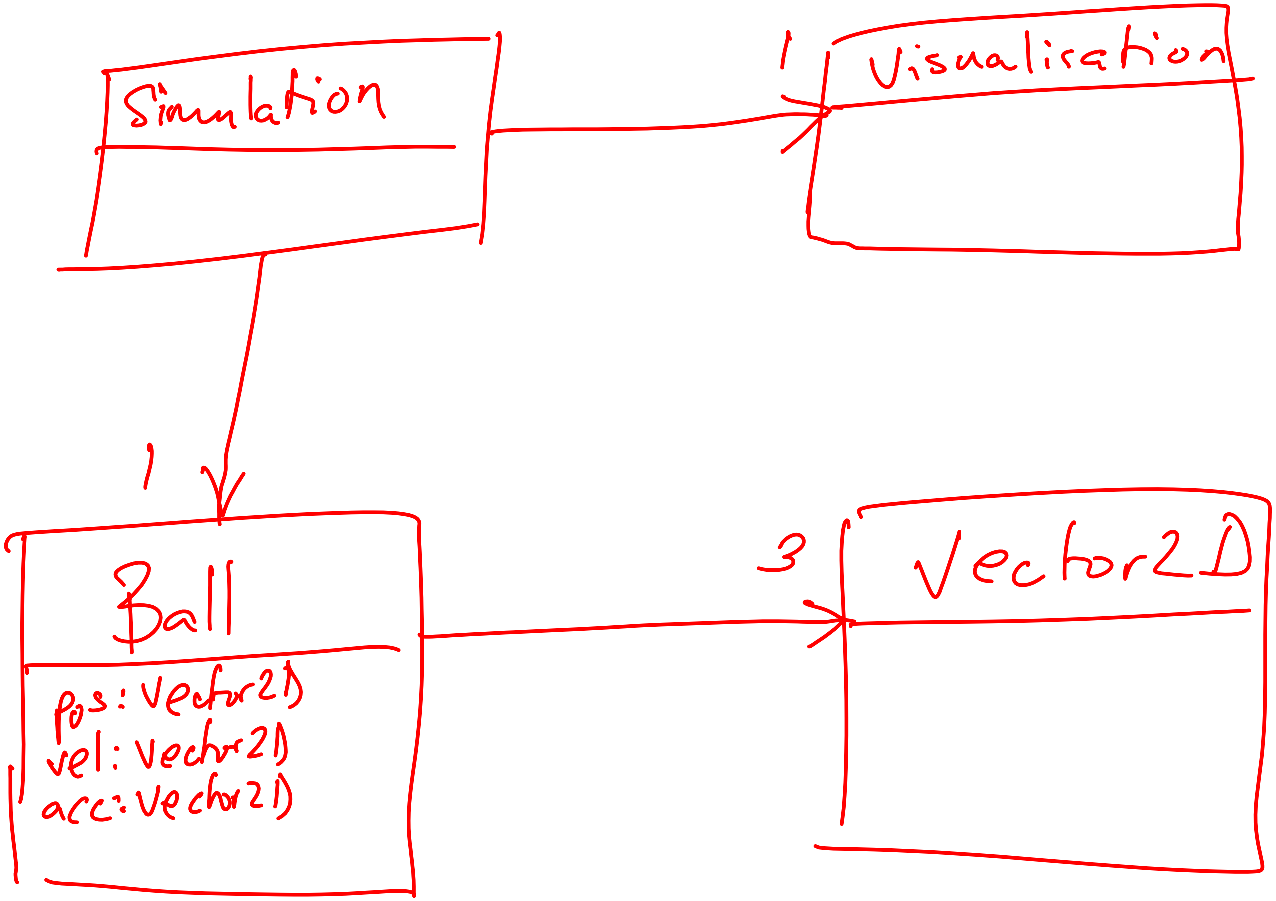
A simulation and visualisation of a moving ball



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

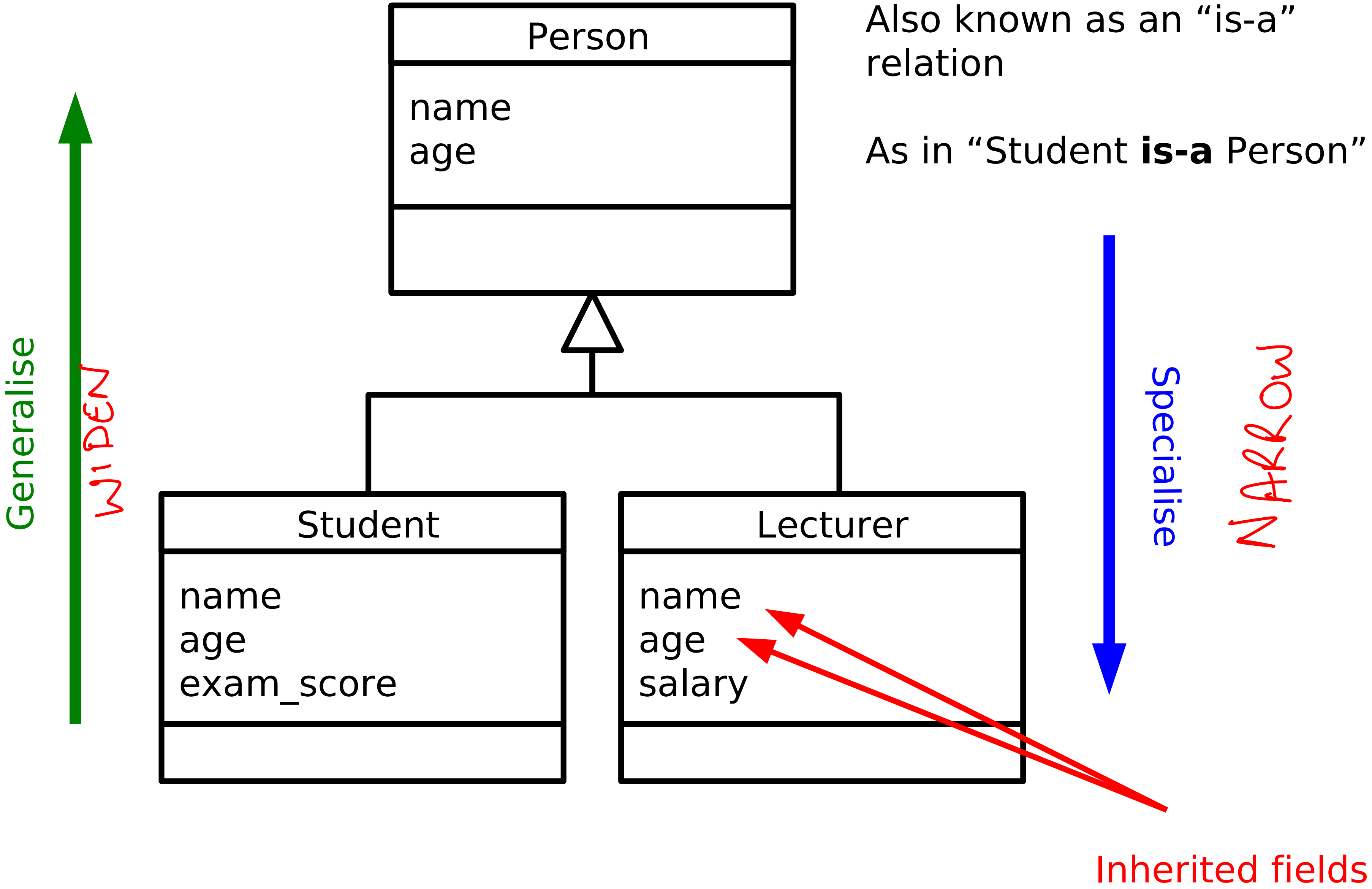
```
class Person {
   public int age;
   Public String name;
}

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

WIDEN

Specialise

NARROW

Inherited fields

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;
```

✓

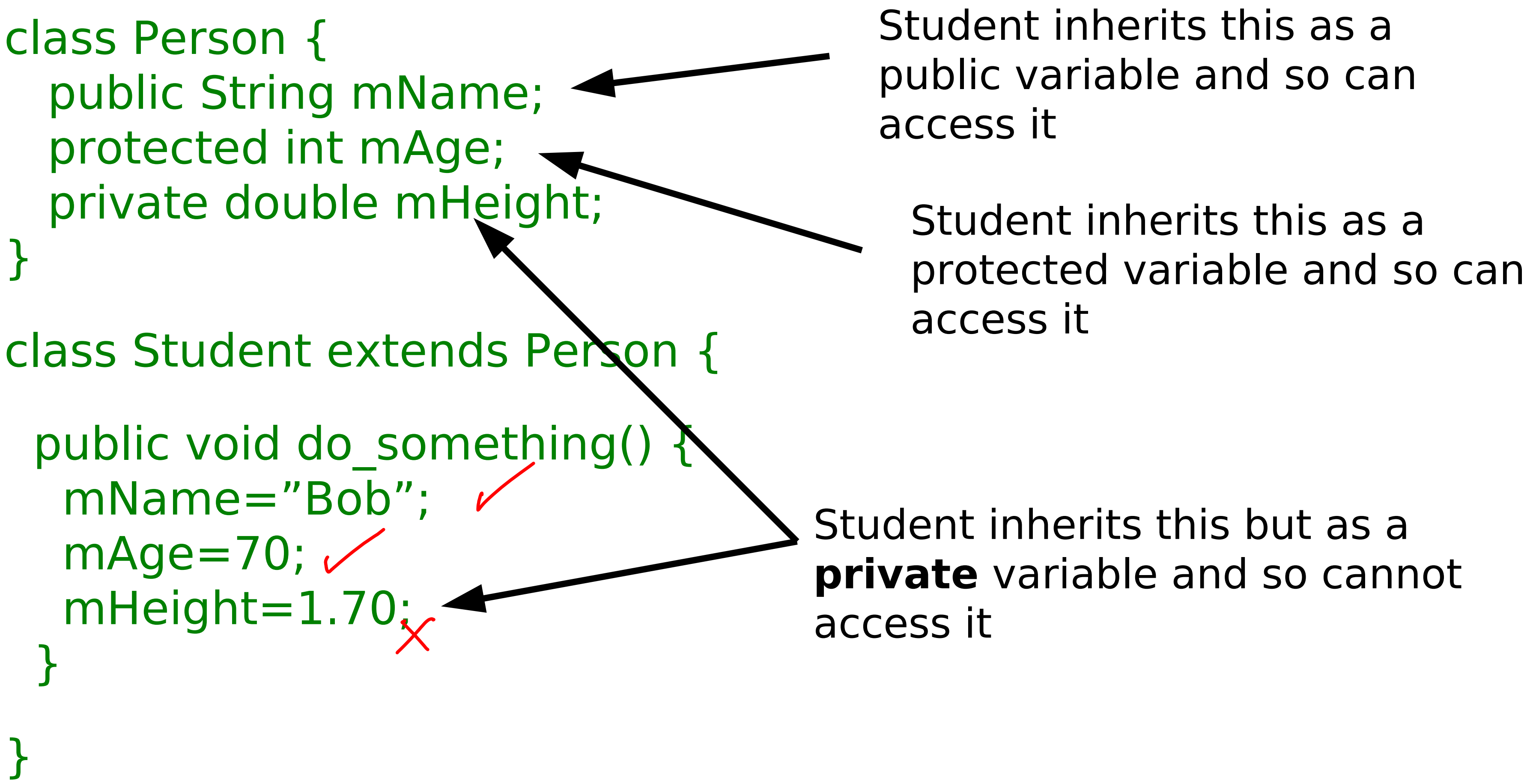
This is a *widening* conversion (we move up the tree, increasing generality: always OK)

```
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;  ✗
  }

}
```

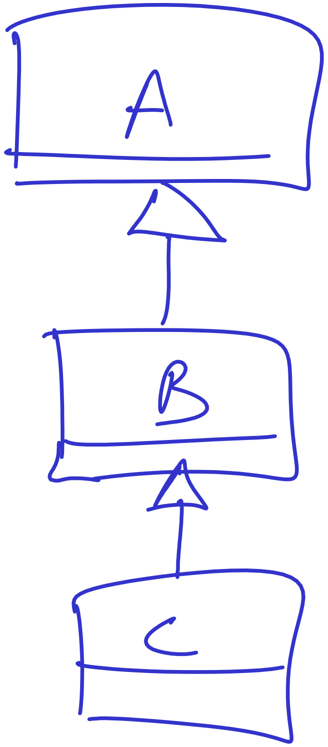
Student inherits this as a public variable and so can access it

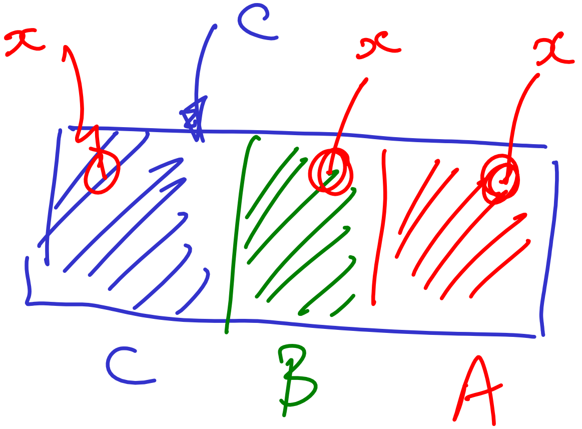
Student inherits this as a protected variable and so can access it

Student inherits this but 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;
  }
}
```





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 {
}
```

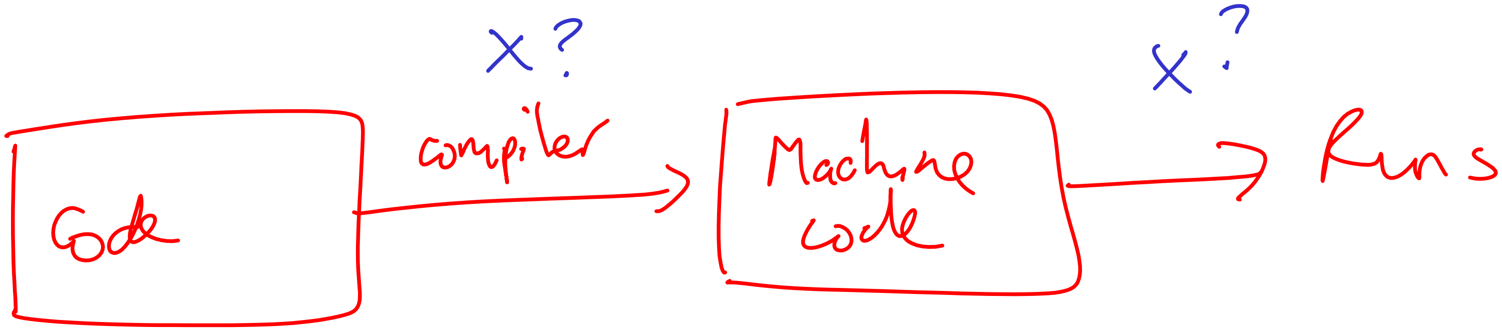
Lecturer just inherits the default implementation and jiggles

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 must 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