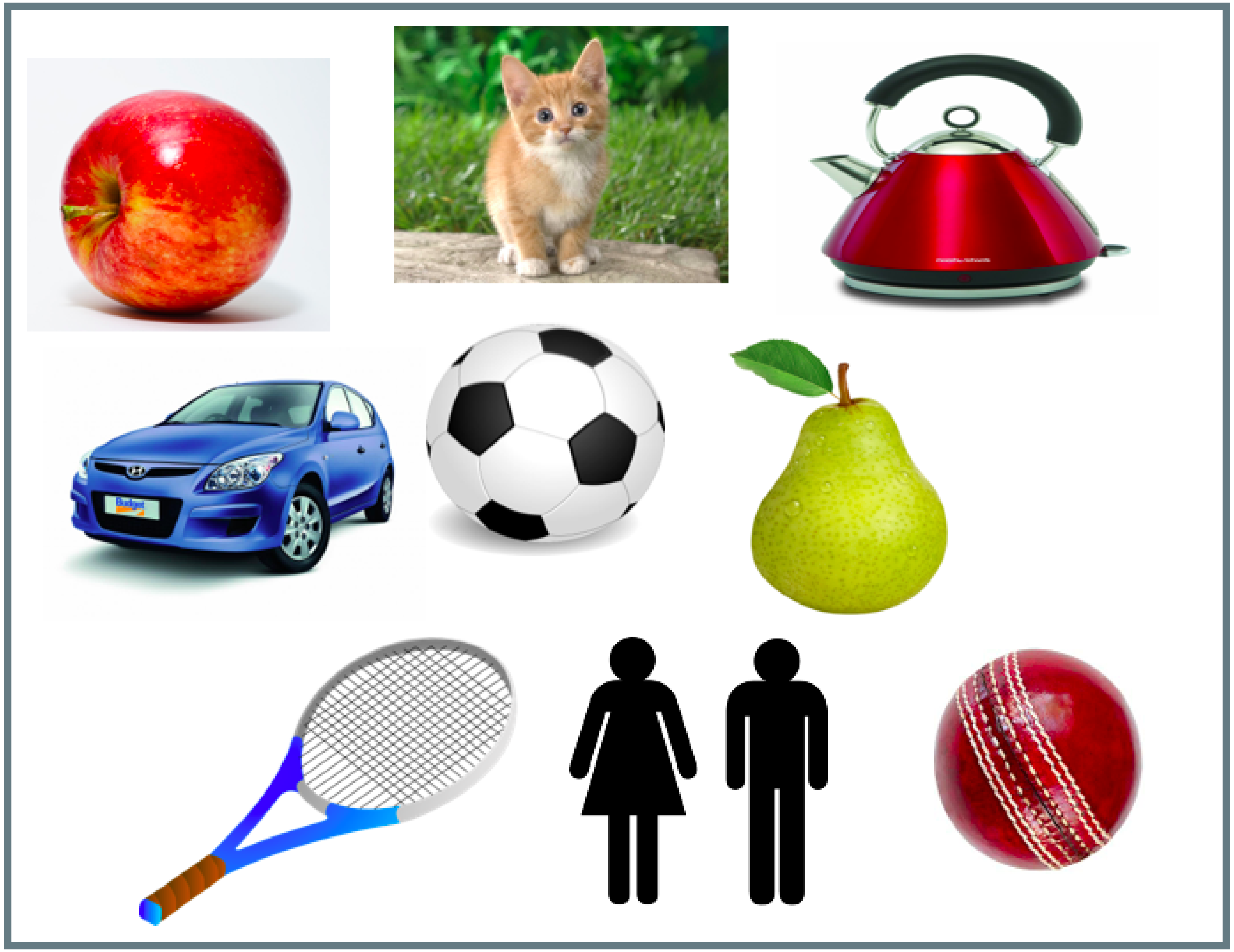
OBJECTS AND CLASSES

Benedict R. Gaster / @cuberoo_

UWE BRISTOL University of the West of England

bettertogether

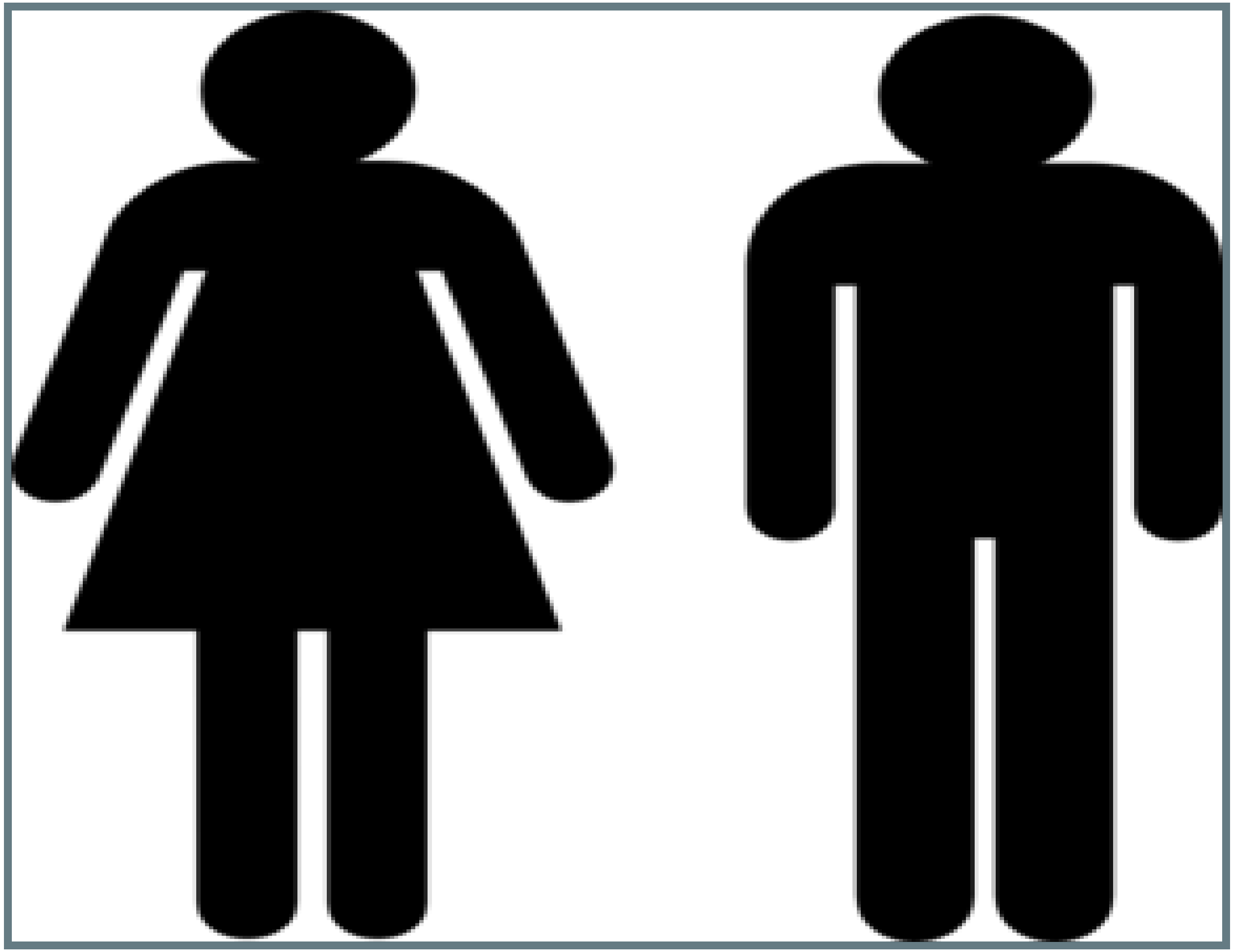
OBJECTS



OBJECTS

- Classification of features
    - Assocate data , often called attributes, that provide characteristics
    - Assocate functions, often called methods, that provide behaviour

OBJECTS - PERSON



ATTRIBUTES

- For a person might we know
  - Name
  - Date of birth
  - Gender
  - Nationality

ATTRIBUTES HAVE TYPES

```
name        : String
dob         : (Int,Int,Int)
gender      : Gender
nationality : Nation
```

ATTRIBUTES ARE OFTEN OBJECTS TOO

```
gender      : Gender
nationality : Nation
```

METHODS (I.E. FUNCTIONS)

- For a person might we be able to compute
  - Get Name
  - Is pensioner
  - Is Female
  - Is European

CLASSES

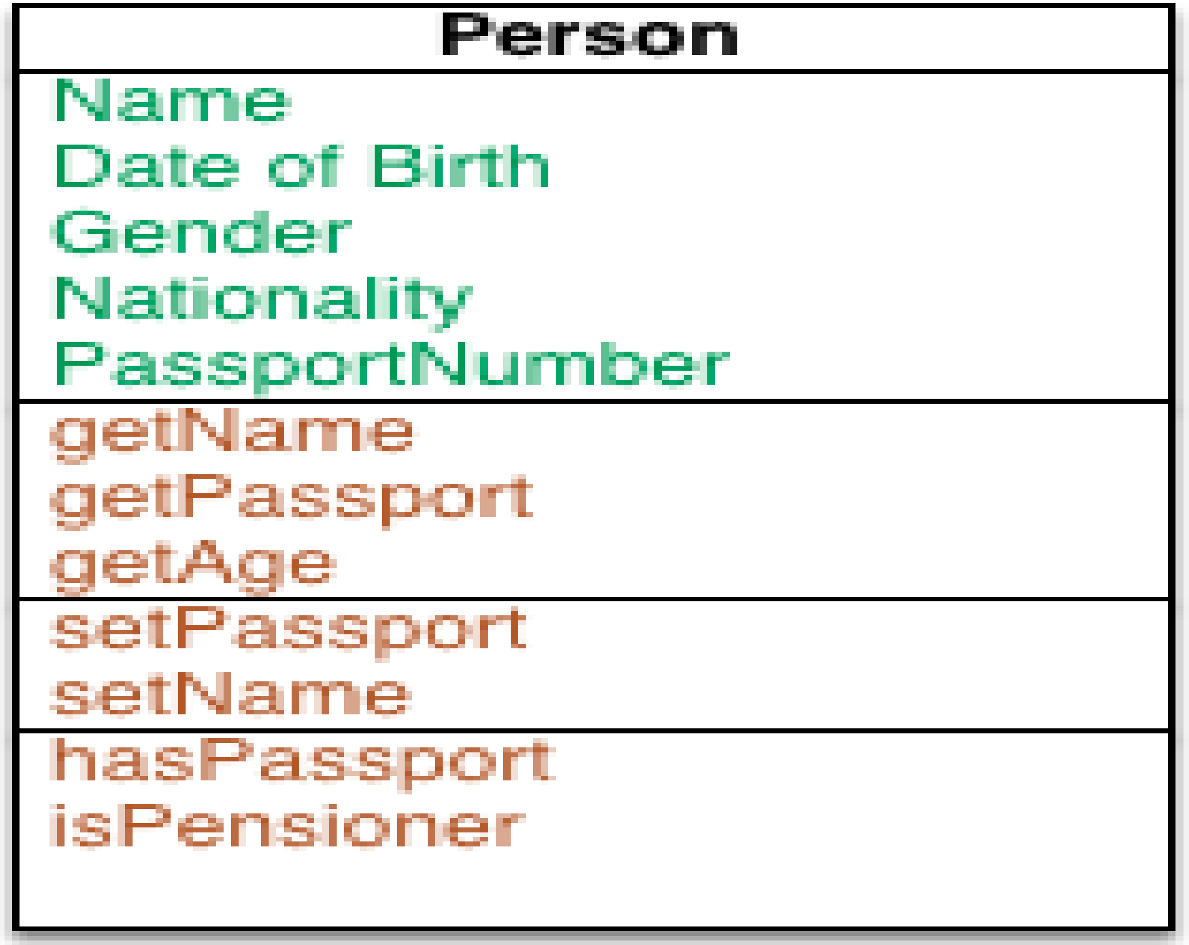
- Abstract represention (description) of an object
- Group attributes and methods together
- Encapsulate data (attributes) and methods (behaviour)

CLASSES ARE TYPES TOO

A class definition in Scala introduces a new type

```
class A {
    // attributes are defined here
    // methods are defined here
}
```

CLASS DIAGRAM



Person

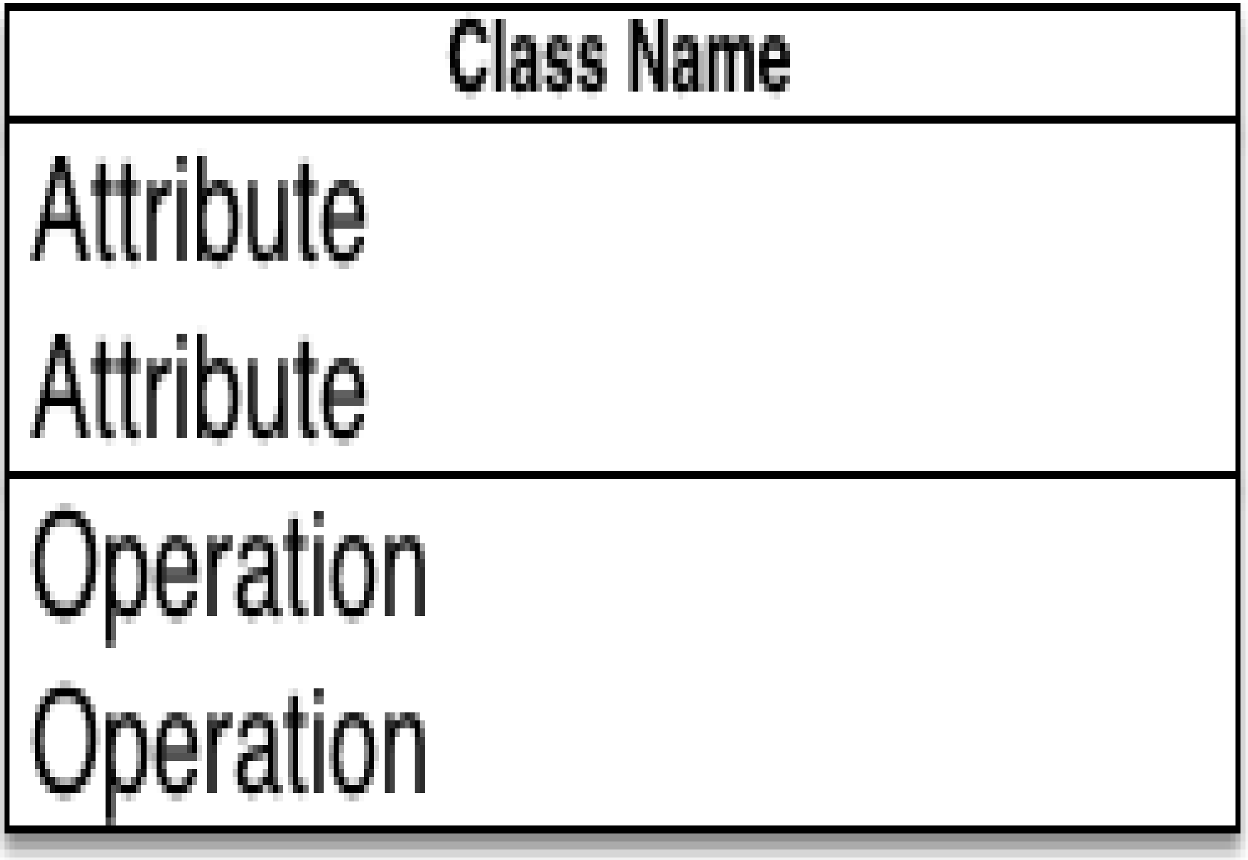
Name
Date of Birth
Gender
Nationality
PassportNumber

getName
getPassport
getAge

setPassport
setName

hasPassport
isPensioner

CLASS DIAGRAM



Class Name

Attribute

Attribute

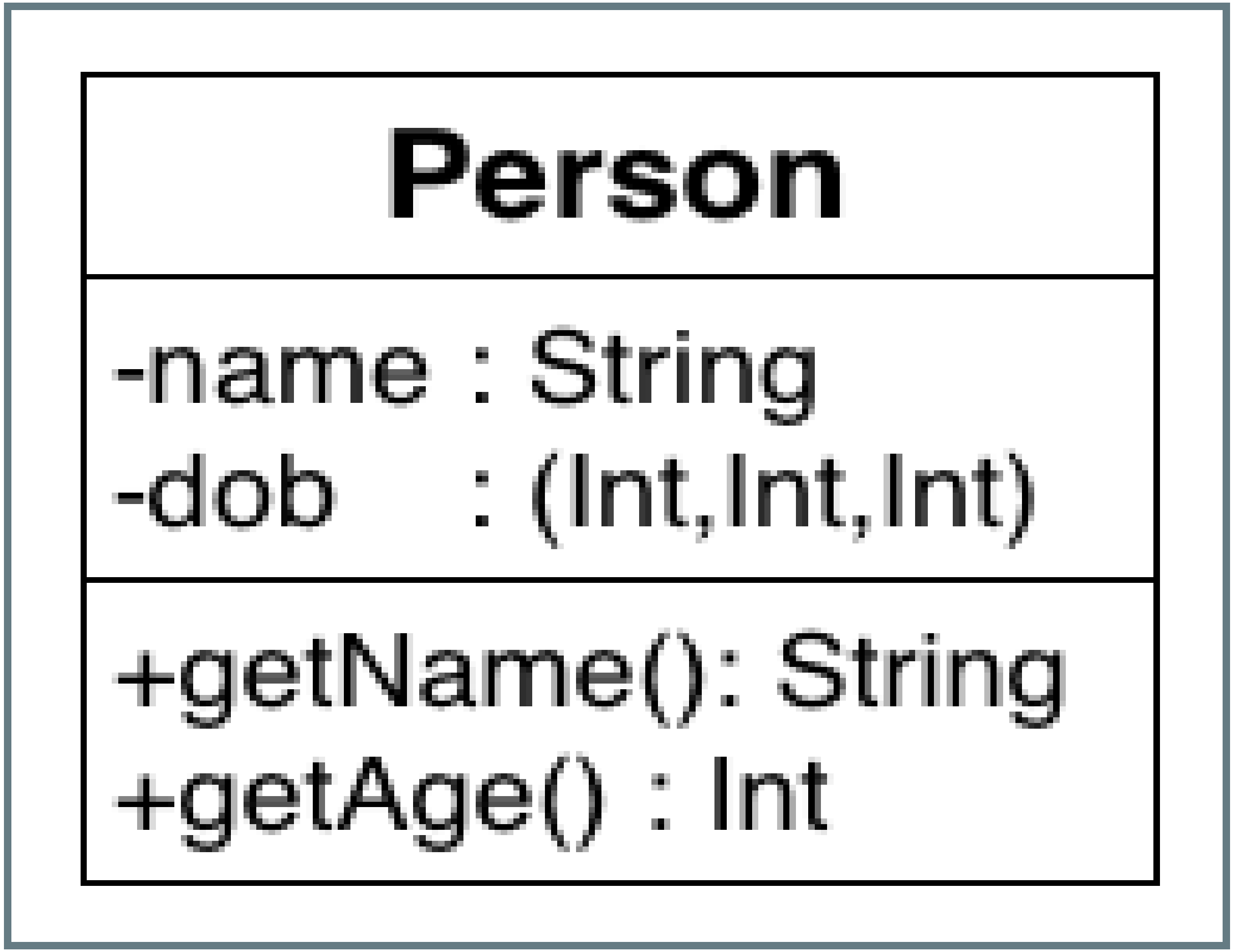
Operation

Operation

ENCAPSULATION CAN CONTROL ACCESS TO ATTRIBUTES AND METHODS

- Private (- in UML): limits access to within the class itself
- Public (+ in UML): allows external access

CLASS DIAGRAM PRIVATE AND PUBLIC ACCESS



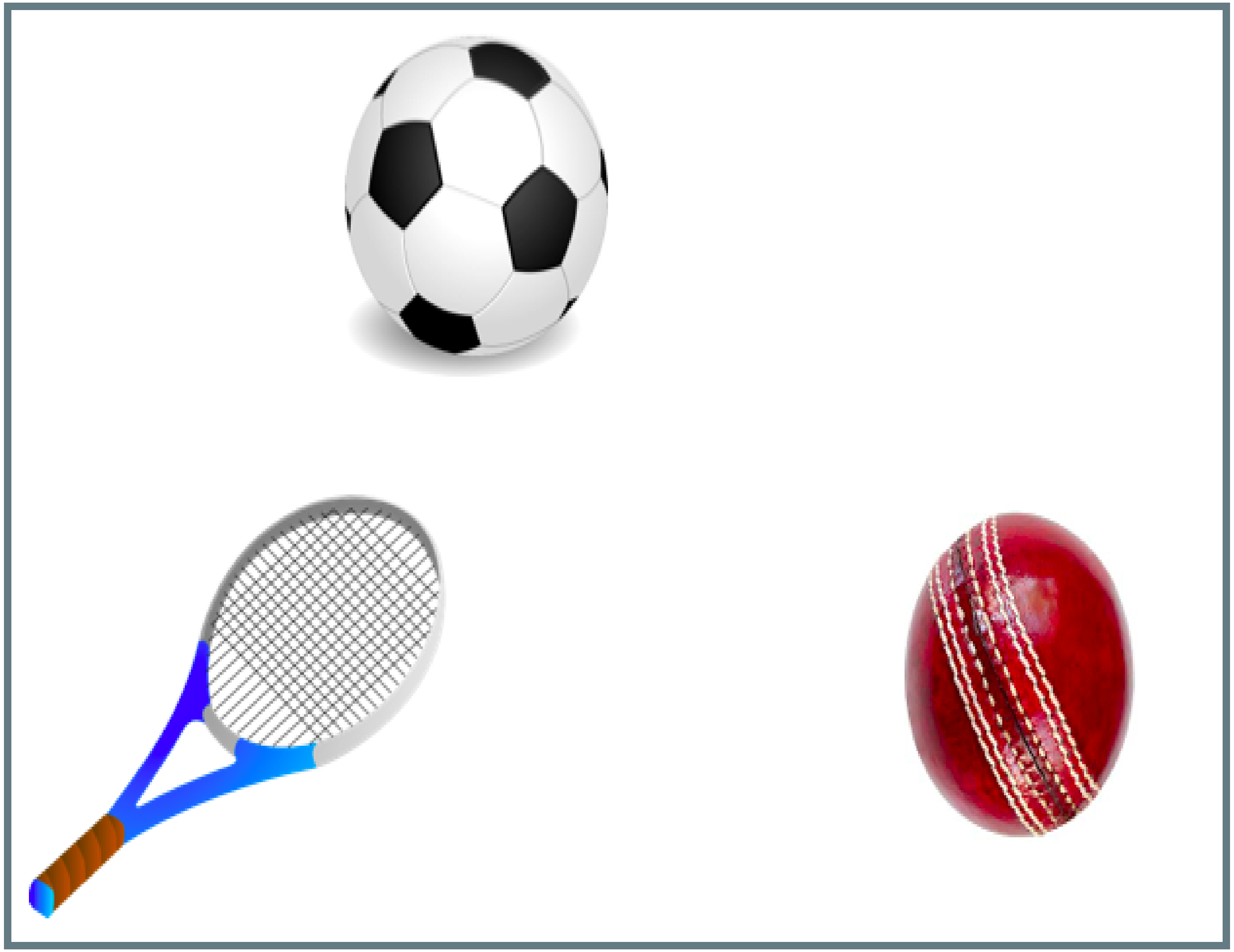
Person

-name : String
-dob : (Int,Int,Int)

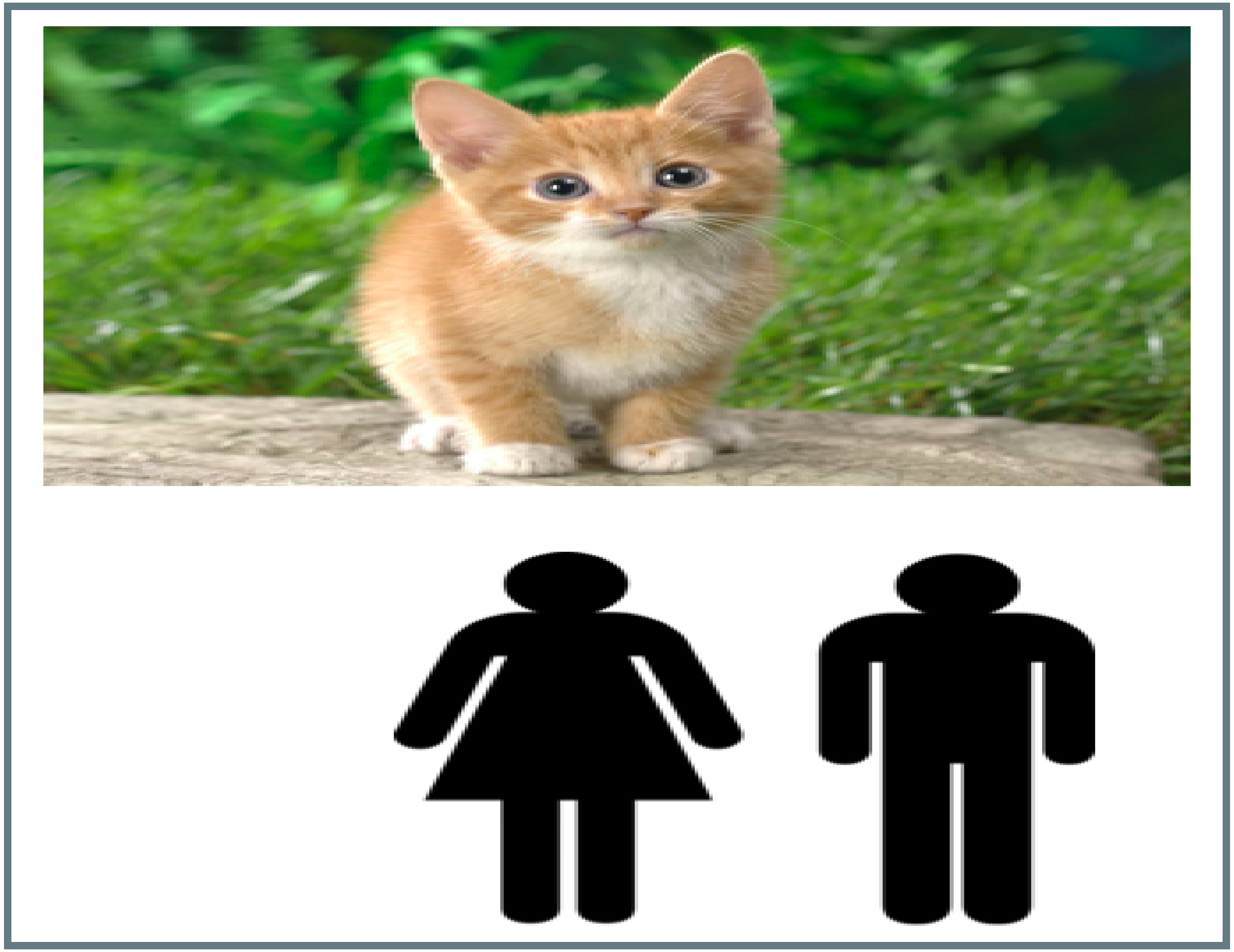
+getName(): String
+getAge() : Int

OBJECTS MAY SHARE CHARACTERISTICS

OBJECTS MAY SHARE CHARACTERISTICS



OBJECTS MAY SHARE CHARACTERISTICS



INHERITANCE

Passes knowledge "down" from one object to another

INHERITANCE

An object may "inherit" characteristics and behaviour from another

INHERITANCE

Creates a hierarchy of "inherited" characteristis and behaviours

INHERITANCE - EXAMPLE(S)

- All students are people
- All children are people
- All workers are people

INHERITANCE USING CLASSES

Remember classes are abstract representations of objects, so

INHERITANCE USING CLASSES

If a class B "inherits" functionality from a class A, we say

INHERITANCE USING CLASSES

- B is a subclass of A, and
- A is a superclass of B

INHERITANCE REFINES ENCAPSULATION

- Private (- in UML): limits access to within the class itself
- Public (+ in UML): allows external access
- Protected (# in UML): restricts access to subclasses

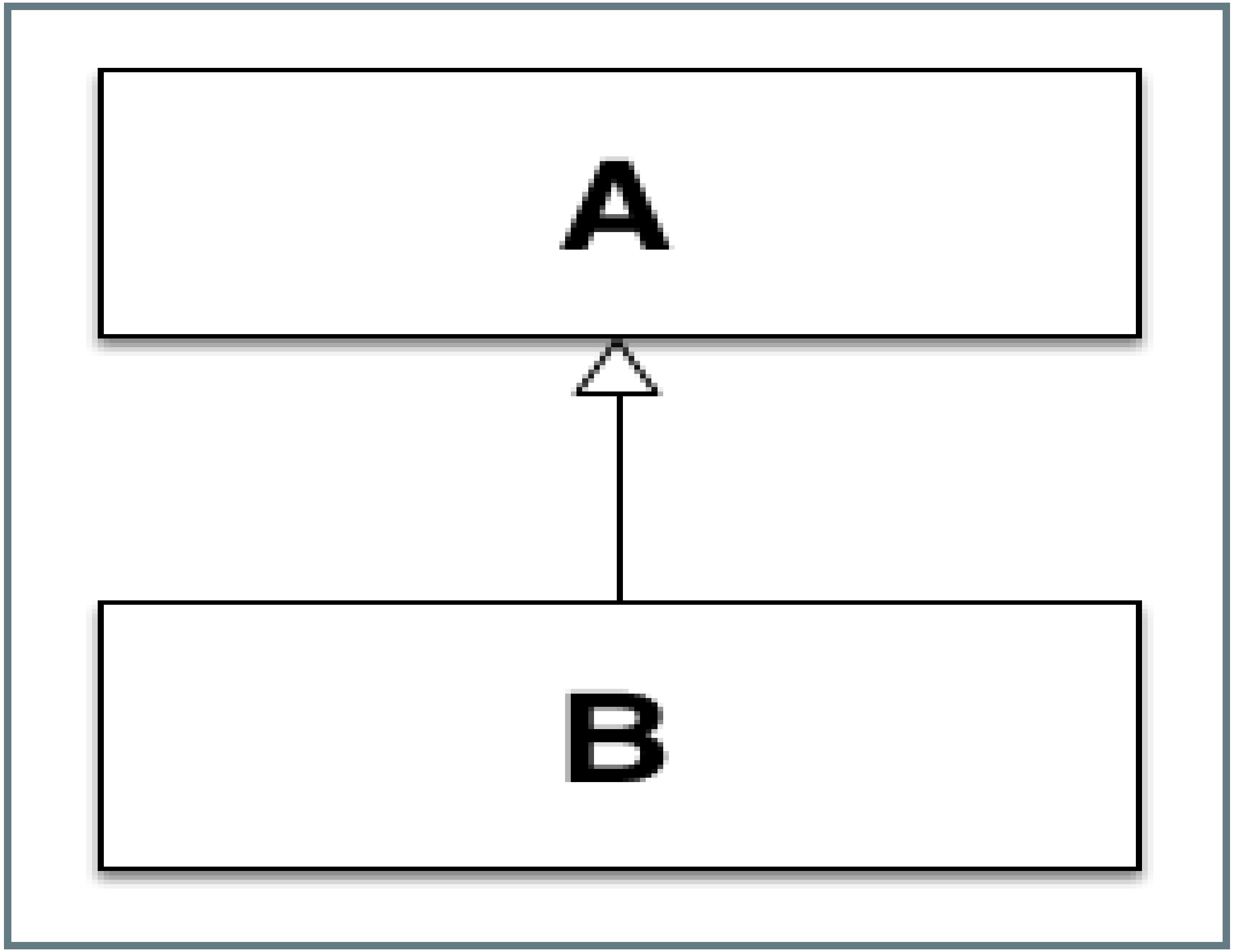
INHERITANCE

- Defines an "is a" relationship between subclass and superclass, e.g.
  - object B "is a" object A

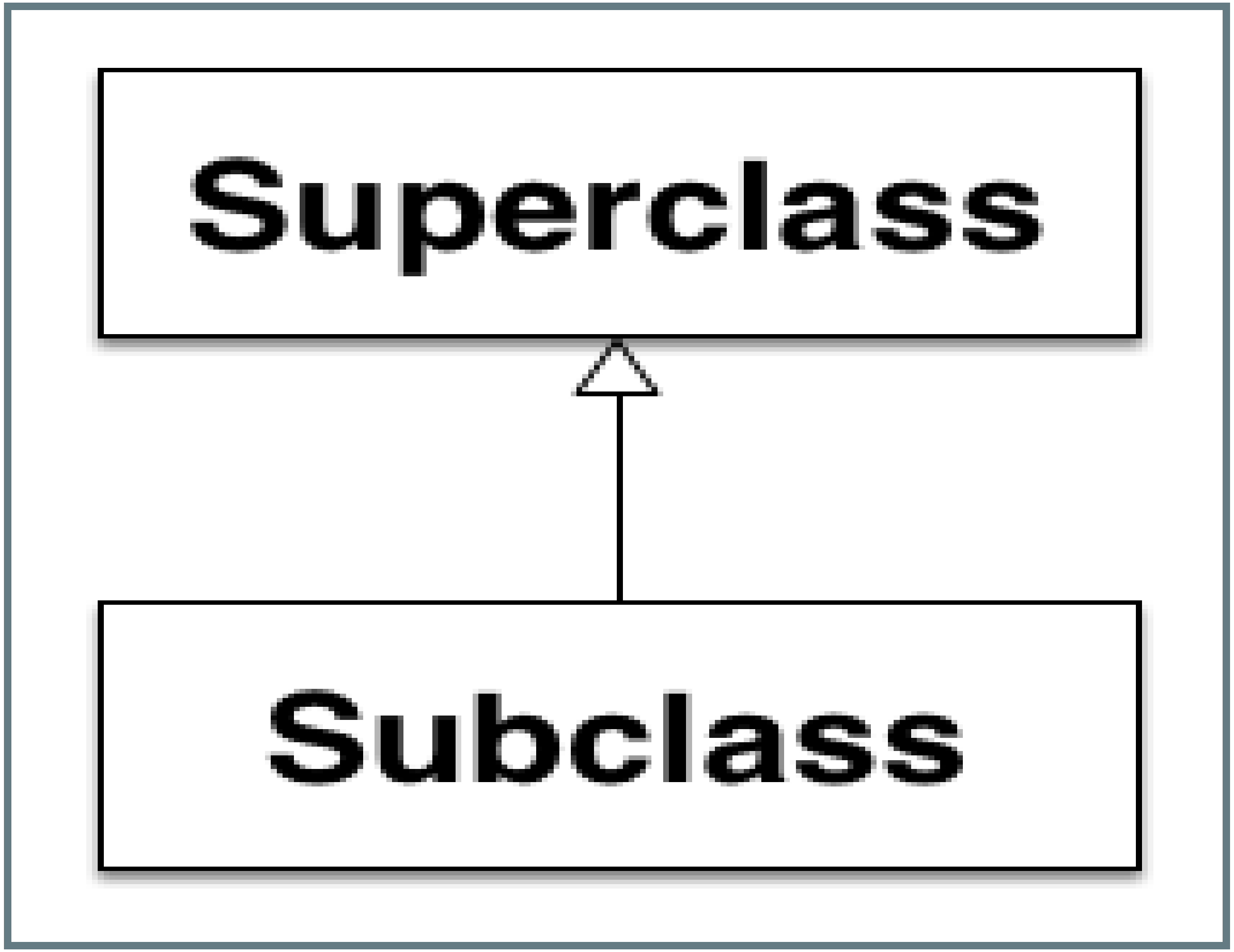
IS A RELATIONSHIP

- Is a relationship meaning a subclass inherits and extends functionality of some base (super)class

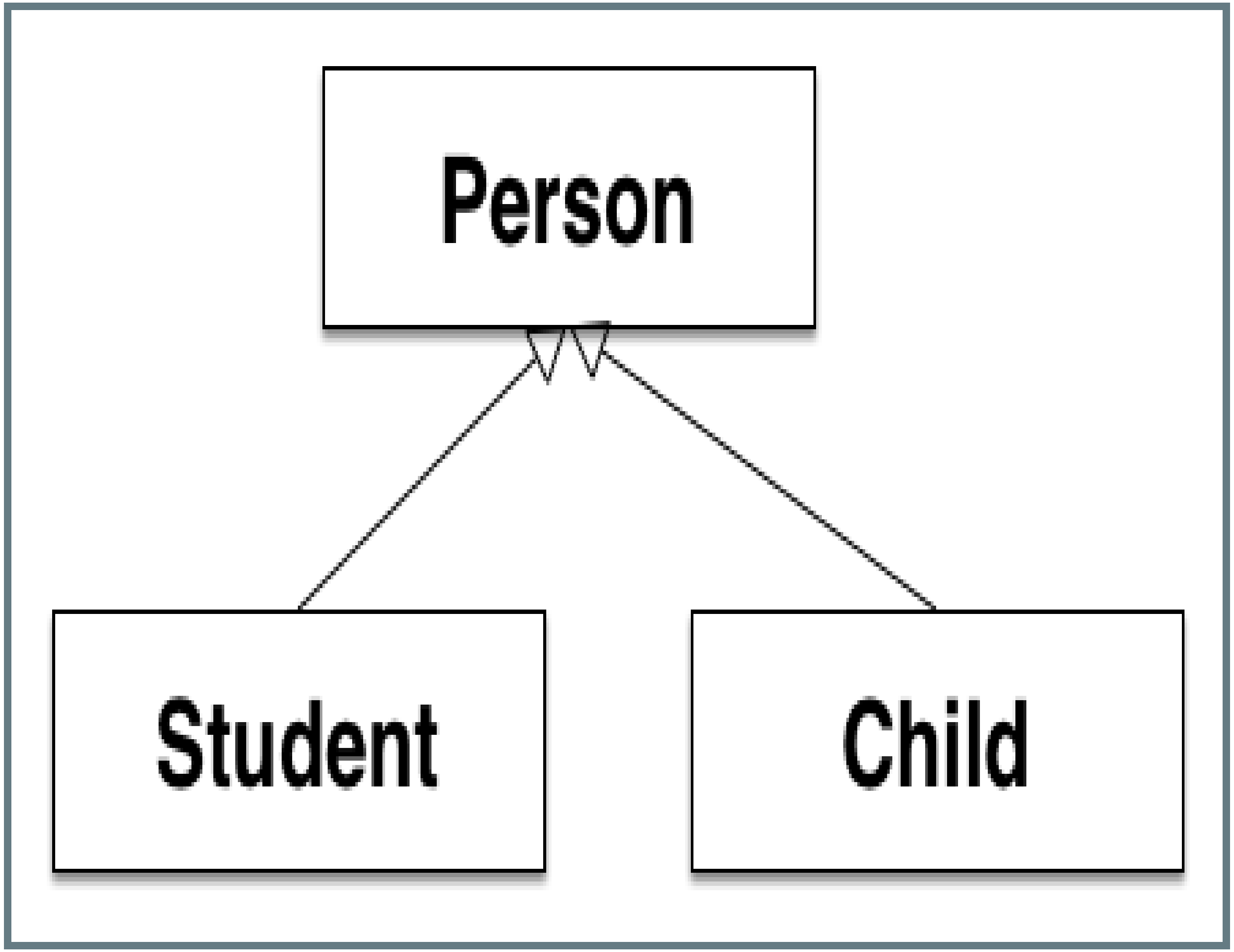
INHERITANCE CLASS DIAGRAM



INHERITANCE CLASS DIAGRAM



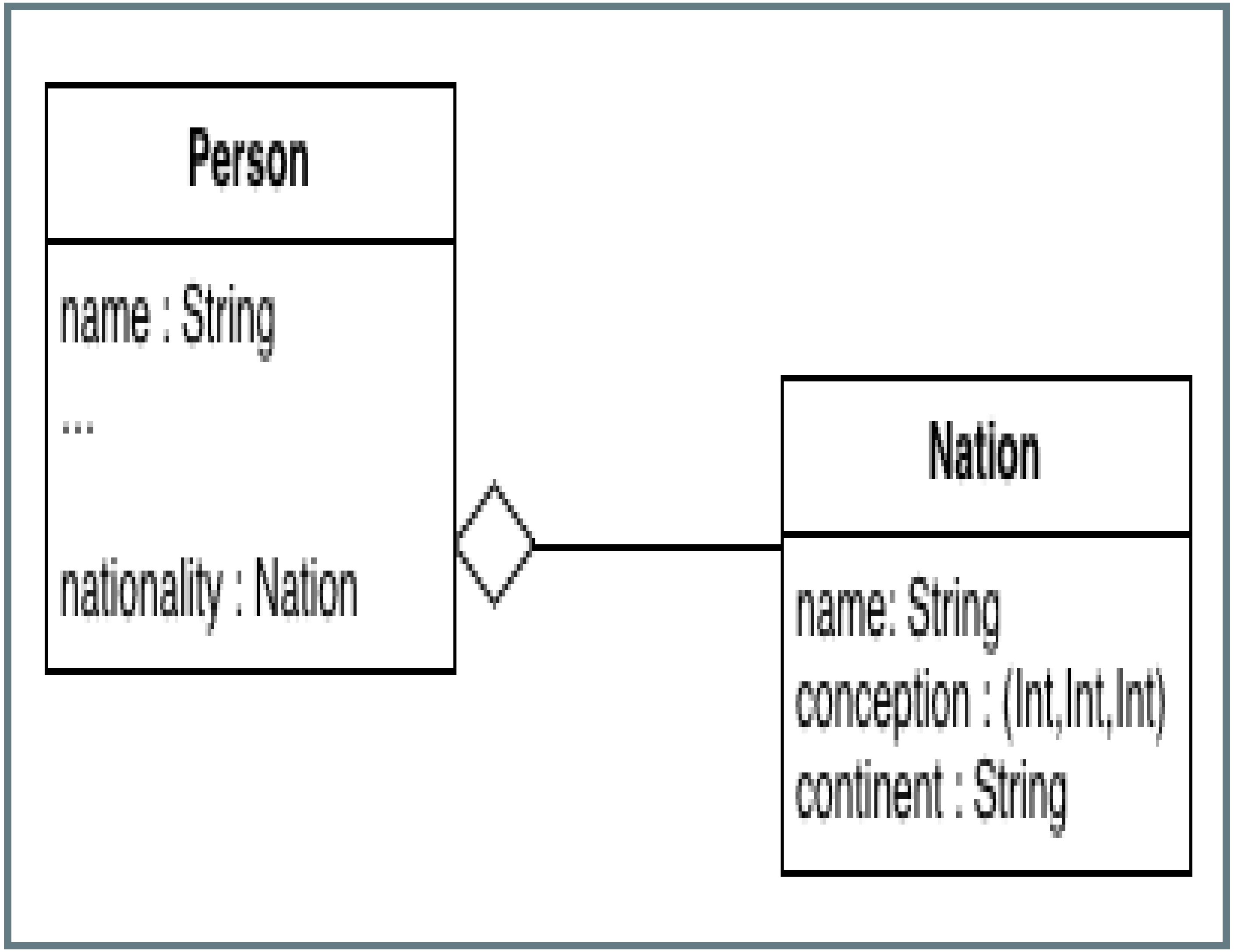
INHERITANCE EXAMPLE



HAS A RELATIONSHIP

- Is a relationship meaning a class is using (contains) another class

HSA A CLASS DIAGRAM



WHEN TO USE IS A OR HSA A?

- If an object is a type of a more general class, then use "is a"
- If an object has a particular "feature", then use "has a"

INHERITANCE POLYMORPHISM

- If B is subclass of A, then
    - We can use an object of type B, in any context that expects an A

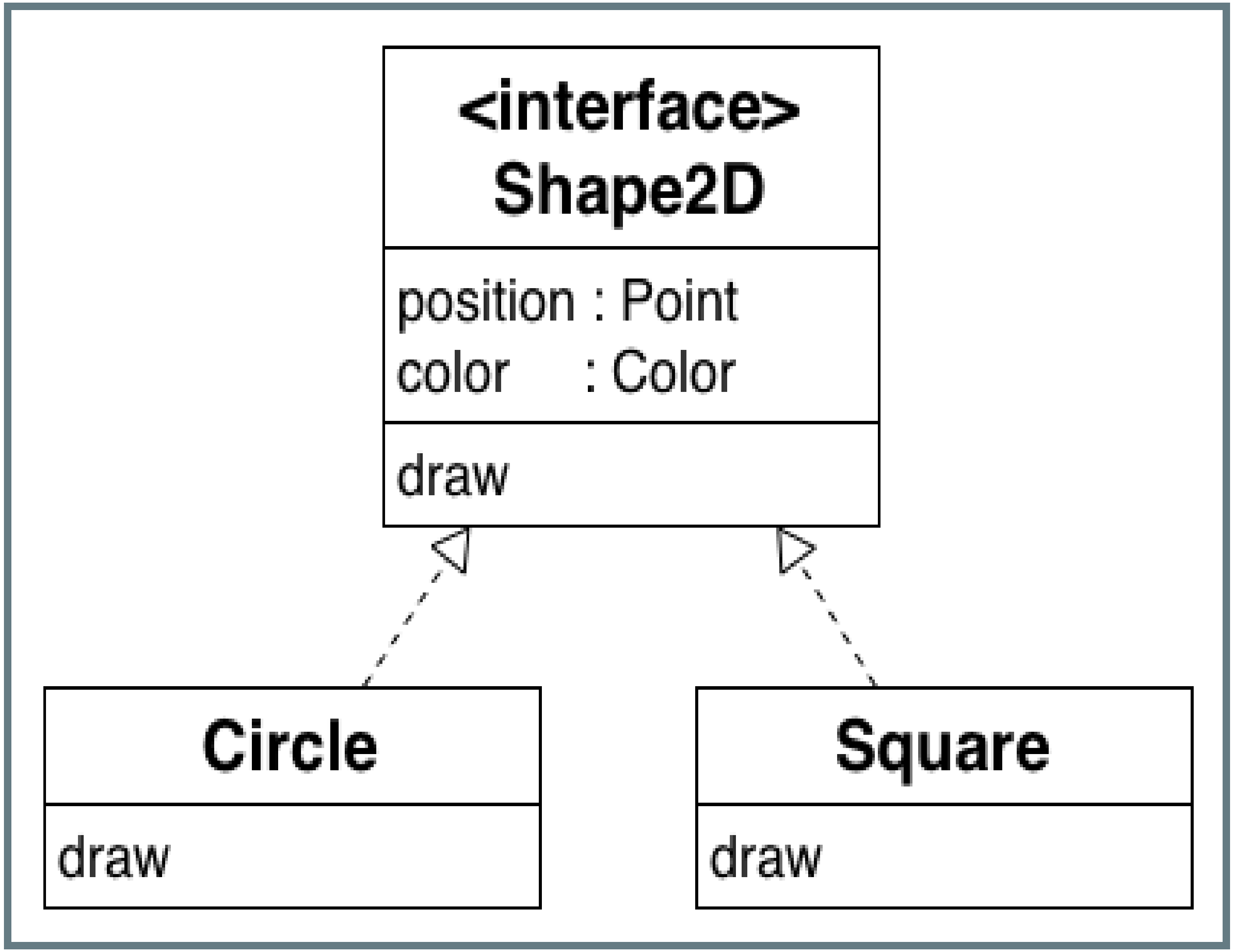
INHERITANCE POLYMORPHISM

- A subclass B can overide methods of a superclass A

2D SHAPES

- Shapes share many of the same attributes, e.g.
  - position
  - color
- Shapes all have a visual representation, however
  - visually they look different
  - the algorithm for drawing a circle is not the same as that for a square

INHERITANCE EXAMPLE



ABSTRACT CLASSES

- A class **without** one or more method implementations, e.g.
  - An abstract shape class might provide a method for drawing shapes, but only specific inherited shapes, e.g. square, can define the specific behaviour (algroithm)
- Java calls these **interfaces**
  - Sadly this is a very overloaded term!
- Scala calls these **traits**
  - Lots more on this later in the course

IN SUMMARY

- Object-oriented programming provides a powerful model for developing applications
- Objects provide for:
    - Encapsulationm, which helps enforces modularity
    - Inheritance, enabling the passing of knowlege, which in turn provide reuse
    - Inheritance Polymorphim, which provides the abilty to specialize "common" functionality