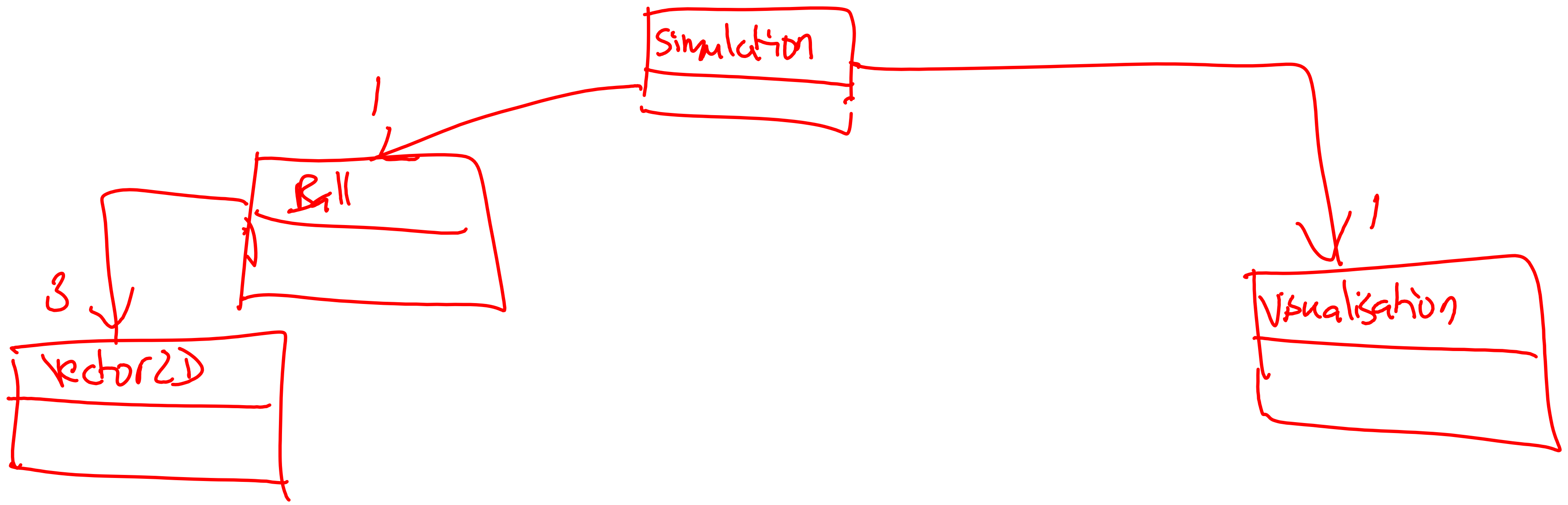
Access Modifiers

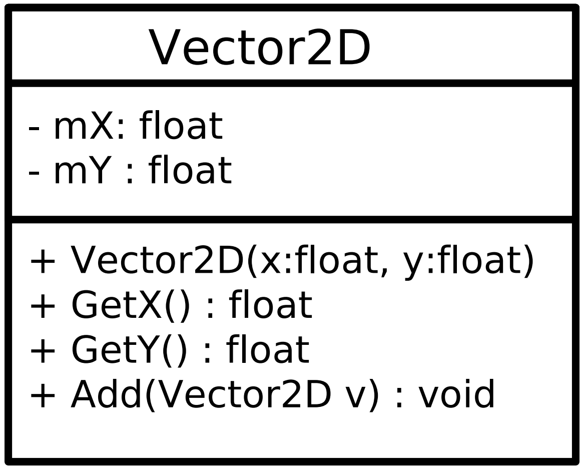
- e.g. **public**, **protected**, **private** in Java and C++
- Can apply to fields *and* methods
  - If a method implementation gets very long, you might want to split it into smaller methods. We make the shorter methods private so no one can call them externally, and expose one public method (that makes use of those private methods)
- Not all OO languages have full access control
  - If interested, take a look at the mess in the python language...   `

Simulate a ball bouncing on the floor. The ball should have a coefficient of restitution, k, as well as position, velocity, and acceleration. The floor should be at height 0.0m. A visualisation should be provided.



Vector2D Example

- We will create a class that represents a 2D vector



Immutability

1. Reduces ambiguities
2. Thread safe
3. Easier to test and understand.
4. Makes copying easy

X Potentially more memory

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

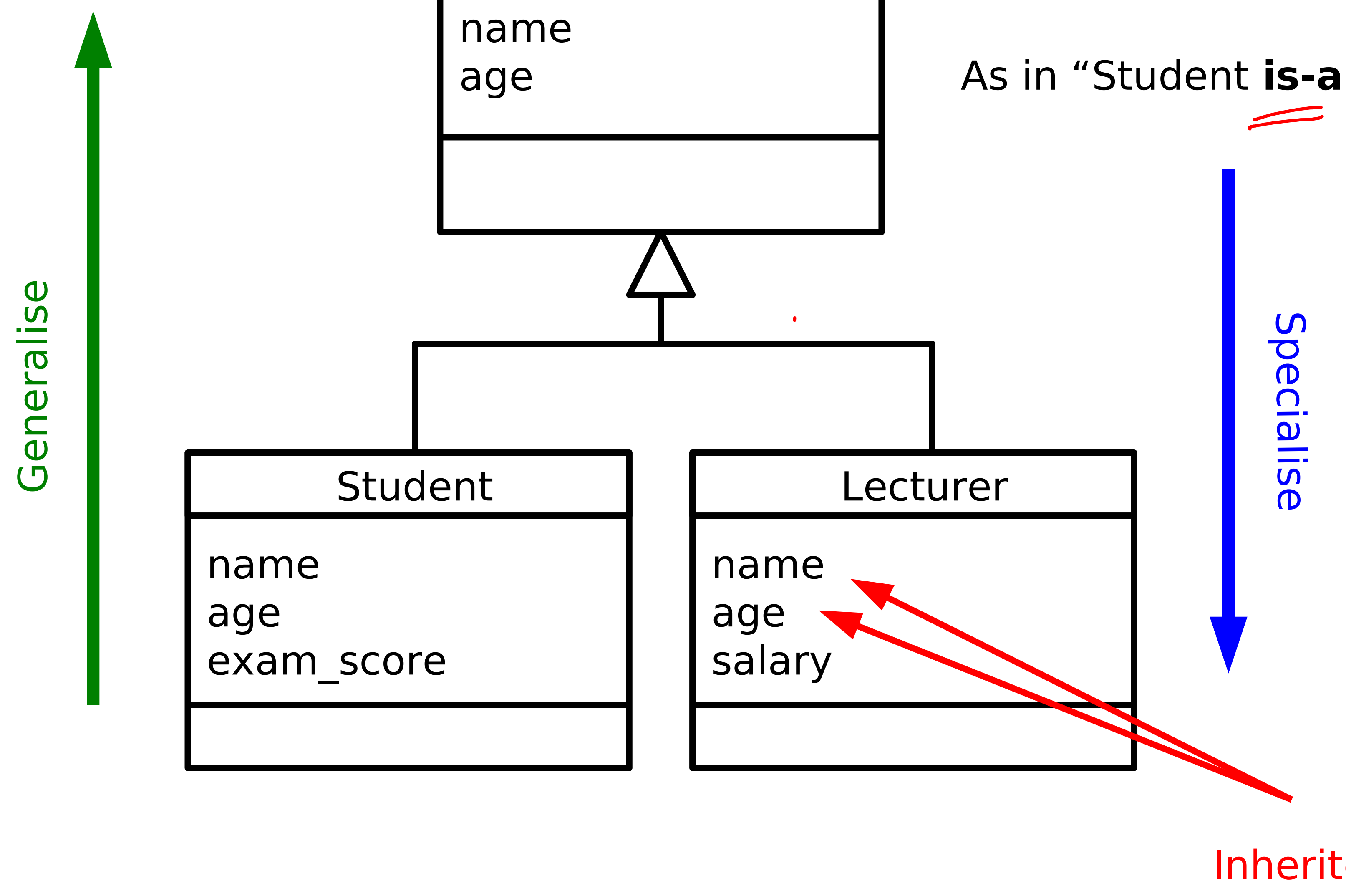
(I should not have used public variables here, but I did it to keep things simple)

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

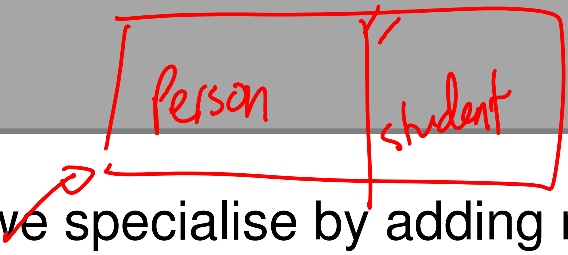
Representing Inheritance Graphically



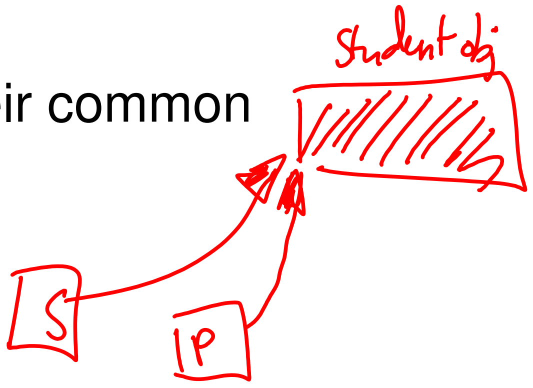
Also known as an “is-a” relation

As in “Student **is-a** Person”

Casting/Conversions



- As we descend our inheritance *tree* we specialise by adding more detail (a salary variable here, a dance() method there)
- So, in some sense, a Student object has all the information we need to make a Person (and some extra).
- It turns out to be quite useful to group things by their common ancestry in the inheritance tree
- We can do that semantically by expressions like:



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

This is a *widening* conversion (we move up the tree, increasing generality: always OK)

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

X

This would be a *narrowing* conversion (we try to move down the tree, but it's not allowed here because the real object doesn't have all the info to be 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;
  }

}
```

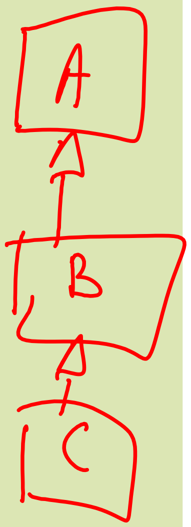
Student inherits this as a public variable and so can access it

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

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

Fields and Inheritance: Shadowing

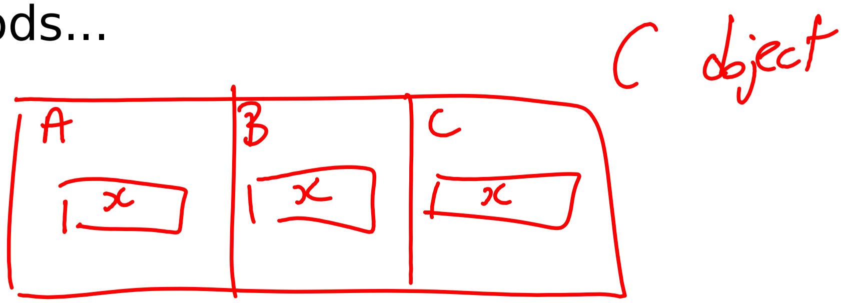
```
class A {
   public int x;
}

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;
  }
}
```



What happens here?? There is an inheritance tree (A is the parent of B is the parent of C). Each of these declares an integer field with the name **x**.

In memory, you will find three allocated integers for every object of type C. We say that variables in parent classes with the same name as those in child classes are *shadowed*.

Note that the variables are being shadowed: i.e. nothing is being replaced. This is contrast to the behaviour with methods...


C object
A
B
C
x
x
x