Section: Object Oriented Programming

Custom Types

- You saw that there was an advantage to declaring your own types in ML
  - First you declared a type and then you wrote functions that could act on it

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

See Workbook 3

Classes, Instances and Objects I

- Primitive types are pre-defined e.g. int defines 32-bit integer in Java
- We create **instances** of a primitive type by declaring a variable of that type
  - E.g.

```
int x=7;
int y=6;
```

    declares two instances of type int

Classes, Instances and Objects II

- Classes are basically templates for various ***concepts***
- We create ***instances*** of classes in a similar way. e.g.

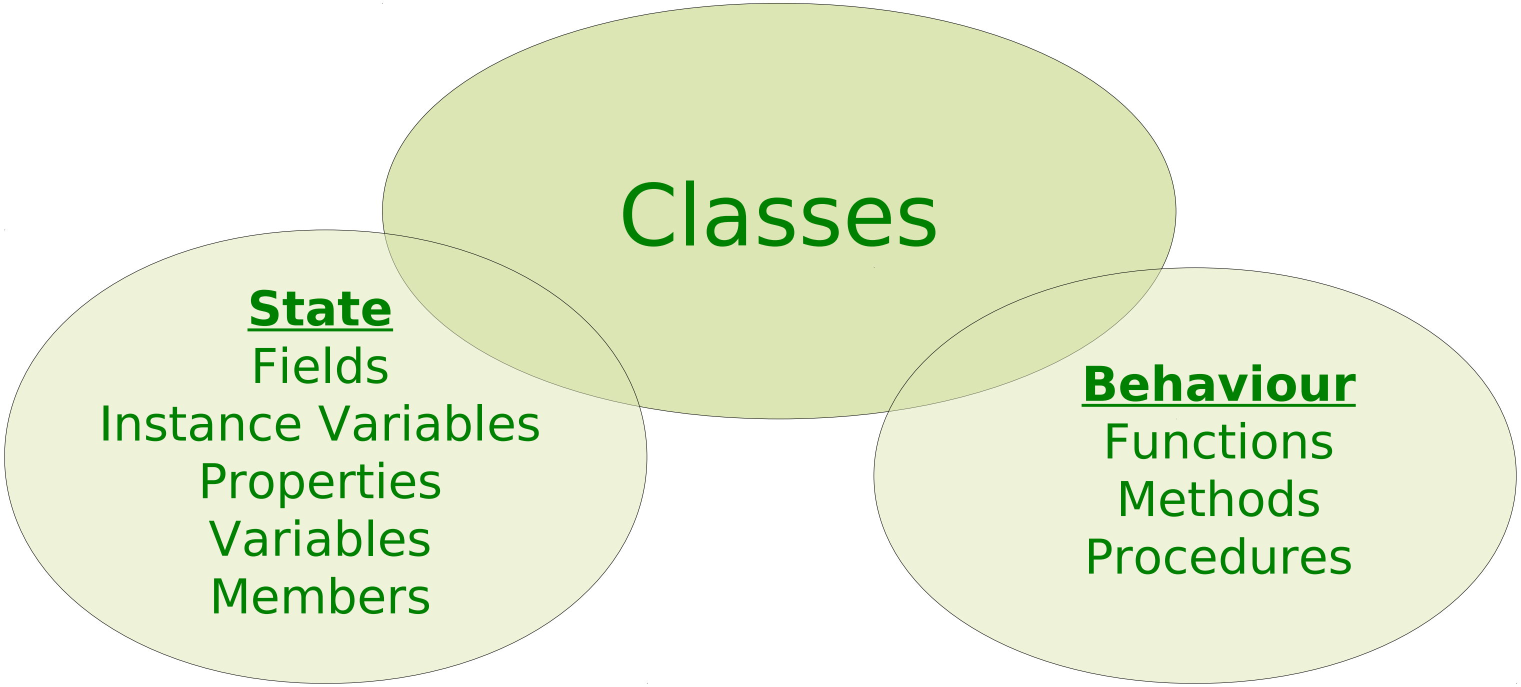
  Reference

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

  Constructor

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

Loose Terminology (again!)



Classes

State
Fields
Instance Variables
Properties
Variables
Members

Behaviour
Functions
Methods
Procedures

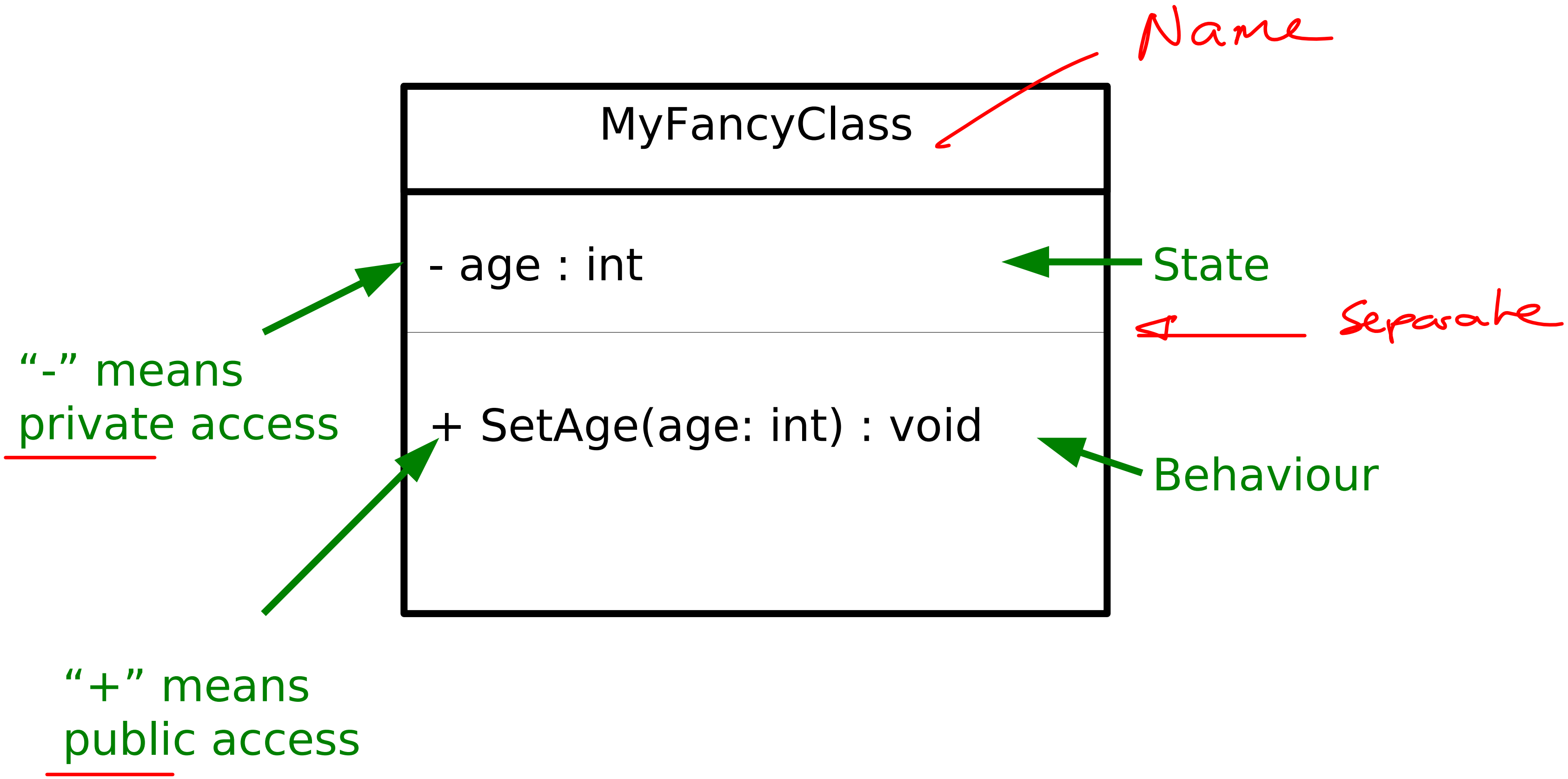
Identifying Classes

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

Adjectives ⇒ Properties/State

"Write a simulation of the Earth's orbit around the Sun"

Representing a Class Graphically (UML)



Name

MyFancyClass

- age : int

State

Separate

+ SetAge(age: int) : void

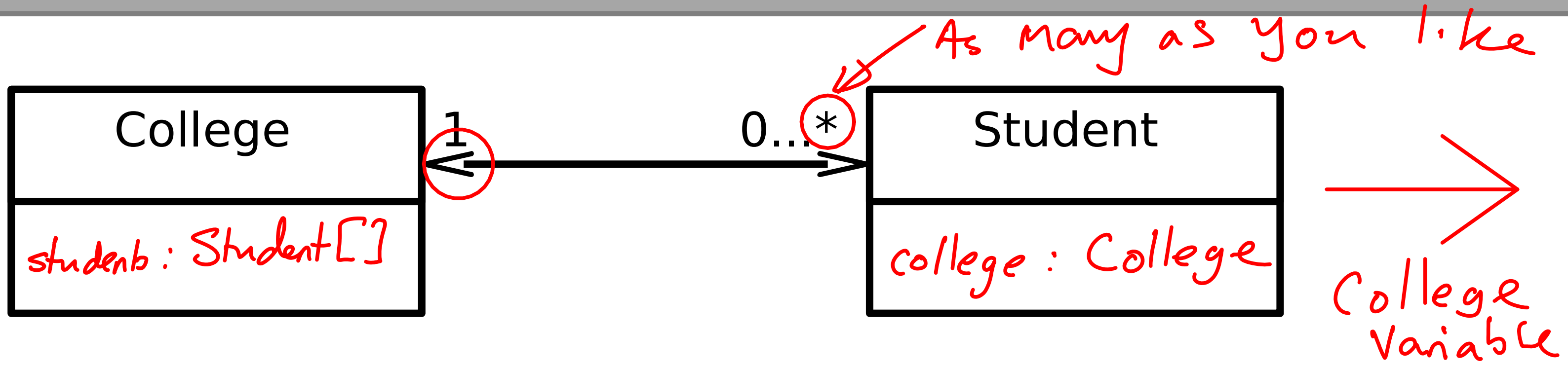
Behaviour

“-” means private access

“+” means public access

The has-a Association

Student var.



- 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();
    }

}
```

State

Behaviour

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 an object of type MyFancyClass in memory and get a reference to it

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;

}
```

Access modifier

Class state

Class behaviour

'Magic' start point
for the program

Create an object of
type MyFancyClass

Section: OOP Concepts

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

```
          public
class Student {
  int age;
};
                          object
void main() {
  Student s = new Student();   1
  s.age = 21;

  Student s2 = new Student();  2
  s2.age=-1;

  Student s3 = new Student();  3
  s3.age=10055;
}
```

- Here we create 3 Student objects when our program runs
- Problem is obvious: nothing stops us (*or anyone using our Student class*) from putting in garbage as the age
- Let's add an *access modifier* that means nothing outside the class can change the age

Encapsulation II

```
class Student {
   private int ~~age~~; years
   public
   boolean SetAge(int a) {
      if (a>=0 && a<130) {
  years  ~~age~~=a;
         return true;
      }                      setter
      return false;
   }
                           year
   int GetAge() {return ~~age~~;}
}

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

Getter

- Now nothing outside the class can access the *age* variable directly
- Have to add a new method to the class that allows *age* to be set (but only if it is a sensible value). i.e. **SetAge()**
- Also needed a **GetAge()** method so external objects can find out the age.

Encapsulation III

- We hid the state implementation to the outside world (no one can tell we store the age as an int without seeing the code), but provided mutator methods to... errr, mutate the state!

Data Hiding

- This is *data encapsulation*
  - We define interfaces to our objects without committing long term to a particular implementation
- Advantages
  - We can change the internal implementation whenever we like so long as we don't change the interface other than to add to it (E.g. we could decide to store the age as a float and add GetAgeFloat())
  - Encourages us to write clean interfaces for things to interact with our objects

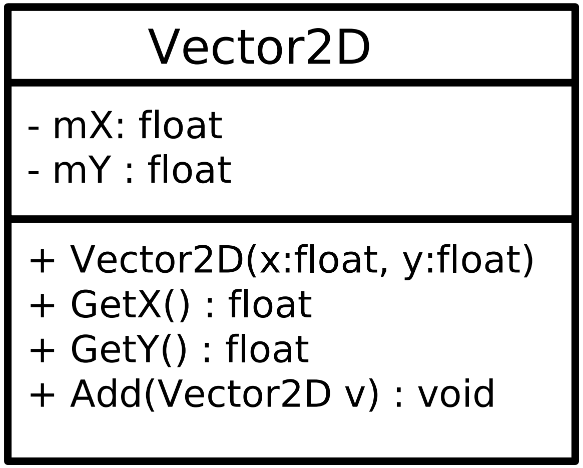
Access Modifiers

package in Java

- 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 a **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...

Vector2D Example

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



Vector2D

- mX: float
- mY : float

+ Vector2D(x:float, y:float)
+ GetX() : float
+ GetY() : float
+ Add(Vector2D v) : void

To make a class immutable

1. Make state private
2. Remove setters
3. Add constructor
4. Make state final

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

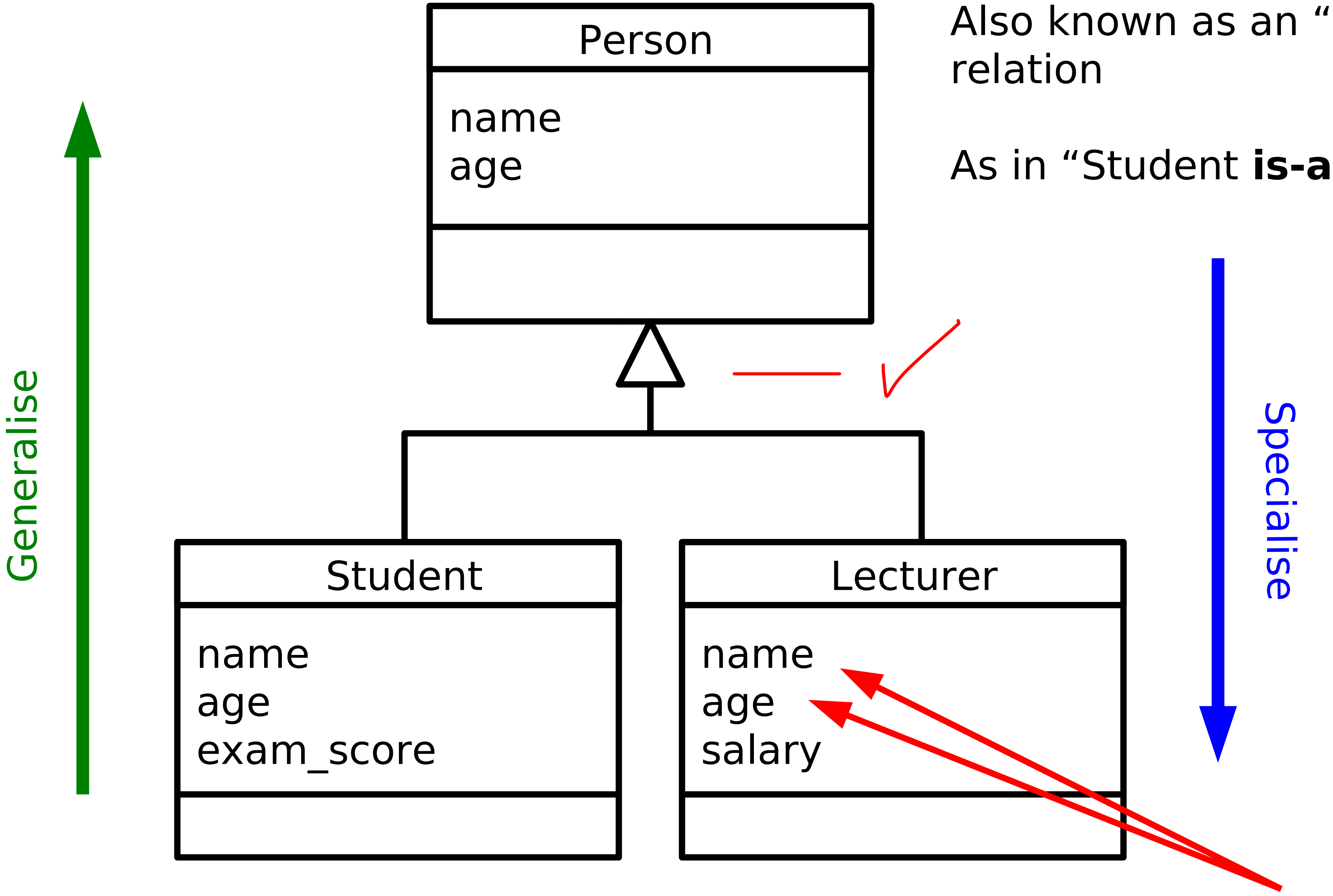
```
class Person {
   public int age;
   Public String name;
}

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”



Inherited fields