓ Clean code. Pretty diagram.

X More classes in code

State vs Strategy

intent is different

State

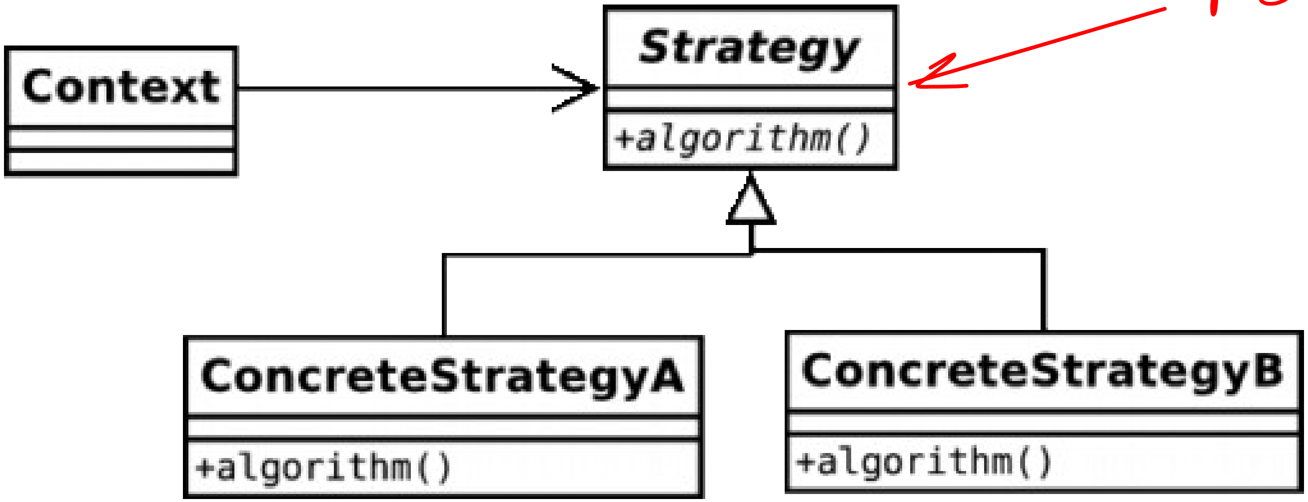
- Encapsulate state + behaviour
- Each state produces a different output
- State pattern hidden from programmers

Strategy

- Encapsulate implementation
- Each strategy produces same output.
- Explicit use

Strategy in General

- The strategy pattern allows us to cleanly interchange between algorithm implementations



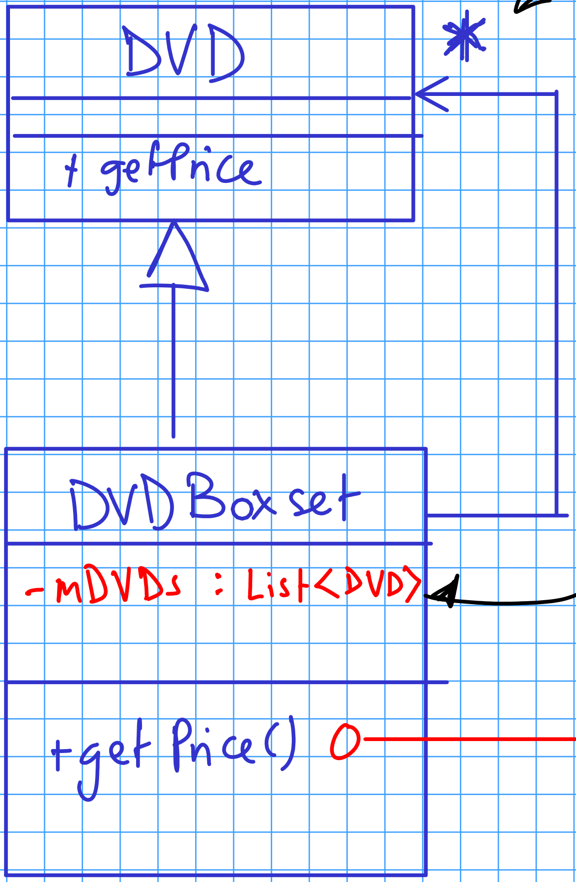
No state

Composite

Abstract problem: How can we treat a group of objects as a single object?

Example problem: Representing a DVD box-set as well as the individual films without duplicating info and with a 10% discount

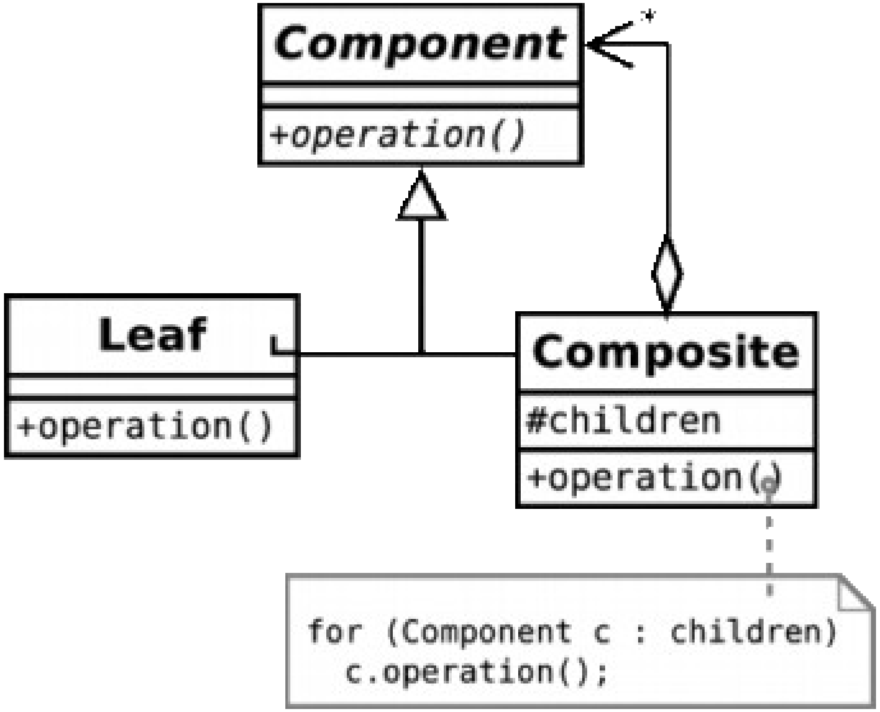
Solution : Composite



Like decorator but this is a multiple, not 1.

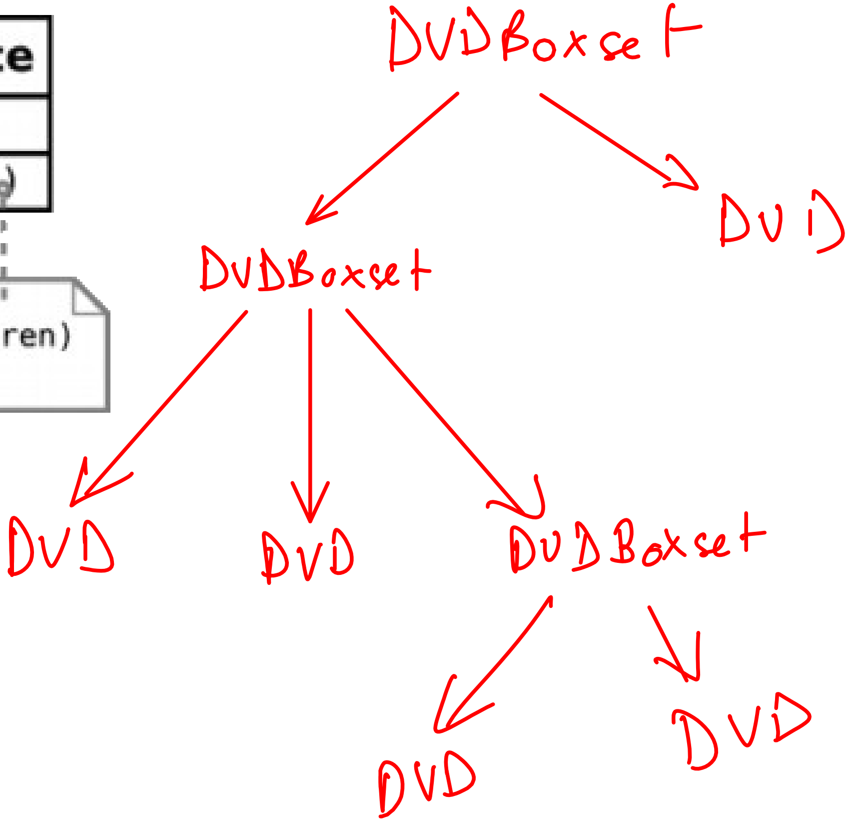
```
price = 0.0
for ( d in mDVDs)
        price = price + d.getPrice();
return price* 0.9;
```

Composite in General



```
for (Component c : children)
  c.operation();
```

- The composite pattern lets us treat objects and groups of objects uniformly



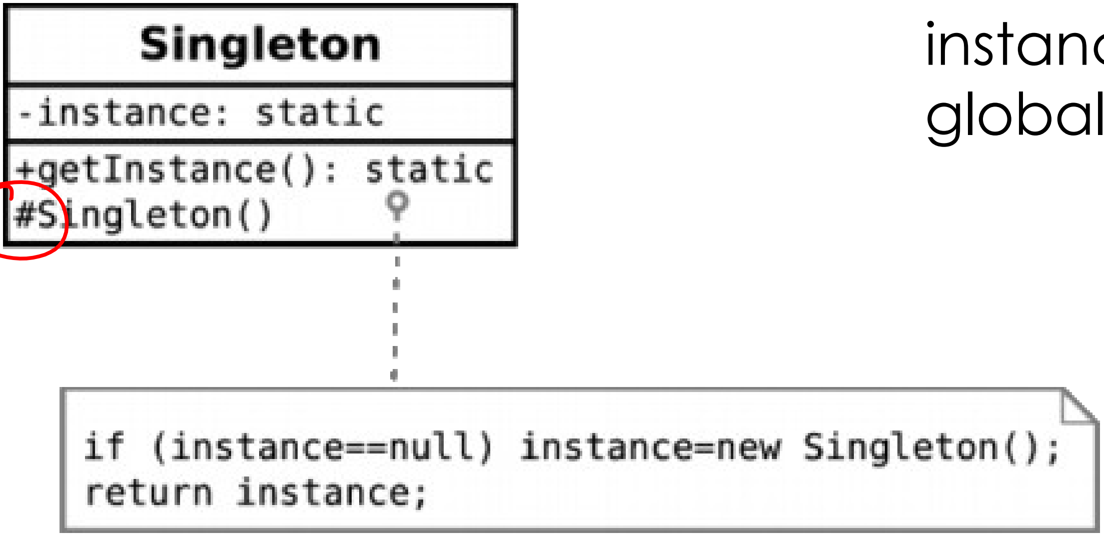
Singleton

Abstract problem: How can we ensure only one instance of an object is created by developers using our code?

Example problem: You have a class that encapsulates accessing a database over a network. When instantiated, the object will create a connection and send the query. Unfortunately you are only allowed one connection at a time.

Singleton in General

- The singleton pattern ensures a class has only one instance and provides global access to it



```
if (instance==null) instance=new Singleton();
return instance;
```

Lazy instantiation

protected

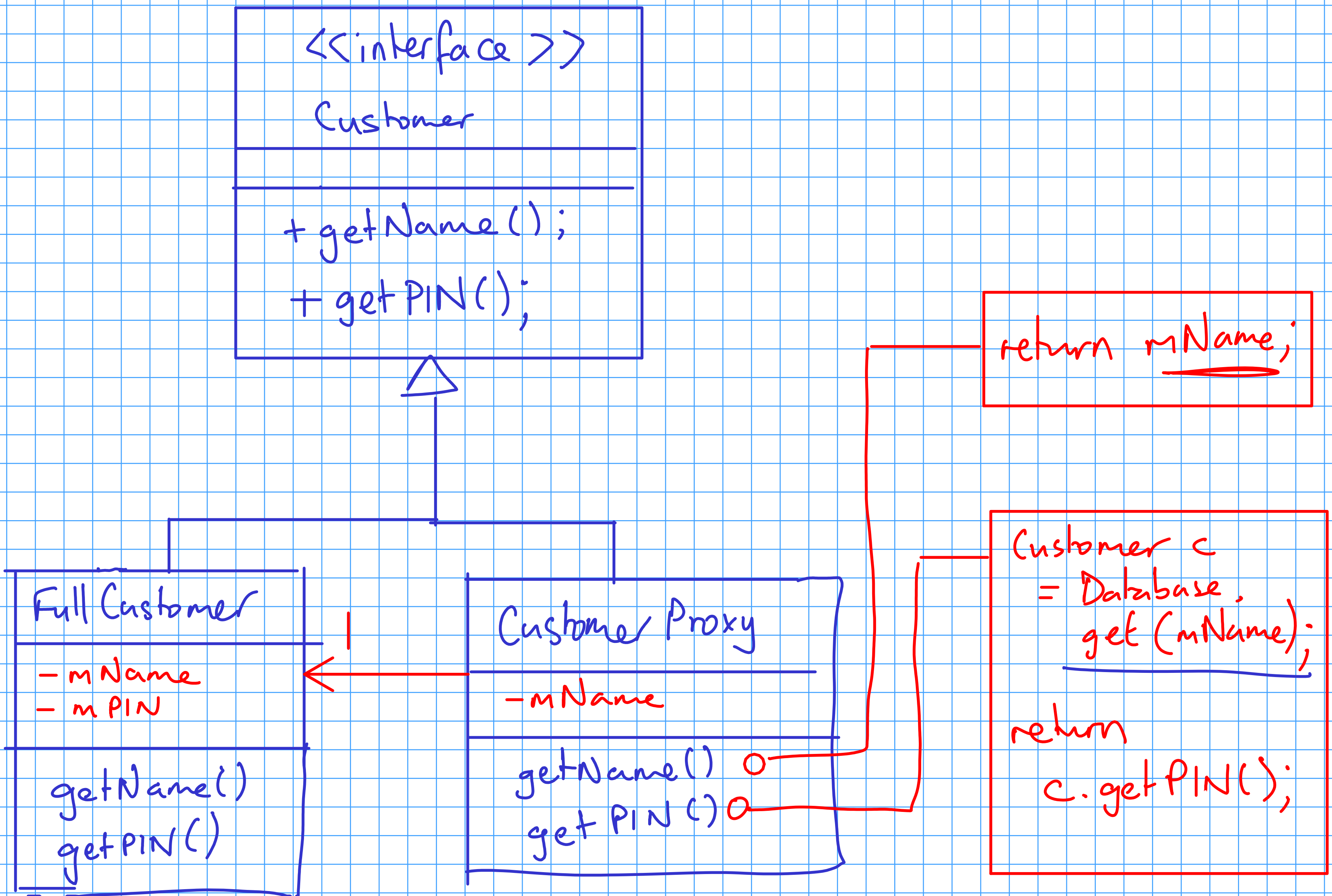
Proxy

Abstract problem: How do we have incomplete objects in memory?

Example problem: In a sales program, we might have a Customer object that holds all the customer info. Often we just need the name and address. How do we avoid loading in all the details into memory?

Solution

Virtual Proxy



```
return mName;
```

```
Customer c
  = Database.
    get(mName);
return
  c.getPIN();
```

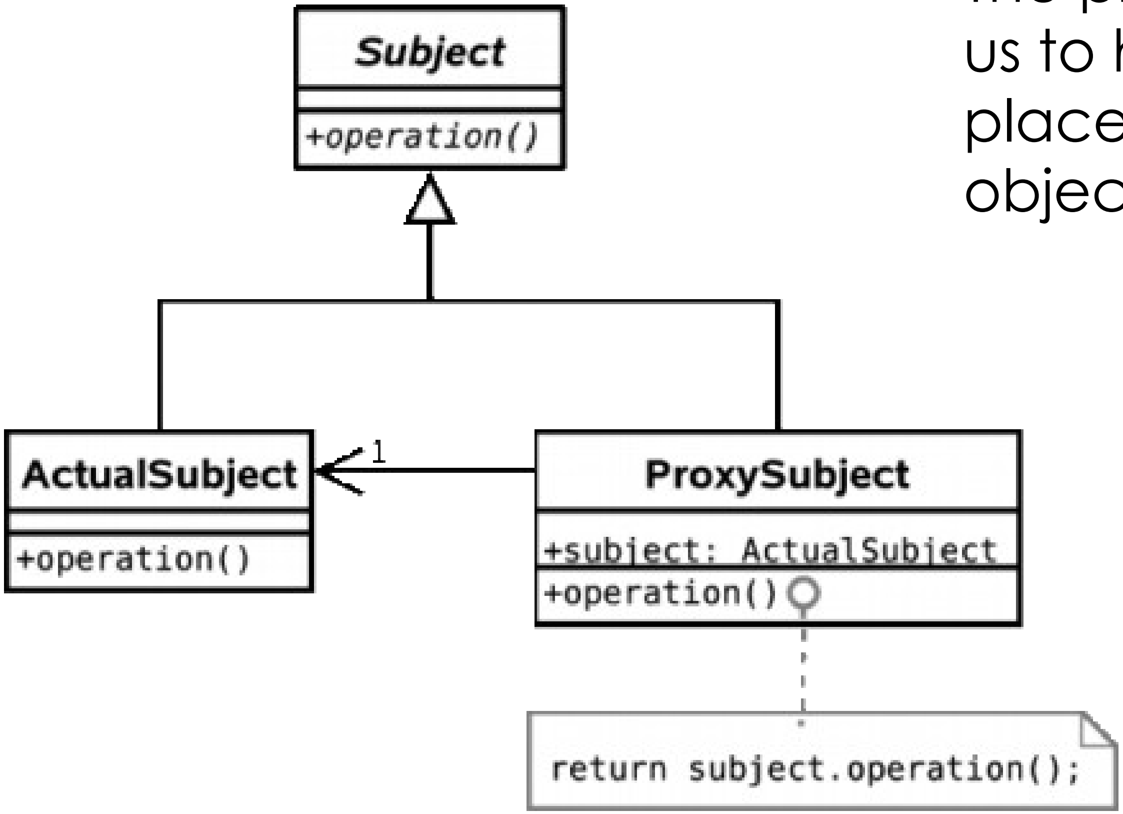
Proxy types / uses

Virtual → Not loading everything to save memory / resource. E.g. Amazon products

Protection → Not loading sensitive data e.g. Passwords

Remote → Distribute your load amongst multiple instances on multiple machines

Proxy in General



return subject.operation();

- The proxy pattern allows us to have surrogates or placeholders for actual objects

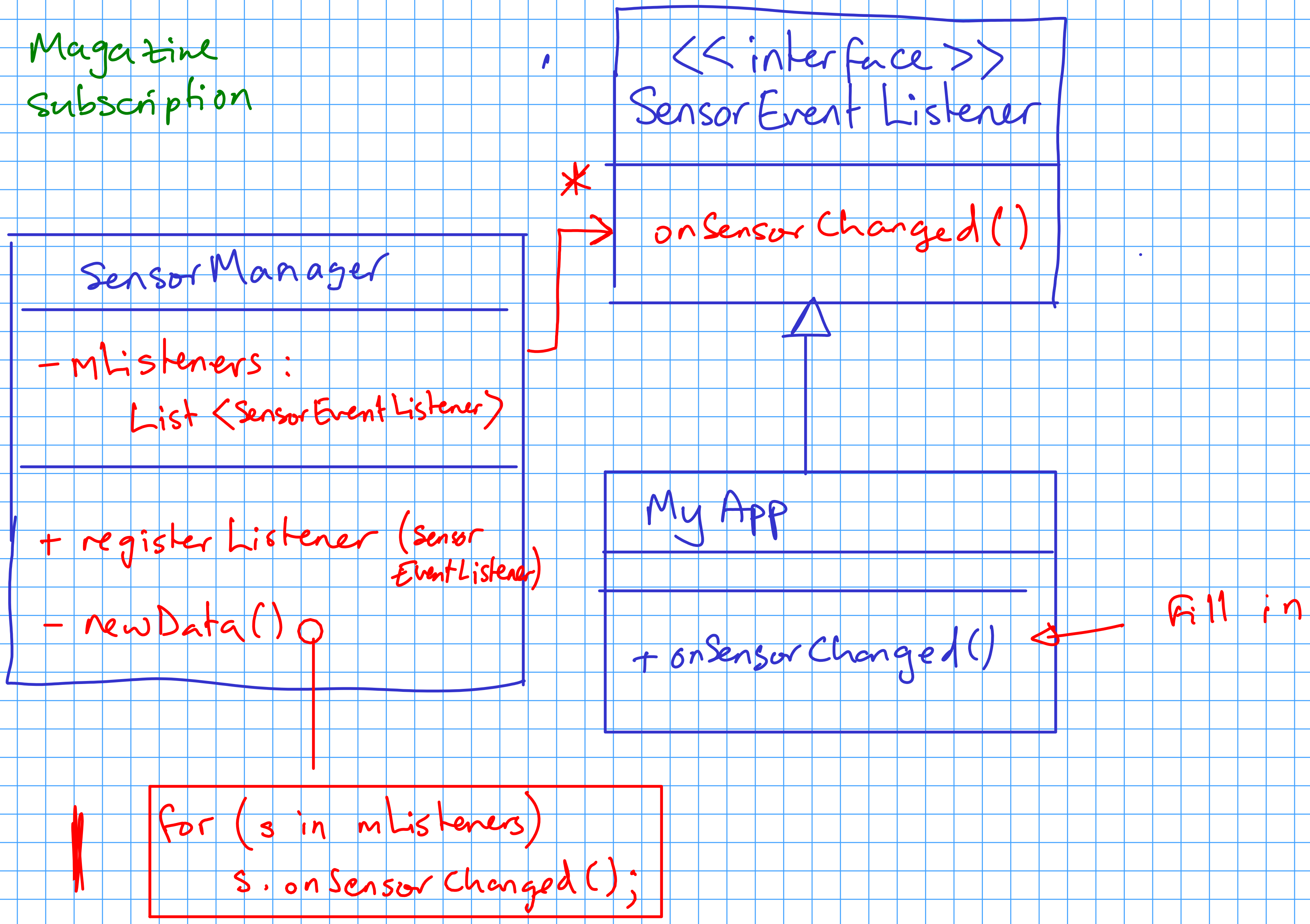
Observer

Abstract problem: When an object changes state, how can any dependent objects know?

Example problem: How can we write phone apps that react to accelerator events?

Solution : Observer pattern

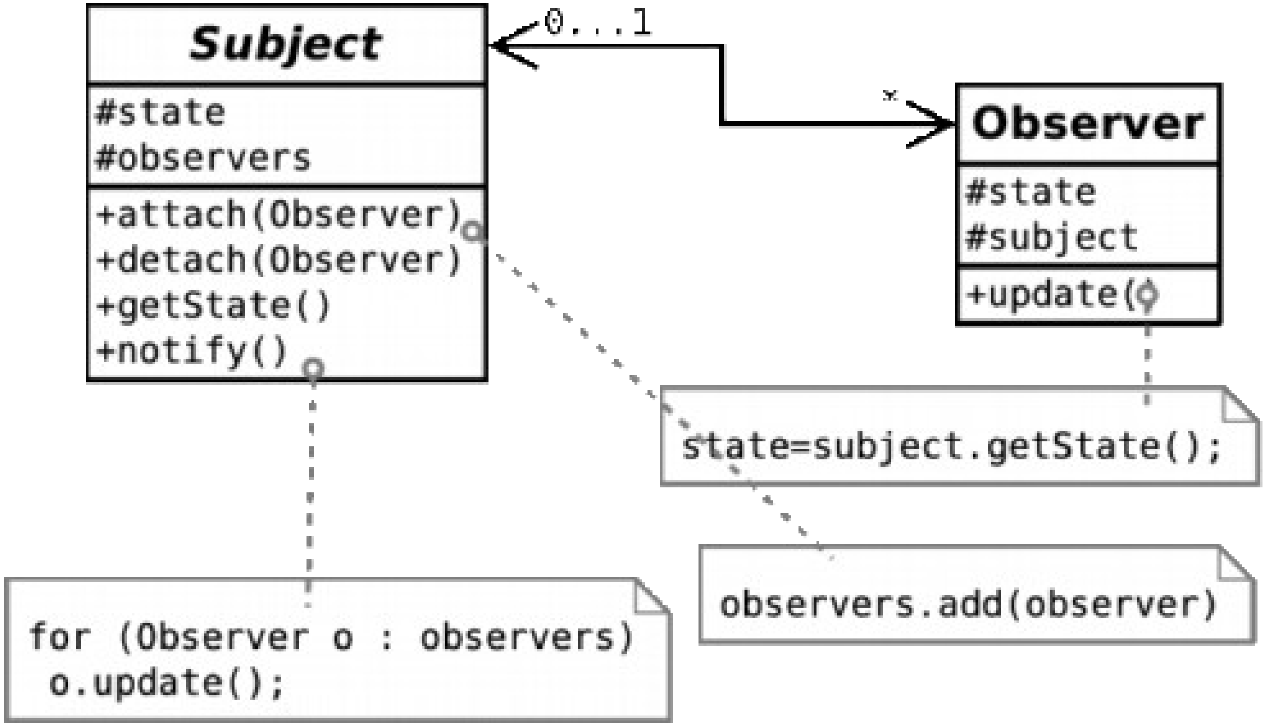
Magazine subscription



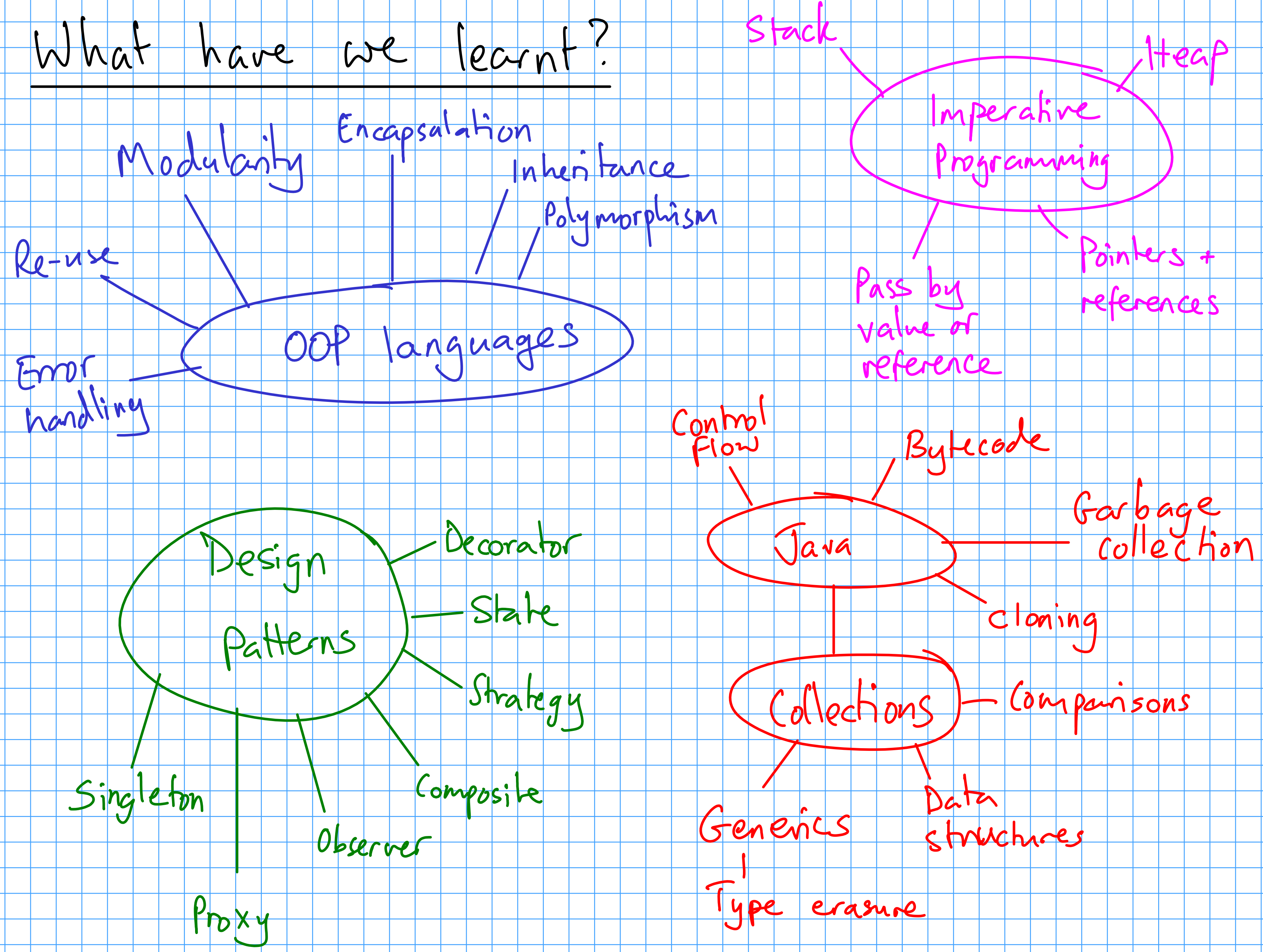
```
for (s in mListeners)
    s.onSensorChanged();
```

Observer in General

- The observer pattern allows an object to have multiple dependents and propagates updates to the dependents automatically.



What have we learnt?



Where from here?

Programming is simultaneously an art, a skill and a science

Practice makes perfect.