7SENG011W Object Oriented Programming

Object Relationships, Class Relationships: Generalisation and Inheritance

Dr Francesco Tusa

Readings

Books

- Head First Java
  - Chapter 7: Better Living in Objectville: Inheritance and Polymorphism
- UML Distilled
  - Chapter 3: Class Diagrams
- Object-Oriented Thought Process
  - Chapter 7: Mastering Inheritance and Composition

Online

- IBM: UML Class Diagrams
- The Java Tutorials: Inheritance

Outline

- Object Relationships
  - Summary and Recap on UML Class Diagrams
  - Association, Aggregation and Composition
- Class Relationships: Generalisation and Inheritance
  - Definitions
  - Class Diagrams
  - Code Implementation and the ‘super’ keyword
  - Protected access modifier
  - Going Further with Inheritance

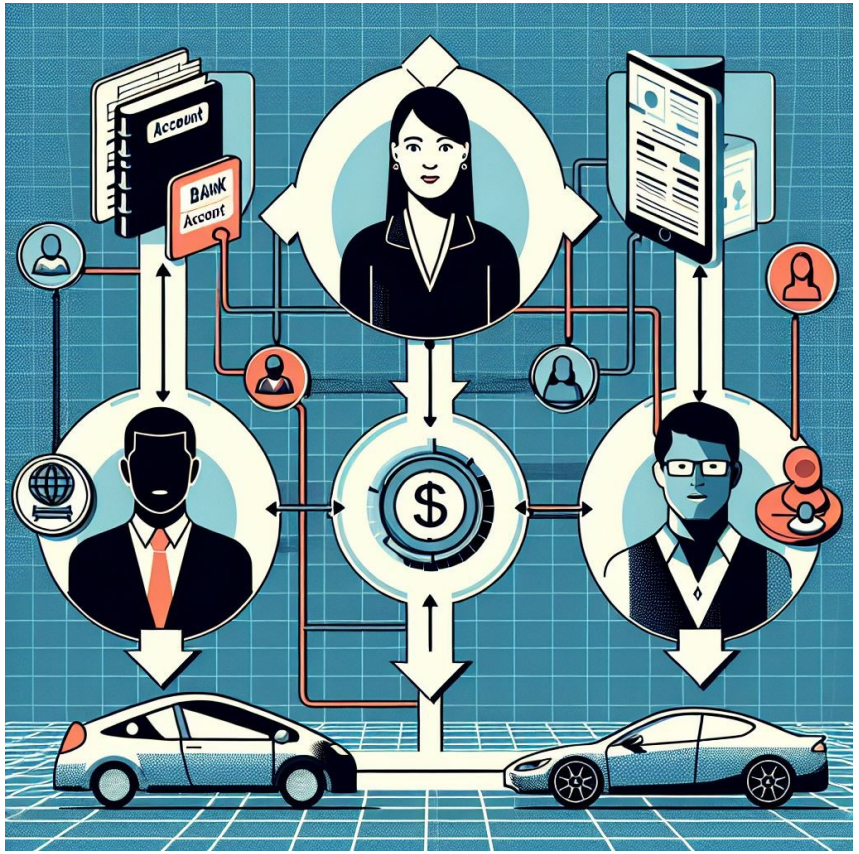
Outline

- Object Relationships
  - **Summary and Recap on UML Class Diagrams**
  - Association, Aggregation and Composition
- Class Relationships: Generalisation and Inheritance
  - Definitions
  - Class Diagrams
  - Code Implementation and the ‘super’ keyword
  - Protected access modifier
  - Going Further with Inheritance

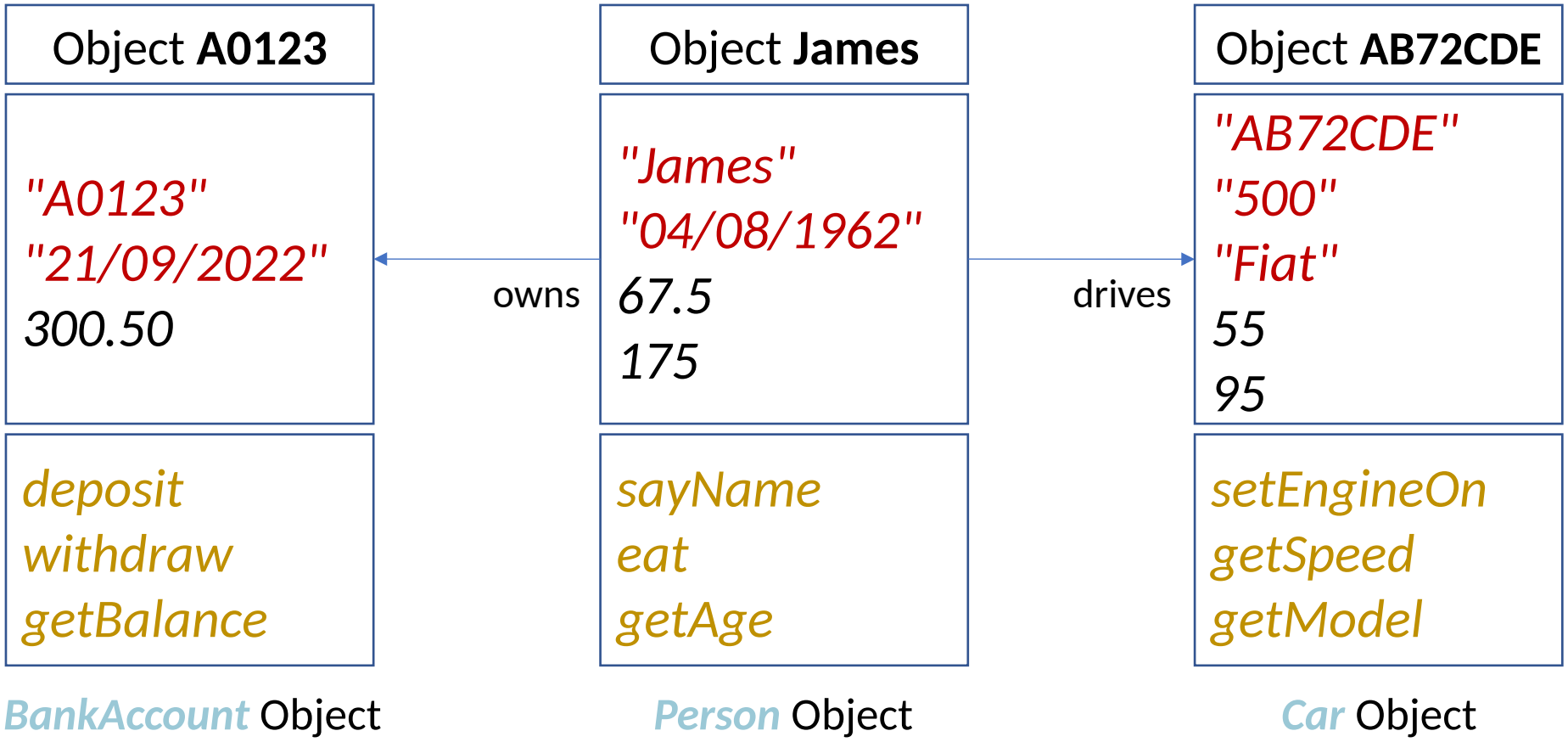
Object Relationships

- **Objects** are (abstracted) representations of **real-world entities** within our programs
- In the real world, one **object** can have a type of **relationship** with another **object**
- These **relationships** are also **modelled** in our programs

- What are the consequences of doing this?



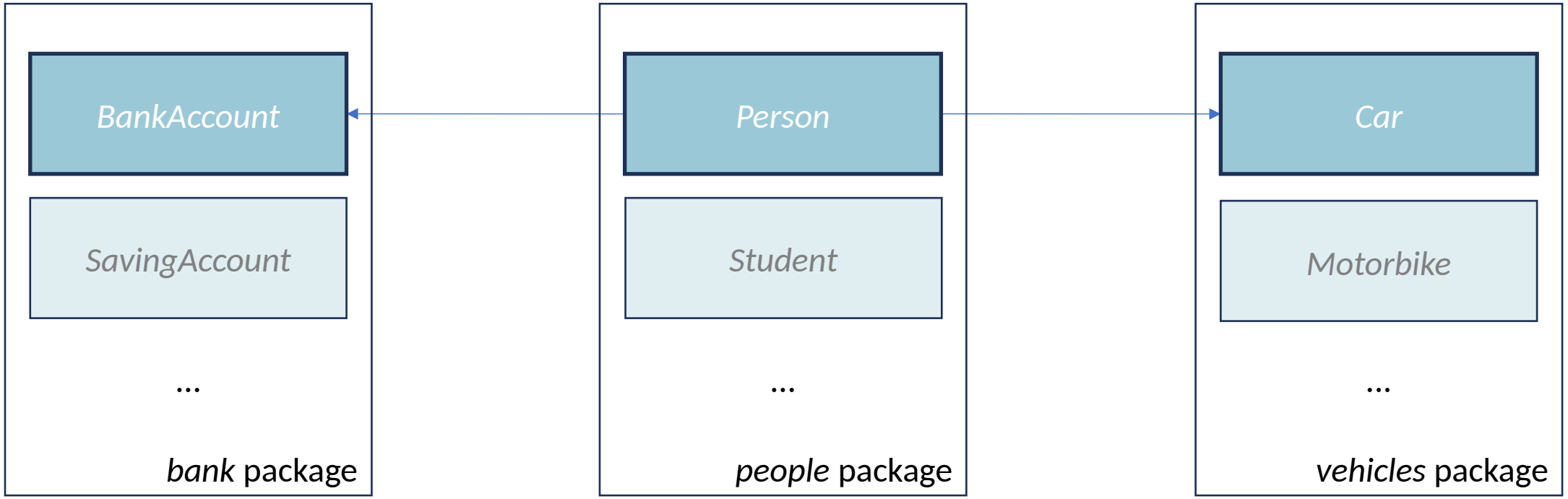
Object Relationships: *Modularity*



Object Relationships: *Modularity*

- The code of the classes from which objects are created are **separate modules** of a program
- Can be *developed*, *tested*, *extended* and *maintained* **independently**
- The code of a class can be **reused** in **several programs**

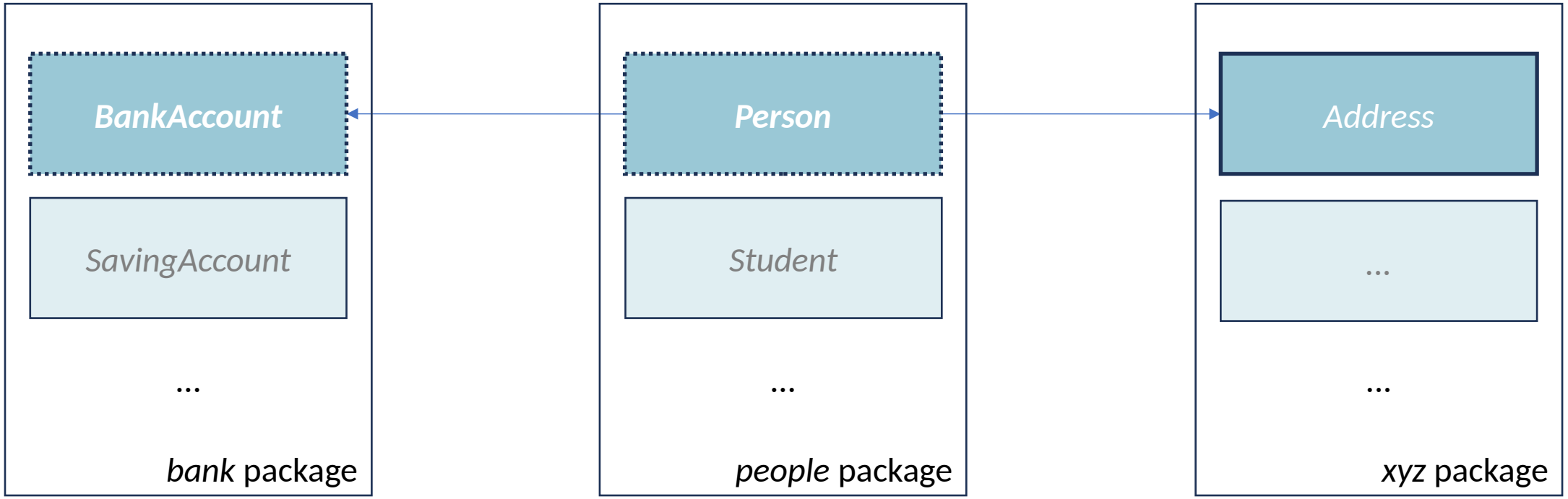
Object Relationships: *Modularity*



A program that calculates the **account balance** of **people** driving different **car** models

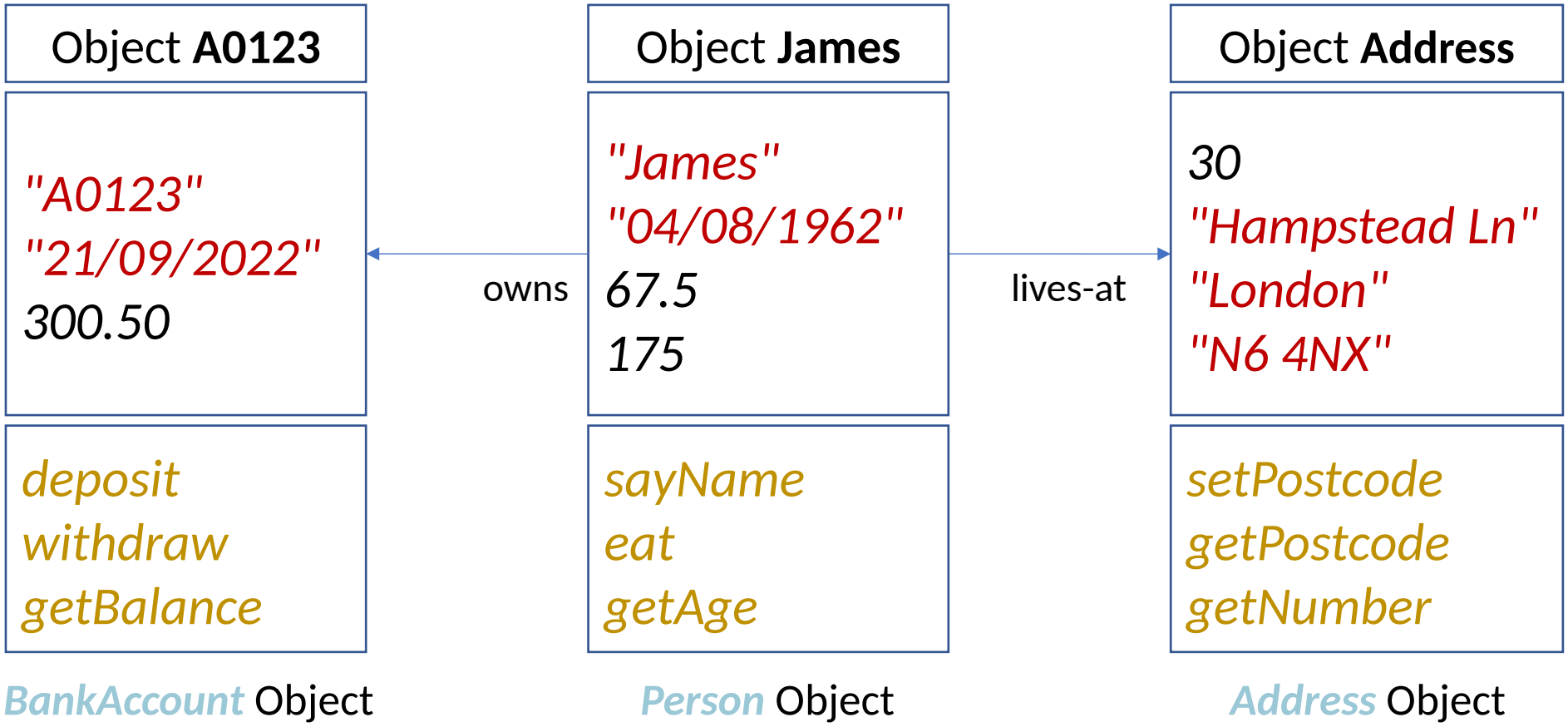
Object Relationships: *Modularity*

The **code** of the *BankAccount* and *Person* classes can be **reused**

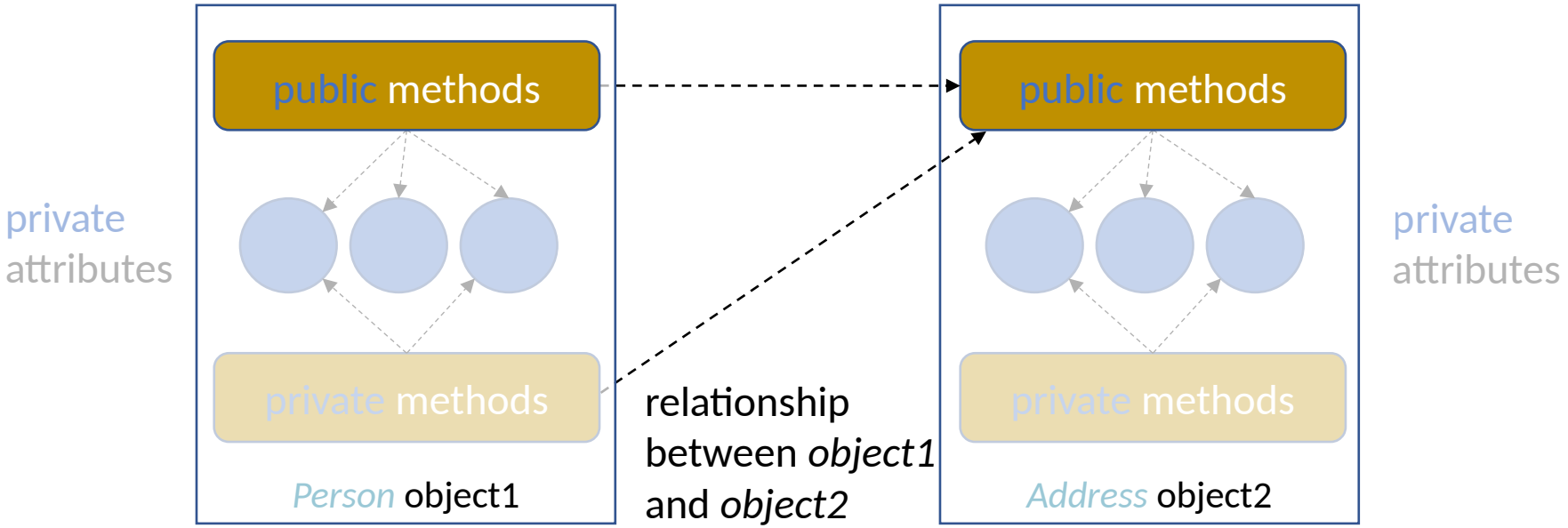


A program that reveals **people**'s **address** based on their **bank account's balance**

Object Relationships: *Modularity*



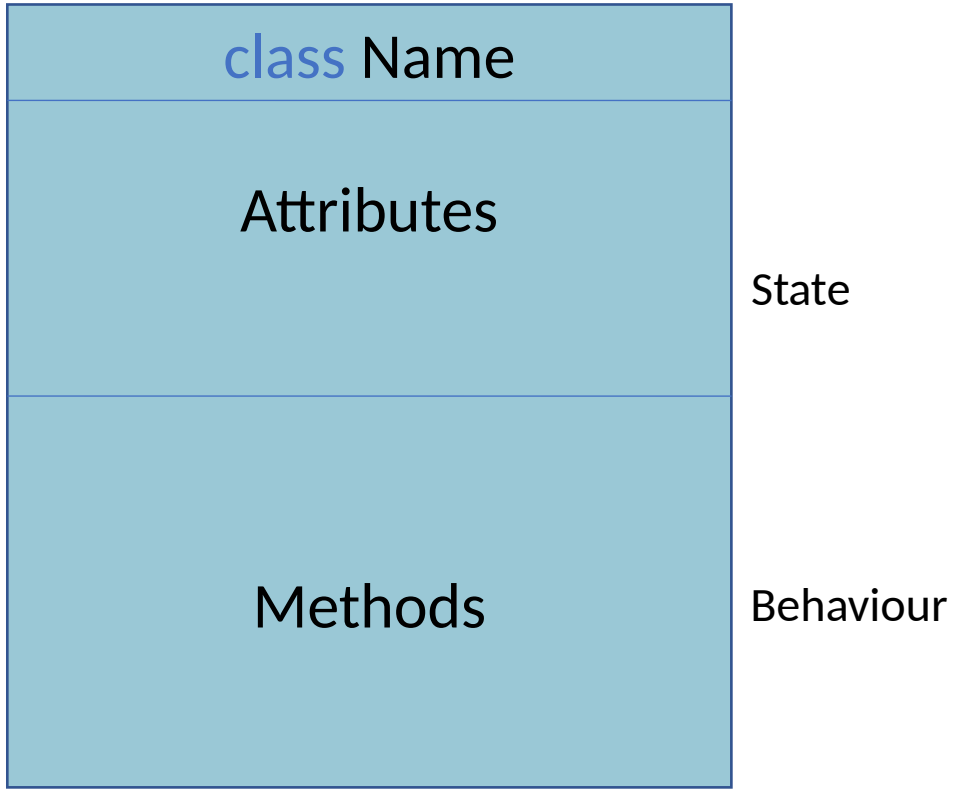
Object Relationships: *Encapsulation*



- public *methods* define the **interface** an object exposes to objects of other classes
- private *members* are **hidden** and **not accessible** by objects of other classes
- minimises **errors** and **anomalies** and makes the code **flexible** and maintainable

Class Diagrams: Recap

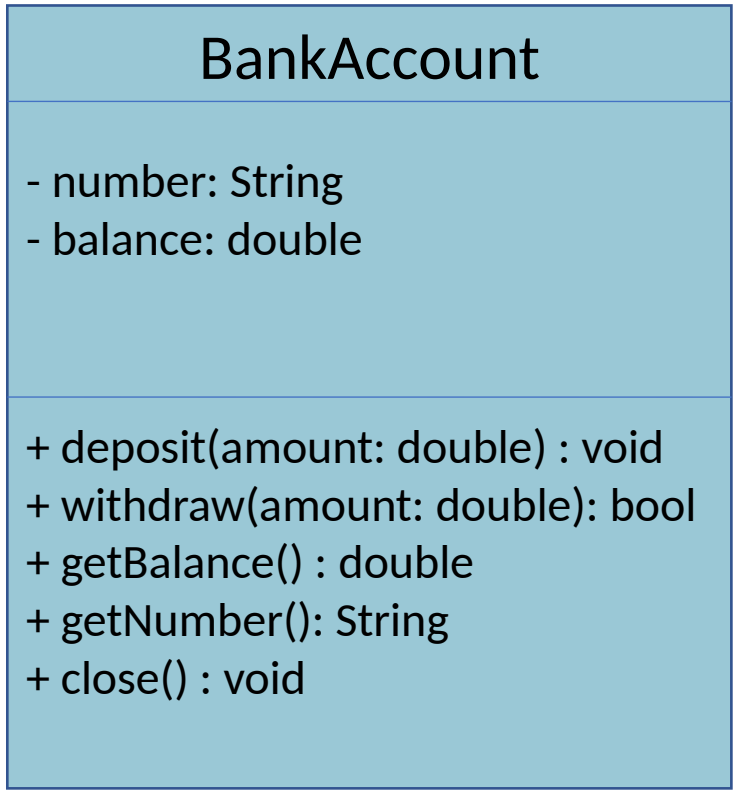
UML (Unified Modelling Language)



Class Diagrams: Recap

– symbol: private access modifier

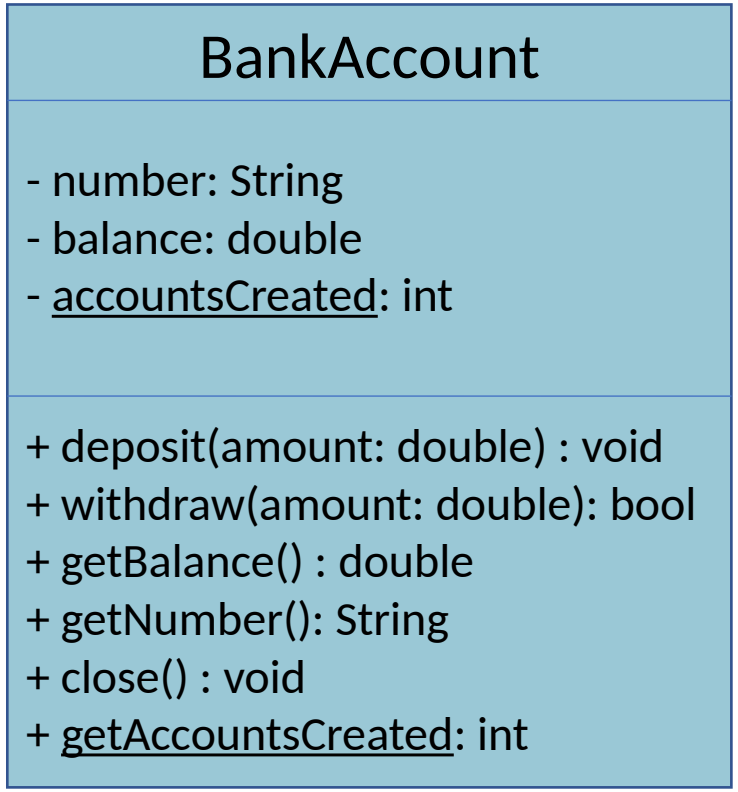
+ symbol: public access modifier



State

Behaviour

Class Diagrams: static members



State

Behaviour

underlined defines static members

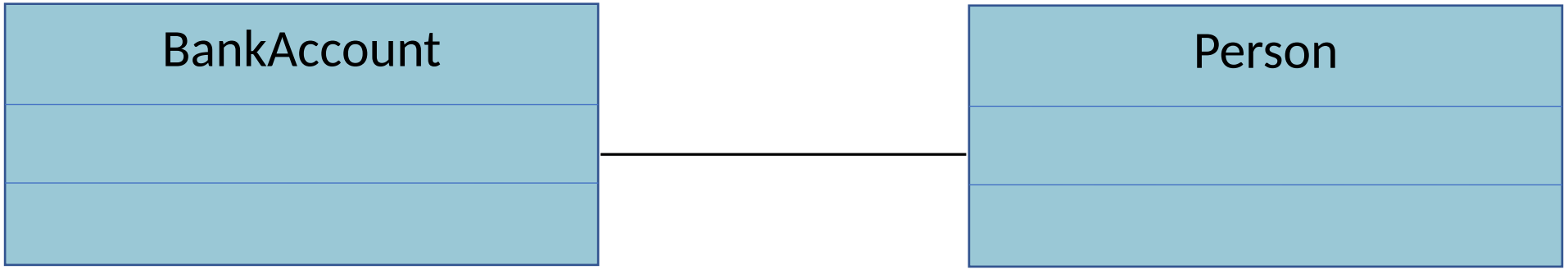
Class Diagrams: Object Relationships

- Let’s explore how **Class Diagrams** can be used to **model** and **represent object relationships**

Outline

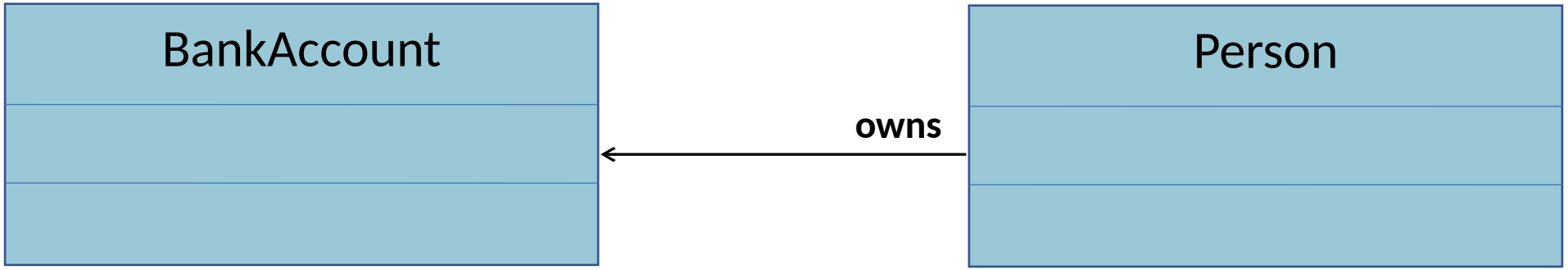
- Object Relationships
  - Summary and Recap on UML Class Diagrams
  - **Association**, Aggregation and Composition
- Class Relationships: Generalisation and Inheritance
  - Definitions
  - Class Diagrams
  - Code Implementation and the ‘super’ keyword
  - Protected access modifier
  - Going Further with Inheritance

Class Diagrams: Association



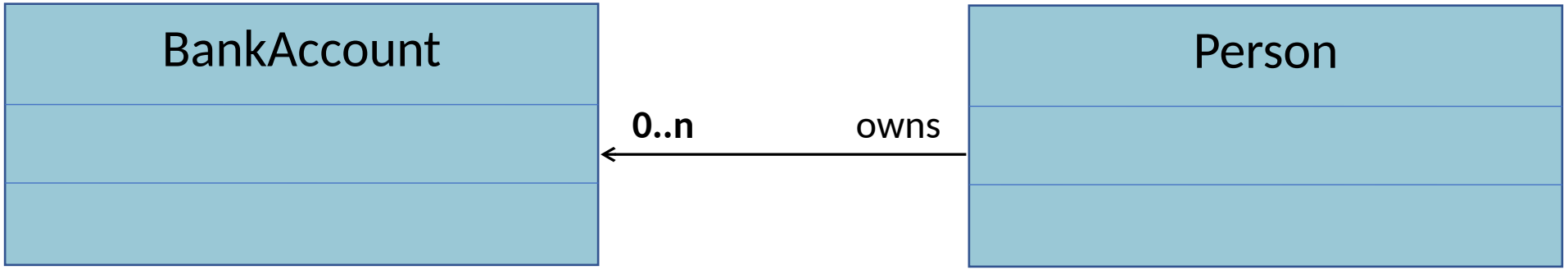
- Structural **relationship** between **peers**
- **Objects** of both classes are conceptually at the **same level**
- **No one** is **more important** than the **other**

Class Diagrams: *Unidirectional* Association



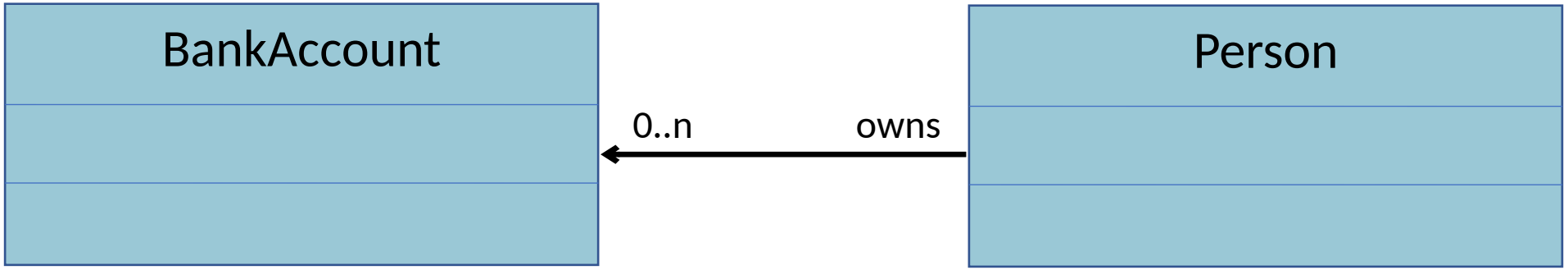
association name and **direction**

Class Diagrams: *Unidirectional* Association



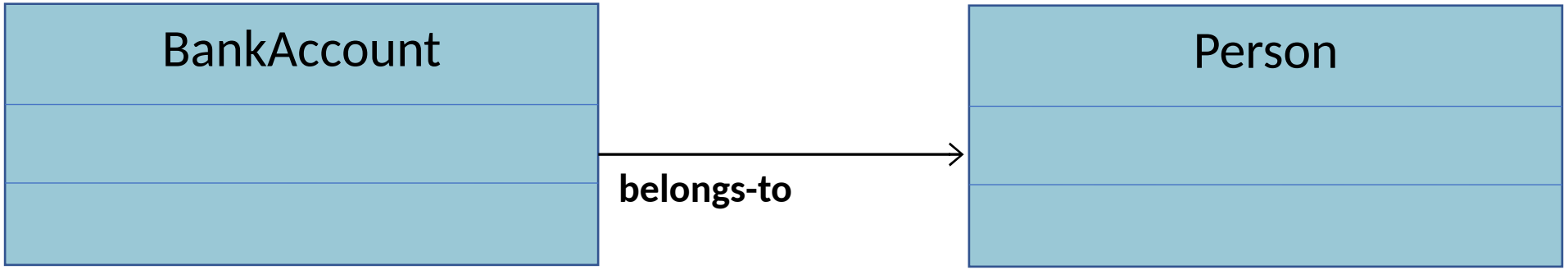
association multiplicity

Class Diagrams: *Unidirectional* Association



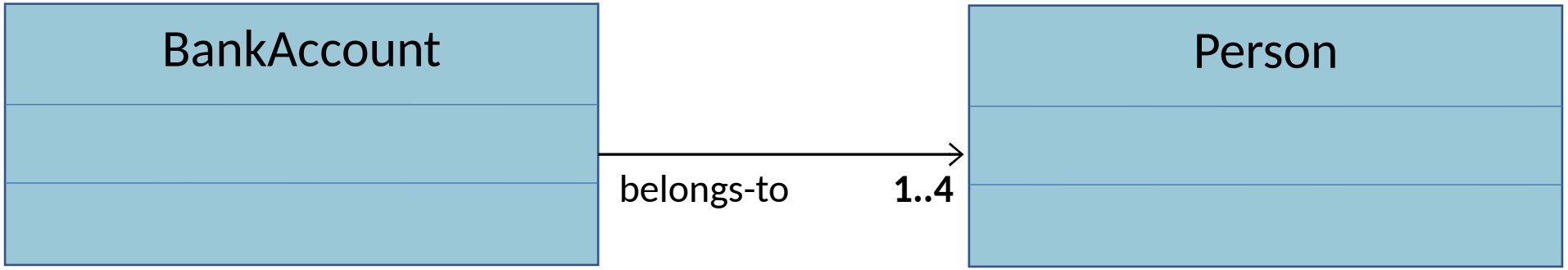
- The **direction** determines how the information can be **navigated**
- What *bank accounts* does *John Doe* **own**?

Class Diagrams: *Unidirectional* Association



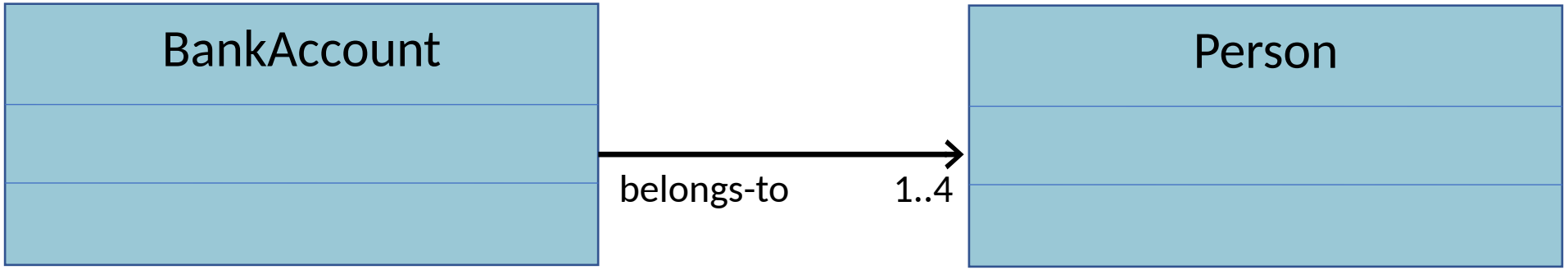
association name

Class Diagrams: *Unidirectional* Association



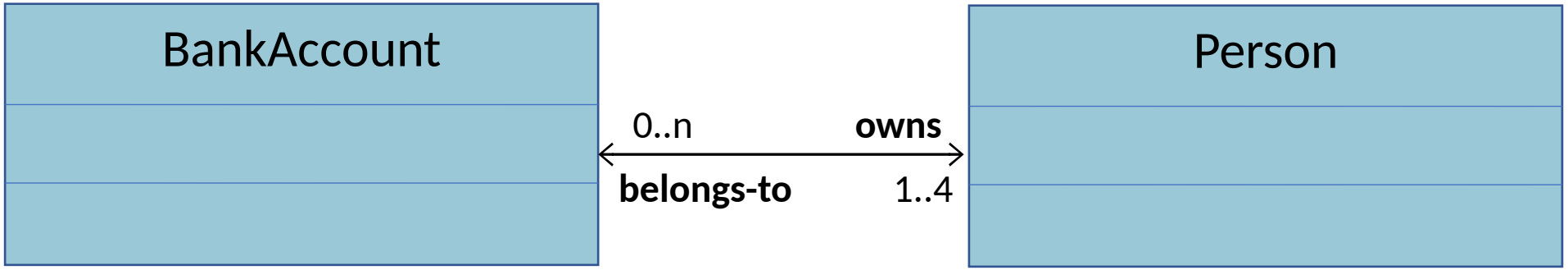
association multiplicity

Class Diagrams: *Unidirectional* Association



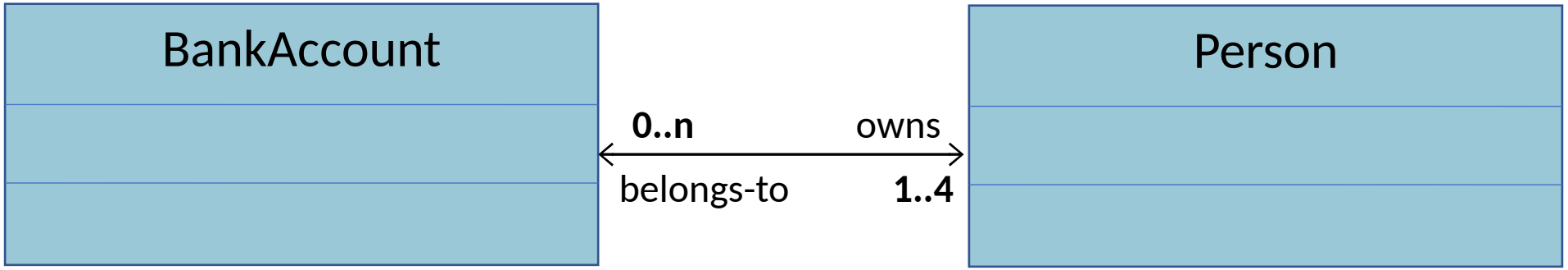
- The **direction** determines how the information can be **navigated**
- What *persons* does the *bank account AB123* **belong to**?

Class Diagrams: *Bidirectional* Association



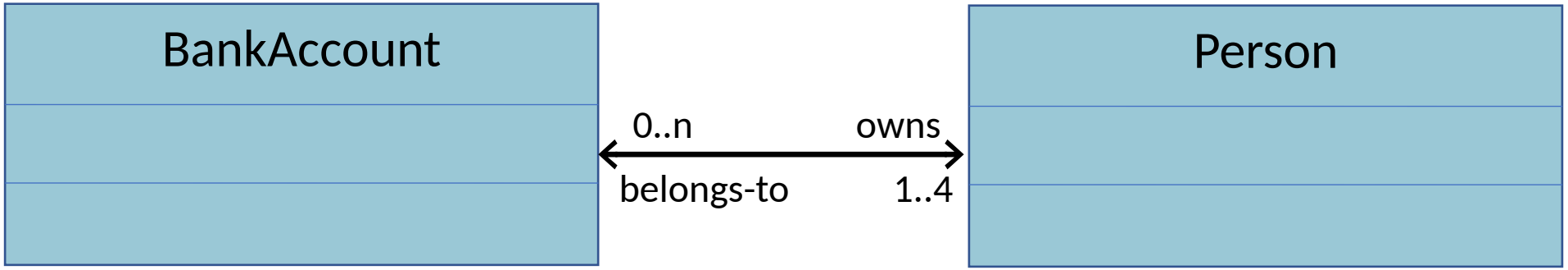
association name

Class Diagrams: *Bidirectional* Association



association multiplicity

Class Diagrams: *Bidirectional* Association



- **Bidirectional navigation**
- What *bank accounts* does *John Doe* **own**?
- What *persons* does the *bank account AB123* **belong to**?

Class Diagrams: *Association* Example 1

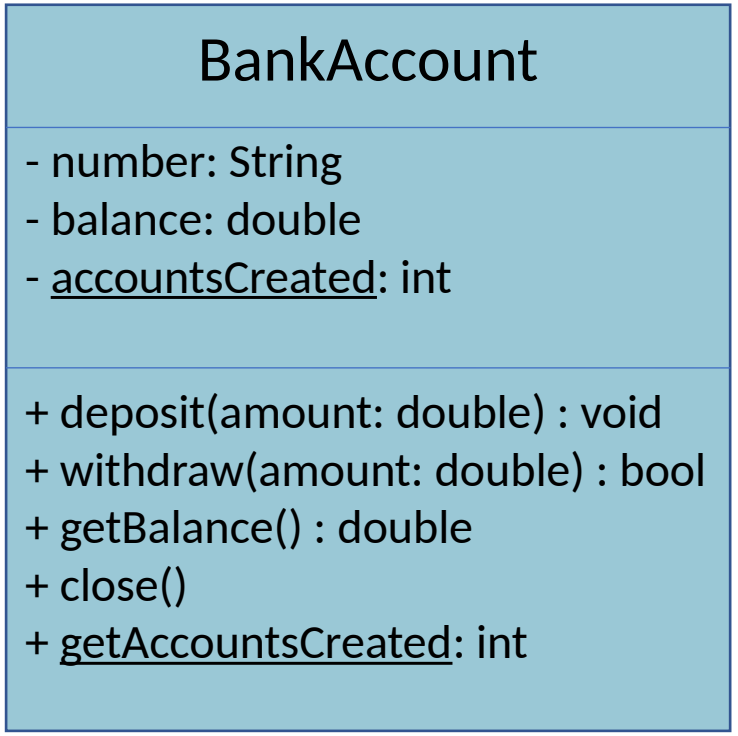
BankAccount
- number: String - balance: double - accountsCreated: int
+ deposit(amount: double) : void + withdraw(amount: double) : bool + getBalance() : double + close() + getAccountsCreated: int

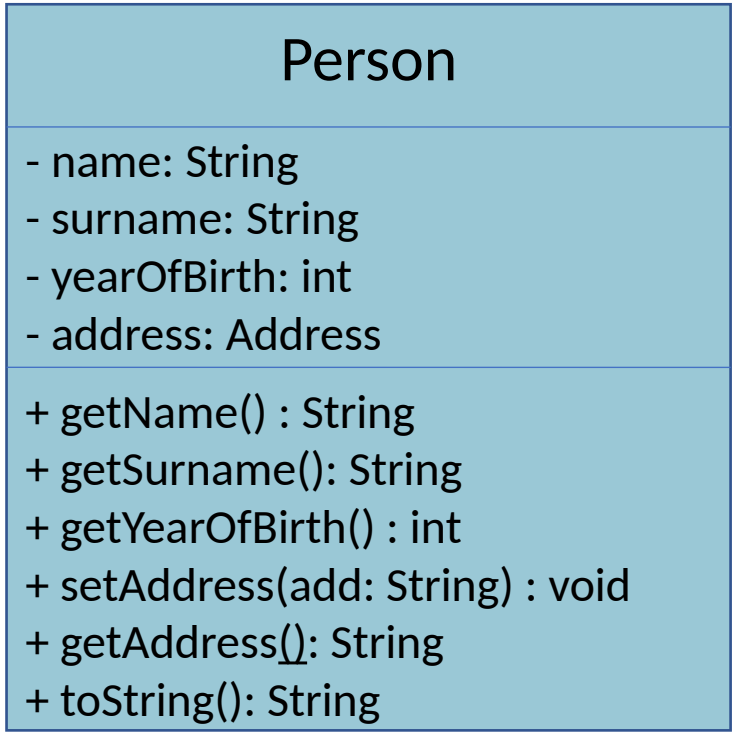
Person
- name: String - surname: String - yearOfBirth: int - address: Address
+ getName() : String + getSurname(): String + getYearOfBirth() : int + setAddress(add: String) : void + getAddress(): String + toString(): String

Association: *one BankAccount belongs-to one Person*

constructors have not been represented in this diagram

Class Diagrams: *Association* Example 1

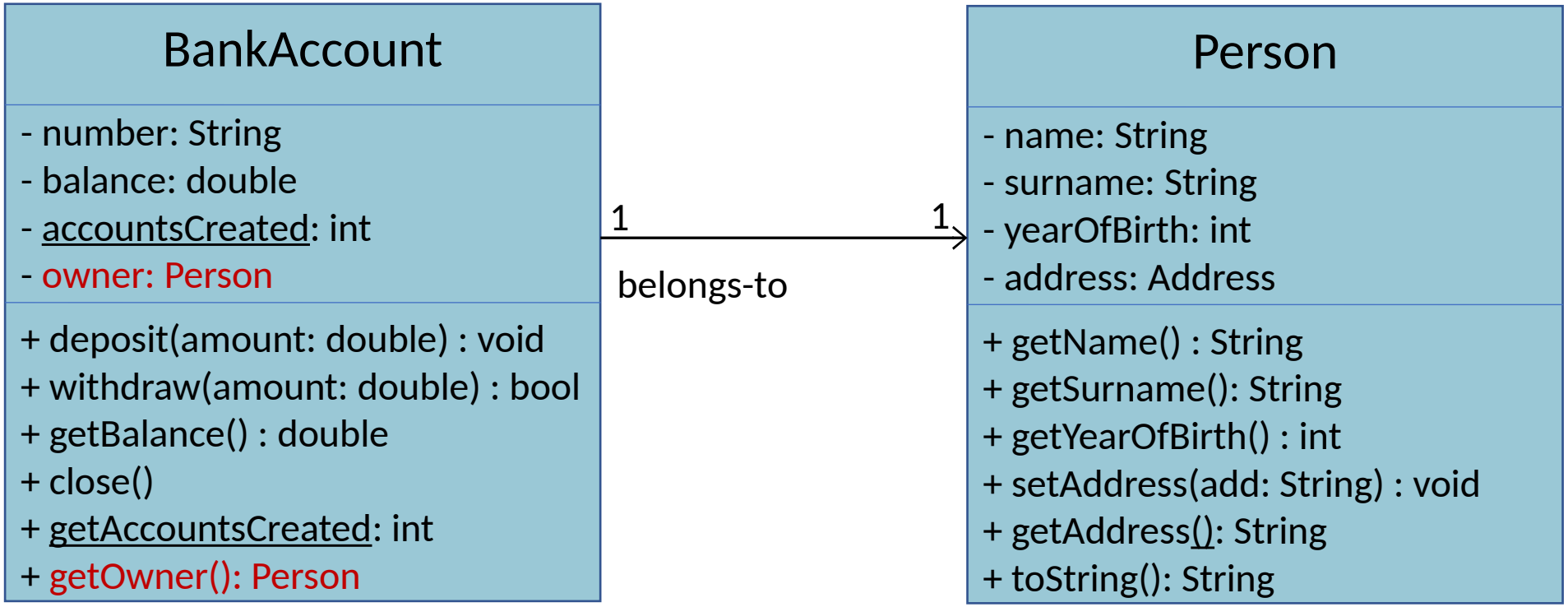




Association: *one BankAccount belongs-to one Person*

What do we need to add to the diagram to implement the association?

Class Diagrams: *Association* Example 1



Association: *one BankAccount belongs-to one Person*

A reference variable to the (one) *owner Person* object

Implementing an *Association*: Example 1

```
class BankAccount
{
    private String number;
    private double balance;

    private static int accountsCreated = 0;

    public BankAccount(String num, double bal) {
        number = num;
        balance = bal;

    }

    // other methods of BankAccount




}
```

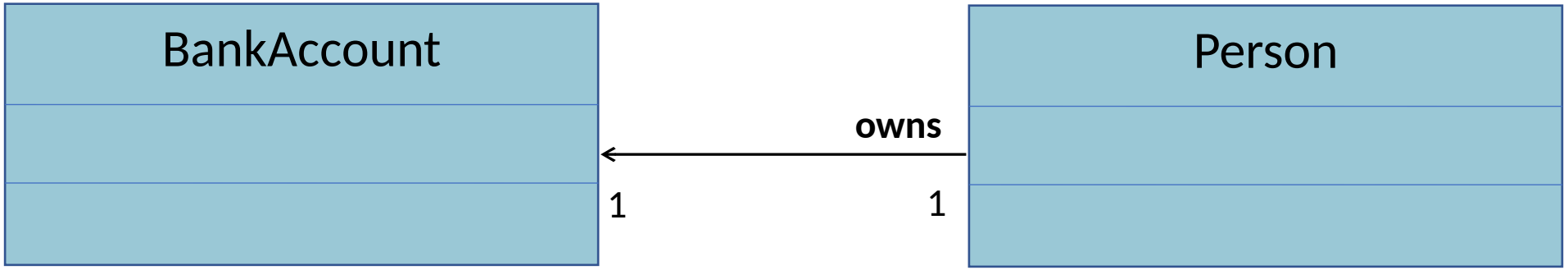
Implementing an *Association*: Example 1

```
class BankAccount
{
    private String number;
    private double balance;

    private static int accountsCreated = 0;

    public BankAccount(String num, double bal) {
        number = num;
        balance = bal;

    }

    // other methods of BankAccount



}
```

```
class BankAccount
{
    private String number;
    private double balance;
    private Person owner; // implements the association
    private static int accountsCreated = 0;

    public BankAccount(String num, double bal, Person p) {
        number = num;
        balance = bal;
        owner = p;
    }

    // other methods of BankAccount

    public String getOwner() {
        return owner;

}
```

existing (external) object passed to the constructor

Implementing an *Association*: Example 2



Association: *one Person owns one BankAccount*

How do we implement it?

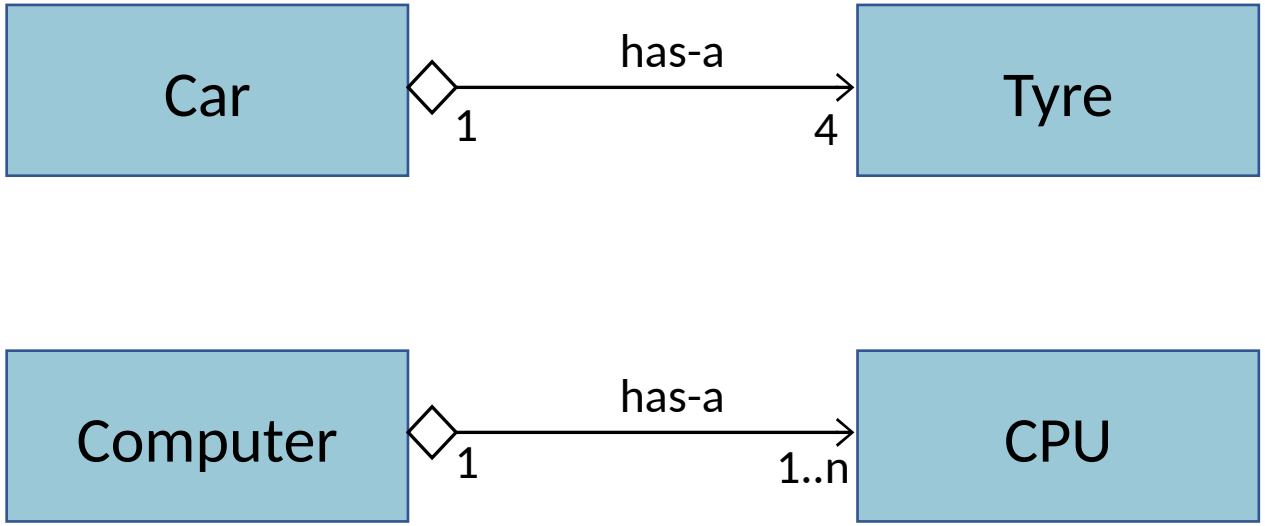
Implementing an *Association*: Example 2

```
class Person
{
    private String name;
    private String surname;
    private int yearOfBirth;
    private Address address;
    private BankAccount account;

    public Person(String n, String s, int yob, BankAccount acc)
    {
        ...
        account = acc;
    }

    // other methods...

    public String getAccount() {
        return account;
    }
}
```

Outline

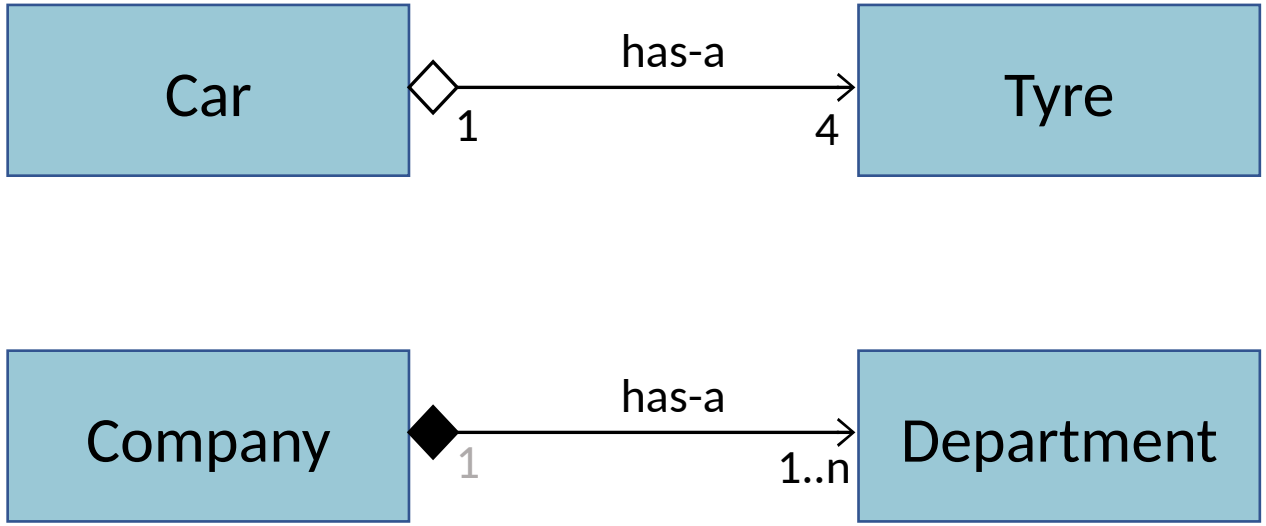
- Object Relationships
  - Summary and Recap on UML Class Diagrams
  - Association, **Aggregation** and **Composition**
- Class Relationships: Generalisation and Inheritance
  - Definitions
  - Class Diagrams
  - Code Implementation and the ‘super’ keyword
  - Protected access modifier
  - Going Further with Inheritance

Aggregation: Definition and Examples



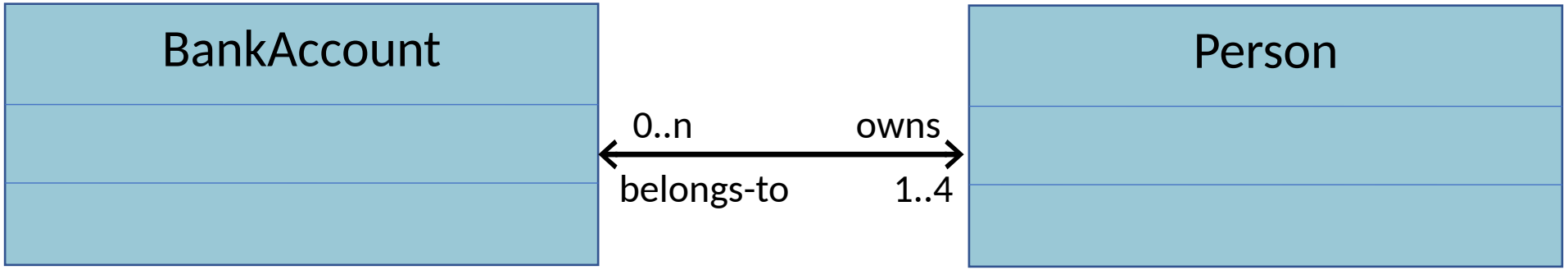
- **Aggregation** models the **has-a** relationship
- **One** object is **more important** than the **other**
- **One** object represents a *larger thing*—the **whole**, consisting of *smaller things*—the **parts**
- **Direction** always **from** the **whole to** the **parts**

Aggregation: Question



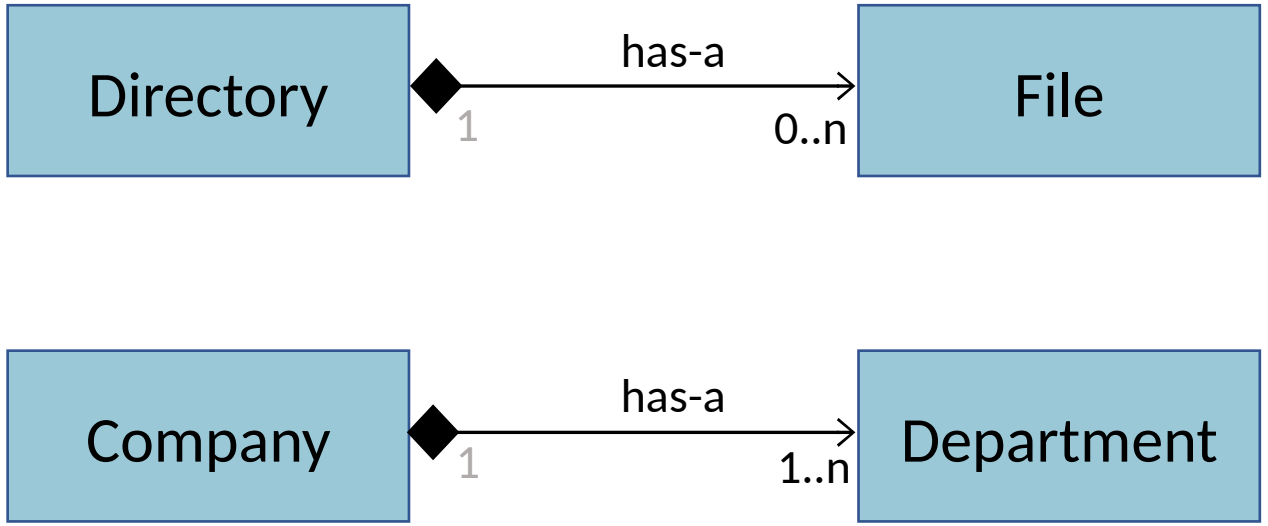
What is the *conceptual* difference between the above relationships?

Class Diagrams: *Bidirectional* Association



- **Bidirectional navigation**
- What *bank accounts* does *John Doe* **own**?
- What *persons* does the *bank account AB123* **belong to**?

Composition: Definition and Examples



- **Composition** is a form of *aggregation*
  - **Strong ownership**—a **part** only **belongs** to a **single whole** (*1* multiplicity)
  - **Coincident lifetime** between the *parts* of the *whole*
- Example: A *file* is **part of** a *directory*; it is **destroyed** if its *directory* is **destroyed**

Question

- How would you implement the previous *Aggregation* between *Computer* and *CPU*?

Answer on PollEveryWhere

https://pollev.com/francescotusa

Implementing an *Aggregation*: Example

```
public class Computer
{
    private String model;
    private String brand;
    private CPU [] cpus;
    private int lastIndex;

    public Computer(String model, String brand, CPU cpu)
    {
        this.model = model;
        this.brand = brand;
        cpus = new CPU[64];
        lastIndex = 0;
        cpus[0] = cpu;
    }

    public void addCPU(CPU cpu)
    {
        cpus[lastIndex++] = cpu;
    }
}
```

a Computer object contains a **reference** to a *collection* (e.g., an array) of CPU objects

Implementing an *Aggregation*: Example

```
public class Computer
{
    private String model;
    private String brand;
    private CPU [] cpus;
    private int lastIndex;

    public Computer(String model, String brand, CPU cpu)
    {
        this.model = model;
        this.brand = brand;
        cpus = new CPU[64];
        lastIndex = 0;
        cpus[0] = cpu;
    }

    public void addCPU(CPU cpu)
    {
        cpus[lastIndex++] = cpu;
    }
}
```

A reference to an externally defined CPU object is passed to the method and is added to the *cpus* array.

The object referenced by the *cpu* parameter has its lifetime

It can live **independently** of the Computer object to which is passed

Implementing an *Aggregation*: Example

```
public class Computer
{
    private String model;
    private String brand;
    private CPU [] cpus;
    private int lastIndex;

    public Computer(String model, String brand, CPU cpu) { ... }

    public void addCPU(CPU cpu) { ... }

}

class Program {
    static void main(String[] args)  {
        CPU cpu1 = new CPU("i7", "Intel", 3.0);
        Computer pc1 = new Computer("ModelX", "Dell", cpu1);
        // if pc1 goes out of scope and is removed from the heap, cpu1 will still exist
        // ...
        Computer laptop = new Computer("ModelZ", "Lenovo", cpu1); // reuse cpu1 object
        CPU cpu2 = new CPU("Ryzen", "AMD", 3.5);
        laptop.addCPU(cpu2);
    }
}
```

Implementing a *Composition*: Example

```
public class Directory
{
    private String name;
    private String creationTime;
    private File[] files;
    private int lastIndex;

    public Directory(String n)
    {
        name = n;
        creationTime = // assign current date
        files = new File[1000];
        lastIndex = 0;
    }

    public void createFile(String name)
    {
        files[lastIndex++] = new File(name);

    }
}
```

A File object is completely encapsulated within a Directory object

No references are available elsewhere

The File object lives and dies with the Directory object wherein it is created

If a Directory object is garbage collected, so will the (encapsulated) referenced File objects

Outline

- Object Relationships
  - Summary and Recap on UML Class Diagrams
  - Association, Aggregation and Composition
- Class Relationships: Generalisation and Inheritance
  - **Definitions**
  - Class Diagrams
  - Code Implementation and the ‘super’ keyword
  - Protected access modifier
  - Going Further with Inheritance

Question

- Question on *generalisation* and *inheritance*

Answer on PollEveryWhere

https://pollev.com/francescotusa

Class Relationships: Generalisation

- Different types of objects interact with each other through **relationships**

Association: A *Person owns a BankAccount*

- The code of the classes from which those objects are created are **separate modules** of a program
- Can be *developed*, *tested*, *extended* and *maintained* **independently**
- The code of a class can be **reused** in **several programs**

Class Relationships: Generalisation

- Different types of objects interact with each other through **relationships**

Association: A *Person owns a BankAccount*

- What if we have *multiple* types of people: *Student* and *Employee*
- Both share **common characteristics** but also have their **unique features**
- How can the **code** be **organised efficiently** and **reused**?

Class Relationships: Generalisation

- A *Student* “is-a-kind-of” *Person*, an *Employee* “is-a-kind-of” *Person*

- A **general** kind of thing (*superclass* or *parent*) can have a relationship with a **more specific** kind (*subclass* or *child*).
- We call it **generalisation** or “**is-a-kind-of**” relationship

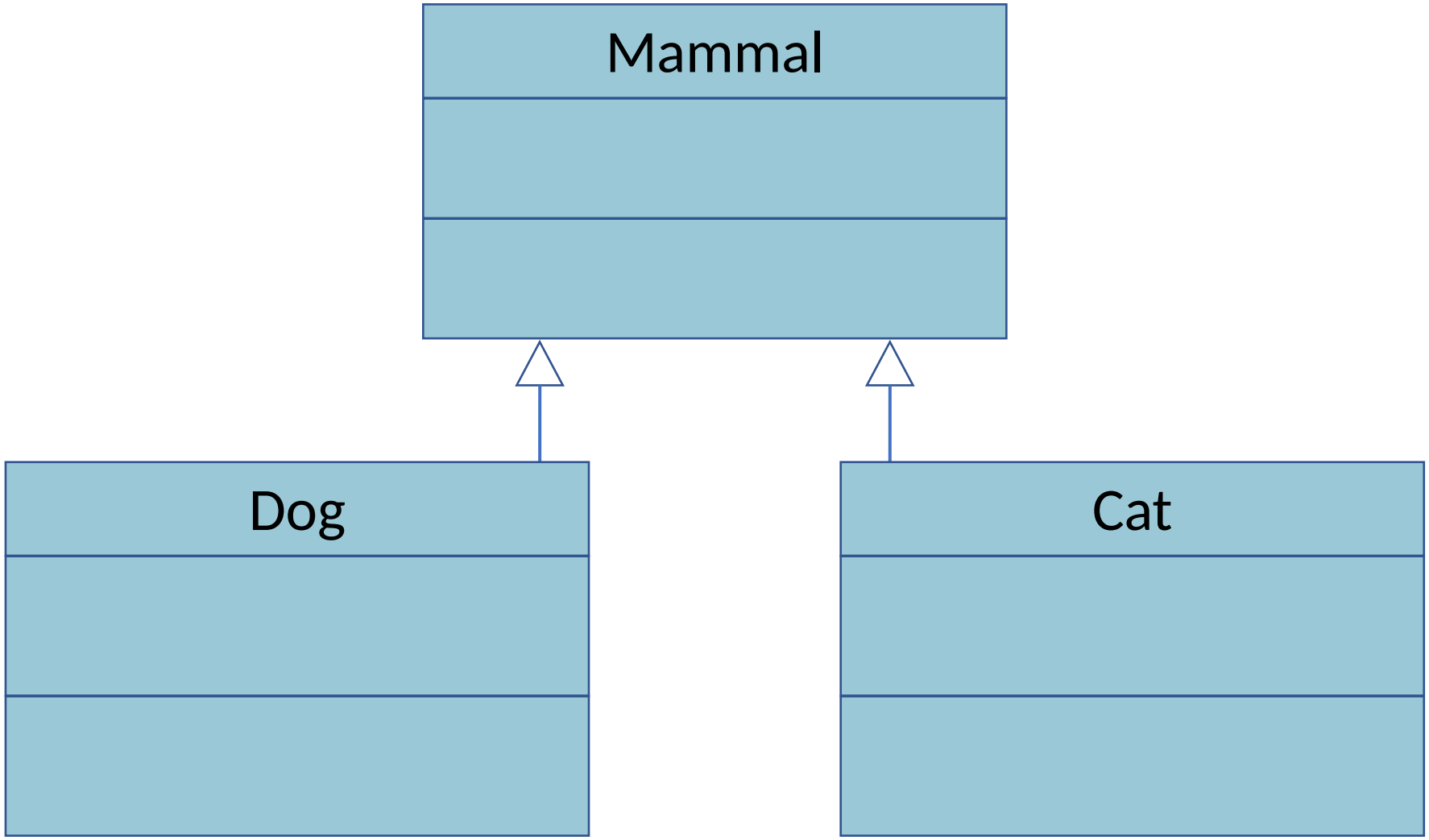
Class Relationships: Generalisation

- A **general** kind of thing: *Mammal*
- Can have a relationship with a **more specific** kind of thing: *Dog*, *Cat*
- A *Dog* (or a *Cat*) “is-a-kind-of” *Mammal*

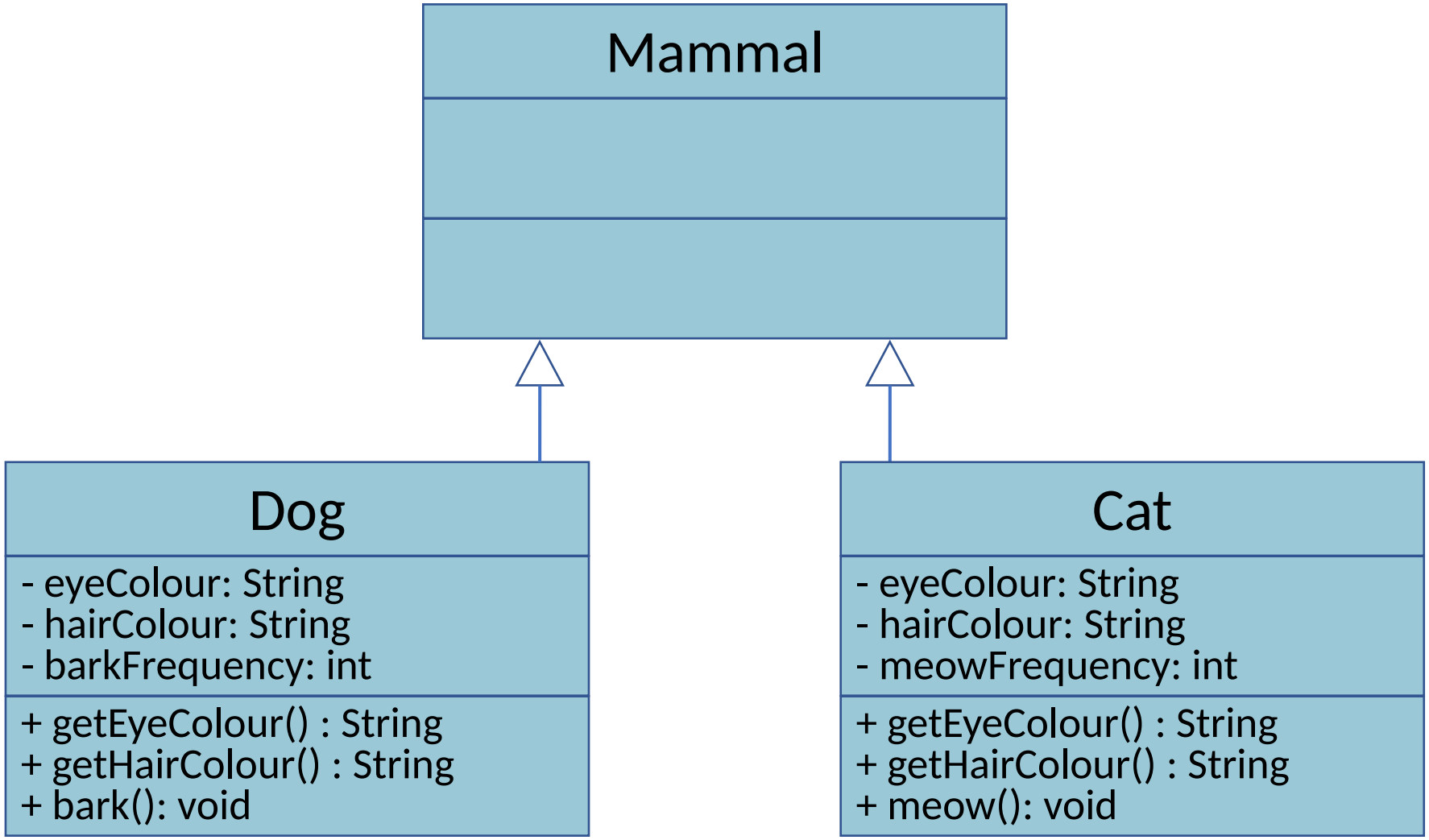
Outline

- Object Relationships
  - Summary and Recap on UML Class Diagrams
  - Association, Aggregation and Composition
- Class Relationships: Generalisation and Inheritance
  - Definitions
  - **Class Diagrams**
  - Code Implementation and the ‘super’ keyword
  - Protected access modifier
  - Going Further with Inheritance

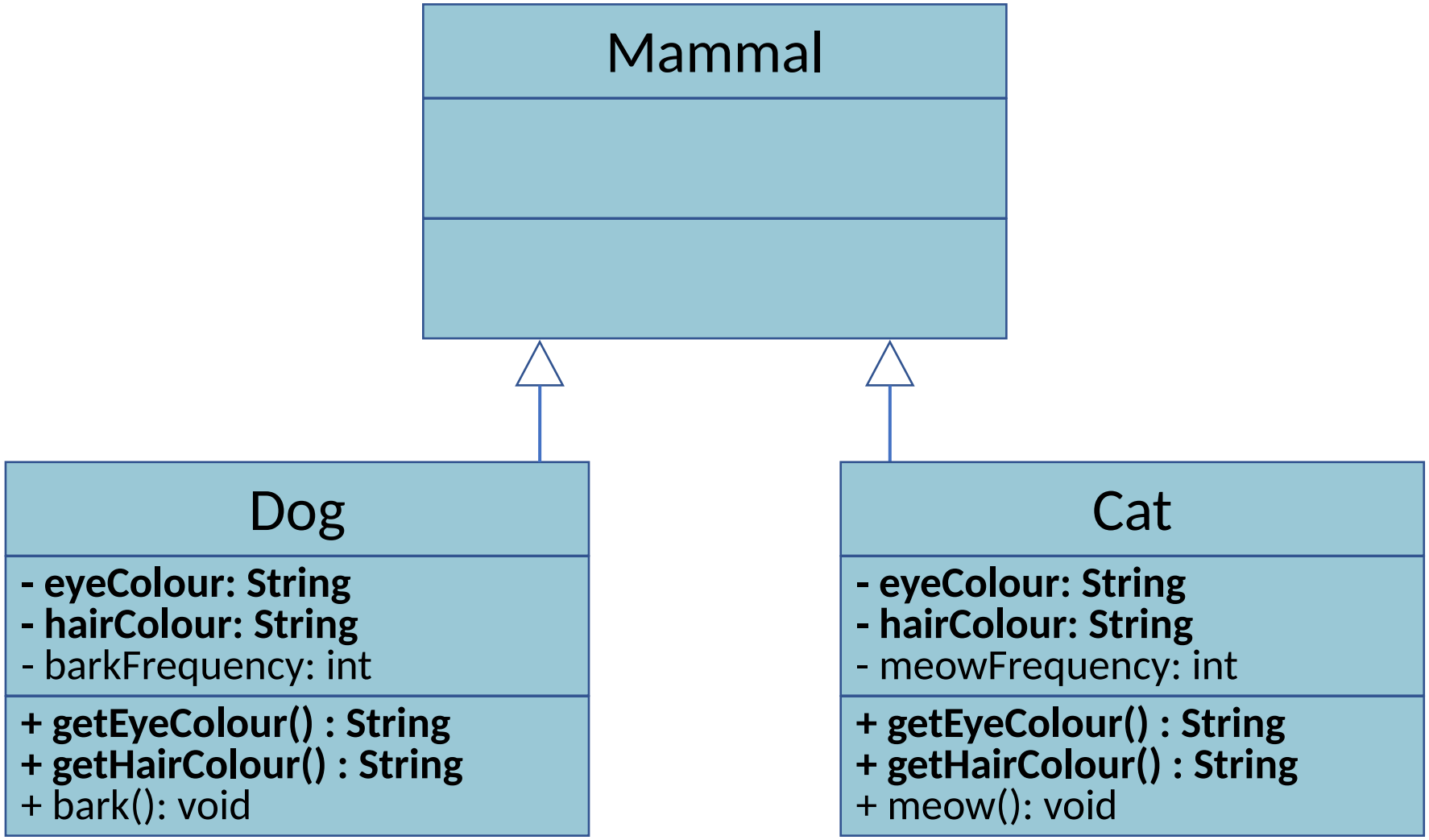
Class Diagrams: Generalisation and Inheritance



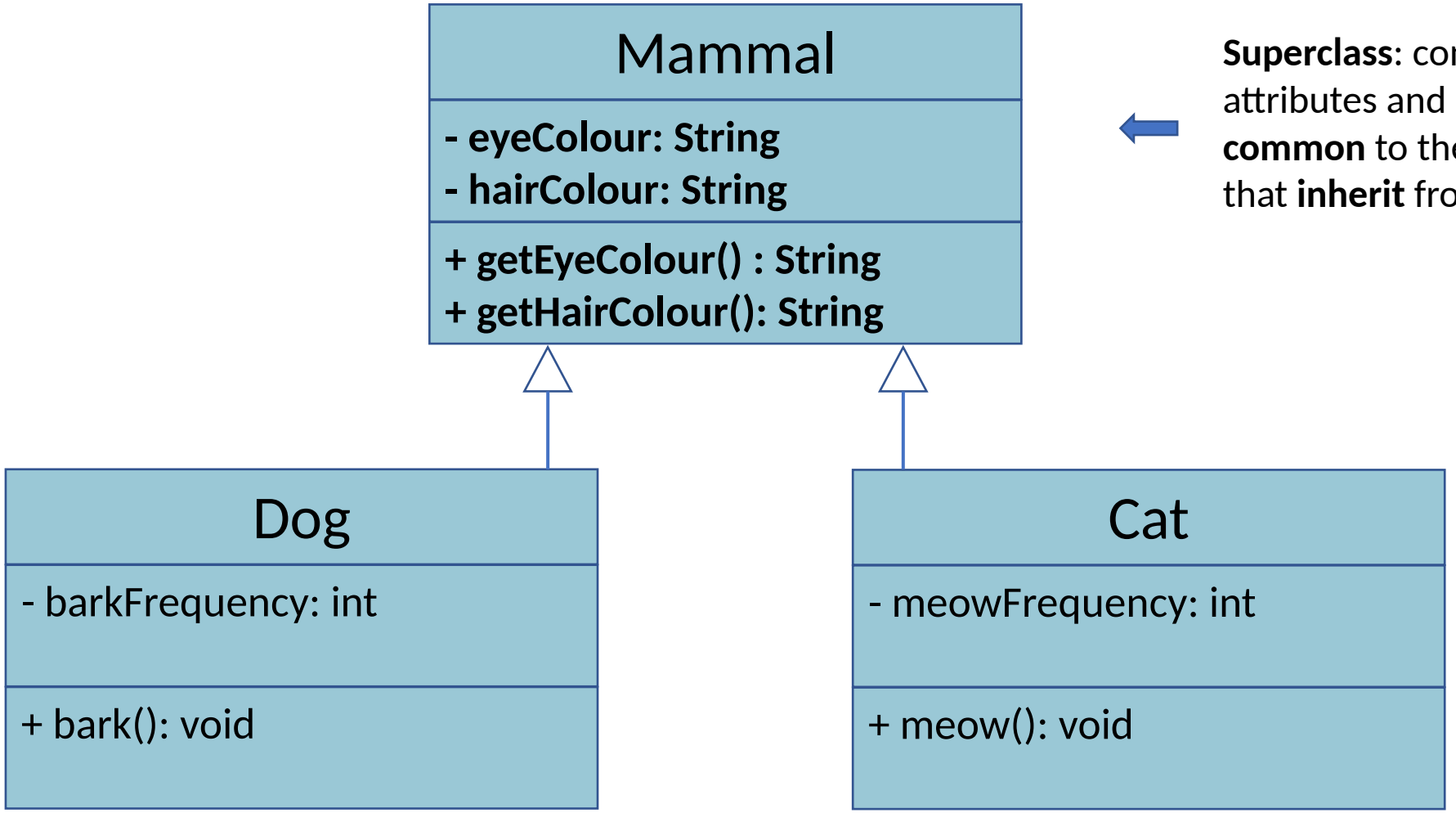
Class Diagrams: Generalisation and Inheritance



Class Diagrams: Generalisation and Inheritance

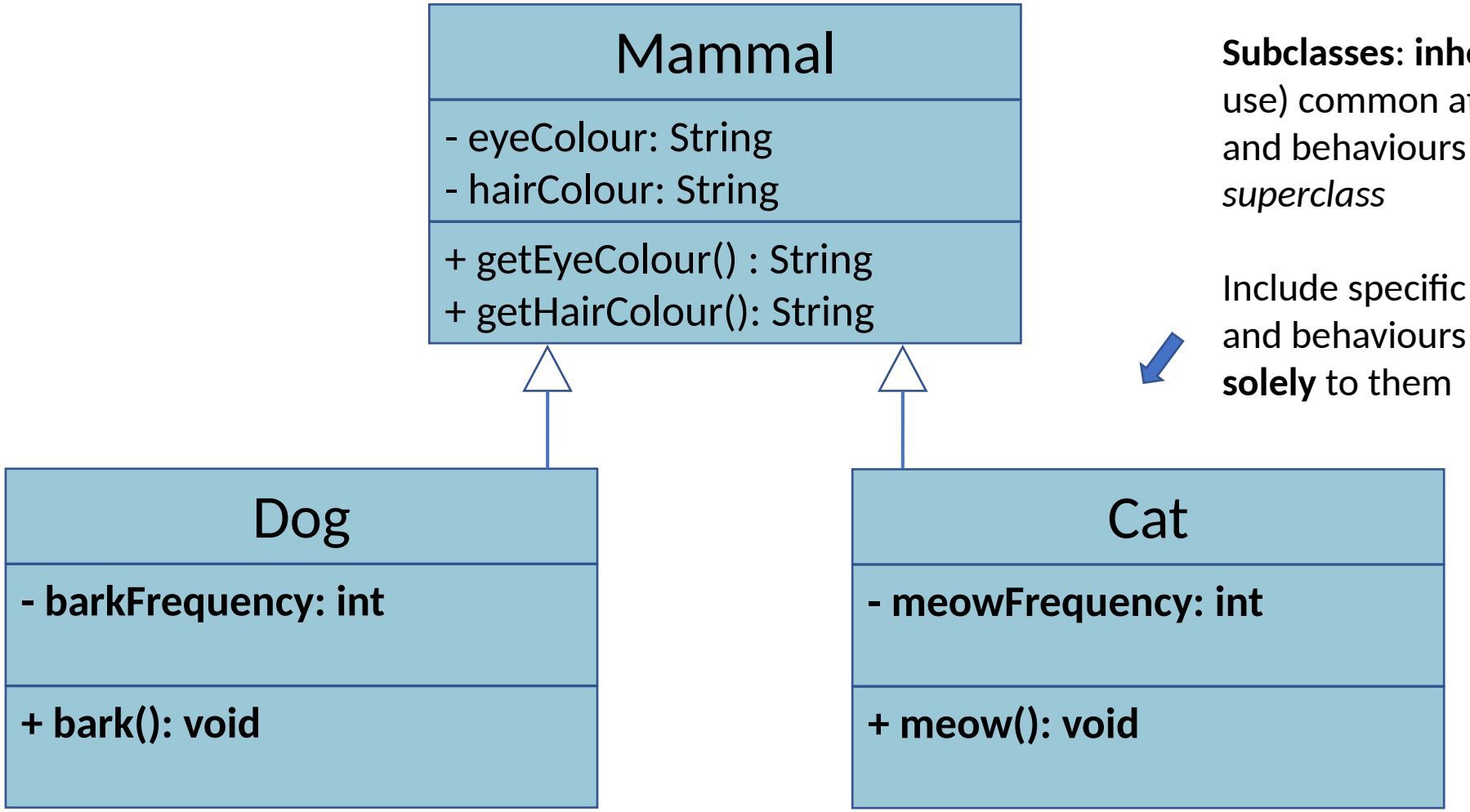


Class Diagrams: Generalisation and Inheritance



Superclass: contains attributes and behaviours **common** to the subclasses that **inherit** from it

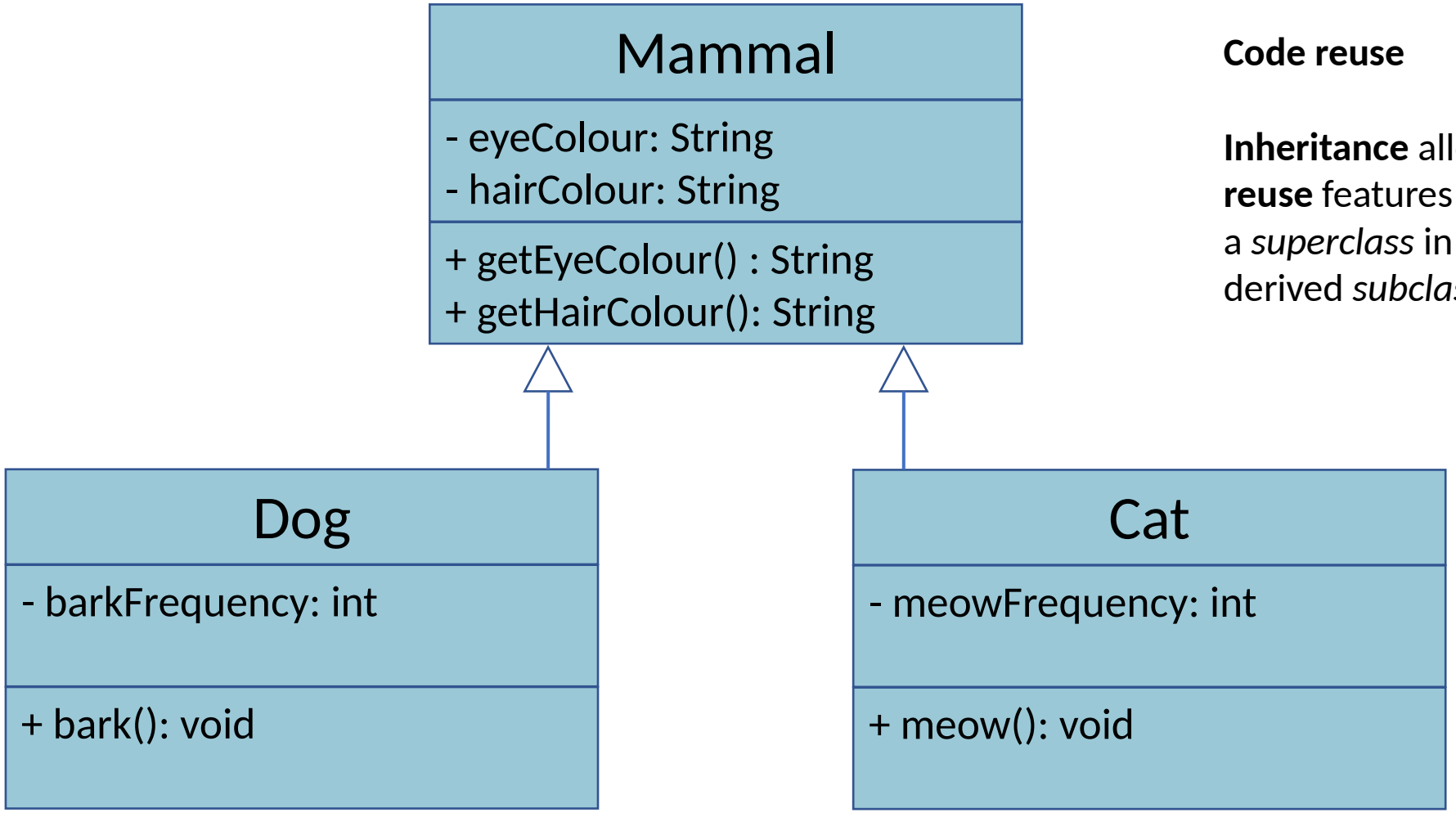
Class Diagrams: Generalisation and Inheritance



Subclasses: **inherit** (can use) common attributes and behaviours from the *superclass*

Include specific attributes and behaviours that pertain **solely** to them

Class Diagrams: Generalisation and Inheritance



Code reuse

Inheritance allows to **reuse** features defined in a *superclass* in all the derived *subclasses*

Question

- If I created an object of the Mammal class called myMammal, what of the following statements would be true?

Answer on PollEveryWhere

https://pollev.com/francescotusa

Outline

- Object Relationships
  - Summary and Recap on UML Class Diagrams
  - Association, Aggregation and Composition
- Class Relationships: Generalisation and Inheritance
  - Definitions
  - Class Diagrams
  - **Code Implementation** and the ‘super’ keyword
  - Protected access modifier
  - Going Further with Inheritance

Code: Generalisation and Inheritance

```
public class Mammal
{
    private String eyeColour;
    private String hairColour;

    public Mammal(String ec, String hc)
    {
        eyeColour = ec;
        hairColour = hc;
    }

    public String getEyeColour()
    {
        return eyeColour;
    }

    public String getHairColour()
    {
        return hairColour;
    }
}
```

Code: Generalisation and Inheritance

usage of **extends** followed by the *superclass* name

```
public class Mammal
{
    private String eyeColour;
    private String hairColour;

    public Mammal(String ec, String hc)
    {
        eyeColour = ec;
        hairColour = hc;
    }

    public String getEyeColour()
    {
        return eyeColour;
    }

    public String getHairColour()
    {
        return hairColour;
    }
}
```

```
public class Dog extends Mammal
{
    int barkFrequency;

    public Dog(String ec, String hc, int bf)
    {
        // attributes initialisation
    }

    public void bark()
    {
        // uses barkFrequency
    }

    // inherits getEyeColour and
    // getHairColour from Mammal
}
```

Code: Generalisation and Inheritance

```
public class Mammal
{
    private String eyeColour;
    private String hairColour;

    public Mammal(String ec, String hc)
    {
        eyeColour = ec;
        hairColour = hc;
    }

    public String getEyeColour()
    {
        return eyeColour;
    }

    public String getHairColour()
    {
        return hairColour;
    }
}
```

```
public class Dog extends Mammal
{
    int barkFrequency;

    public Dog(String ec, String hc, int bf)
    {
        // attributes initialisation
    }

    public void bark()
    {
        // uses barkFrequency
    }

    // inherits getEyeColour and
    // getHairColour from Mammal
}
```

the code would be inherited, no need to duplicate it

Code: Generalisation and Inheritance

```
public class Mammal
{
    private String eyeColour;
    private String hairColour;

    public Mammal(String ec, String hc)
    {
        eyeColour = ec;
        hairColour = hc;
    }

    public String getEyeColour()
    {
        return eyeColour;
    }

    public String getHairColour()
    {
        return hairColour;
    }
}
```

```
public class Dog extends Mammal
{
    int barkFrequency;          ← specific Dog's attribute

    public Dog(String ec, String hc, int bf)
    {
        // attributes initialisation
    }

    public void bark()          ← specific Dog's behaviour
    {
        // uses barkFrequency
    }

    // inherits getEyeColour and
    // getHairColour from Mammal
}
```

Code: Generalisation and Inheritance

```
public class Mammal
{
    private String eyeColour;
    private String hairColour;

    public Mammal(String ec, String hc)
    {
        eyeColour = ec;
        hairColour = hc;
    }

    public String getEyeColour()
    {
        return eyeColour;
    }

    public String getHairColour()
    {
        return hairColour;
    }
}
```

```
public class Cat extends Mammal
{
    int meowFrequency;

    public Cat(String ec, String hc, int mf)
    {
        // attributes initialisation
    }

    public void meow()
    {
        // uses meowFrequency
    }

    // inherits getEyeColour and
    // getHairColour from Mammal
}
```

specific Cat's *attribute*

specific Cat's *behaviour*

the code would be inherited, no need to duplicate it

Answer to the Poll

- Dog and Cats are Mammals so they **inherit**
  - getEyeColour and getHairColour

- **Not the other way around**: a Mammal does **not** have access to the methods of the Dog and Cat classes bark and meow

Code: Generalisation and Inheritance

```
public class Program
{
    public static void main(String[] args)
    {
        // create a Dog called alan
        Dog alan = new Dog ("brown", "white", 10);
        String colour1 = alan.getEyeColour();
        alan.bark();

        // create a Cat called felix
        Cat felix = new Cat ("green", "black", 30);
        String colour2 = felix.getHairColour();
        felix.meow();
    }
}
```

Outline

- Object Relationships
  - Summary and Recap on UML Class Diagrams
  - Association, Aggregation and Composition
- Class Relationships: Generalisation and Inheritance
  - Definitions
  - Class Diagrams
  - Code Implementation and **the ‘super’ keyword**
  - Protected access modifier
  - Going Further with Inheritance

Code: the super keyword

- A subclass **cannot access directly**:
  - The *private* members of its parent class
  - The *constructors* of its parent class

- The subclass constructor will have to initialise **its class attributes** and **those of the superclass**

How?

Code: the super keyword

```
public class Mammal
{
    private String eyeColour;
    private String hairColour;

    public Mammal(String ec, String hc)
    {
        eyeColour = ec;
        hairColour = hc;
    }

    public String getEyeColour()
    {
        return eyeColour;
    }

    public String getHairColour()
    {
        return hairColour;
    }
}
```

```
public class Dog extends Mammal
{
    int barkFrequency;

    public Dog(String ec, String hc, int bf)
    {
        eyeColour = ec
        hairColour = hc
        barkFrequency = bf;
    }

    public void bark()
    {
        // uses barkFrequency
    }

    // inherits getEyeColour and
    // getHairColour from Mammal
}
```

```
new Dog ("brown", "white", 10);
```

Code: the super keyword

```
public class Mammal
{
    private String eyeColour;
    private String hairColour;

    public Mammal(String ec, String hc)
    {
        eyeColour = ec;
        hairColour = hc;
    }

    public String getEyeColour()
    {
        return eyeColour;
    }

    public String getHairColour()
    {
        return hairColour;
    }
}
```

```
public class Dog extends Mammal
{
    int barkFrequency;

    public Dog(String ec, String hc, int bf)
    {
        eyeColour = ec
        hairColour = hc
        barkFrequency = bf;
    }

    public void bark()
    {
        // uses barkFrequency
    }

    // inherits getEyeColour and
    // getHairColour from Mammal
}
```

NO! They are defined as private in Mammal

```
new Dog ("brown", "white", 10);
```

Code: the super keyword

```
public class Mammal
{
    private String eyeColour;
    private String hairColour;

    public Mammal(String ec, String hc)
    {
        eyeColour = ec;
        hairColour = hc;
    }

    public String getEyeColour()
    {
        return eyeColour;
    }

    public String getHairColour()
    {
        return hairColour;
    }
}
```

```
public class Dog extends Mammal
{
    int barkFrequency;

    public Dog(String ec, String hc, int bf)
    {
        eyeColour = ec
        hairColour = hc
        barkFrequency = bf;
    }

    public void bark()
    {
        // uses barkFrequency
    }

    // inherits getEyeColour and
    // getHairColour from Mammal
}
```

remember, we can chain the invocation of constructors of the same class by using *this(...)*

Code: the super keyword

```
public class Mammal
{
    private String eyeColour;
    private String hairColour;

    public Mammal(String ec, String hc)
    {
        eyeColour = ec;
        hairColour = hc;
    }

    public String getEyeColour()
    {
        return eyeColour;
    }

    public String getHairColour()
    {
        return hairColour;
    }
}
```

```
public class Dog extends Mammal
{
    int barkFrequency;

    public Dog(String ec, String hc, int bf)
    {
        eyeColour = ec
        hairColour = hc
        barkFrequency = bf;
    }

    public void bark()
    {
        // uses barkFrequency
    }

    // inherits getEyeColour and
    // getHairColour from Mammal
}
```

similarly, even though superclass constructors are not inherited, they can be called using *super(…)*

Code: the super keyword

```
public class Mammal
{
    private String eyeColour;
    private String hairColour;

    public Mammal(String ec, String hc)
    {
        eyeColour = ec;
        hairColour = hc;
    }

    public String getEyeColour()
    {
        return eyeColour;
    }

    public String getHairColour()
    {
        return hairColour;
    }
}
```

```
public class Dog extends Mammal
{
    int barkFrequency;

    public Dog(String ec, String hc, int bf)
    {
        super(ec, hc);
        barkFrequency = bf;
    }

    public void bark()
    {
        // uses barkFrequency
    }

    // inherits getEyeColour and
    // getHairColour from Mammal
}
```

super(*ec*, *hc*) will call the constructor of Mammal and will pass the arguments *ec* and *hc*

Code: the super keyword

```
public class Mammal
{
    private String eyeColour;
    private String hairColour;

    public Mammal(String ec, String hc)
    {
        eyeColour = ec;
        hairColour = hc;
    }

    public String getEyeColour()
    {
        return eyeColour;
    }

    public String getHairColour()
    {
        return hairColour;
    }
}
```

```
public class Dog extends Mammal
{
    int barkFrequency;

    public Dog(String ec, String hc, int bf)
    {
        super(ec, hc);
        barkFrequency = bf;
    }

    public void bark()
    {
        // uses barkFrequency
    }

    // inherits getEyeColour and
    // getHairColour from Mammal
}
```

the private attributes of Mammal will be initialised through that call

Code: the super keyword

```
public class Mammal
{
    private String eyeColour;
    private String hairColour;

    public Mammal(String ec, String hc)
    {
        eyeColour = ec;
        hairColour = hc;
    }

    public String getEyeColour()
    {
        return eyeColour;
    }

    public String getHairColour()
    {
        return hairColour;
    }
}
```

```
public class Dog extends Mammal
{
    int barkFrequency;

    public Dog(String ec, String hc, int bf)
    {
        super(ec, hc);
        barkFrequency = bf;
    }

    public void bark()
    {
        // uses barkFrequency
    }

    // inherits getEyeColour and
    // getHairColour from Mammal
}
```

and will then be accessible via the *inherited* getEyeColour and getHairColour methods

Code: the super keyword

```
public class Mammal
{
    private String eyeColour;
    private String hairColour;

    public Mammal(String ec, String hc)
    {
        eyeColour = ec;
        hairColour = hc;
    }

    public String getEyeColour()
    {
        return eyeColour;
    }

    public String getHairColour()
    {
        return hairColour;
    }
}
```

```
public class Dog extends Mammal
{
    int barkFrequency;

    public Dog(String ec, String hc, int bf)
    {
        super(ec, hc);
        barkFrequency = bf;
    }

    public void bark()
    {
        // uses barkFrequency
    }

    // inherits getEyeColour and
    // getHairColour from Mammal
}
```

another approach could be to declare those attributes as protected in Mammal

Outline

- Object Relationships
  - Summary and Recap on UML Class Diagrams
  - Association, Aggregation and Composition
- Class Relationships: Generalisation and Inheritance
  - Definitions
  - Class Diagrams
  - Code Implementation and the ‘super’ keyword
  - **Protected access modifier**
  - Going Further with Inheritance

Code: the protected access modifier

So far, we have used:

- private: access to members restricted to the **same** class
- public: access to members allowed to **any external** class

Code: the protected access modifier

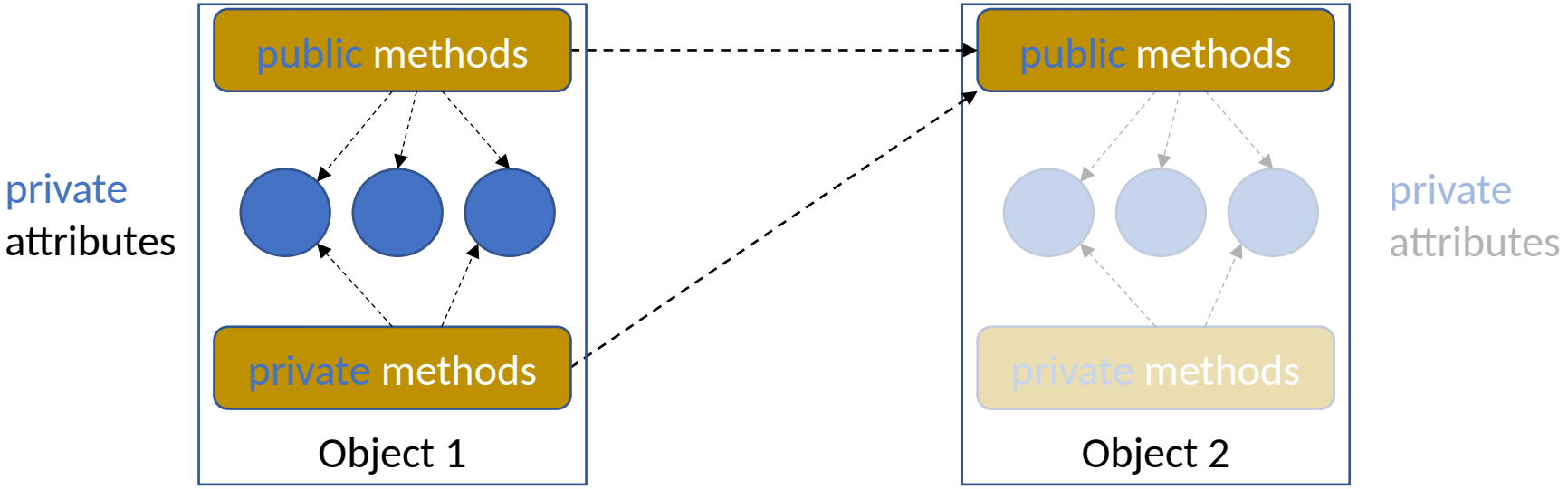
So far, we have used:

- private: access to members restricted to the **same** class
- public: access to members allowed to **any external** class

protected: members can be accessed from the **same** class, other classes of the **same package**, and from **any subclass**

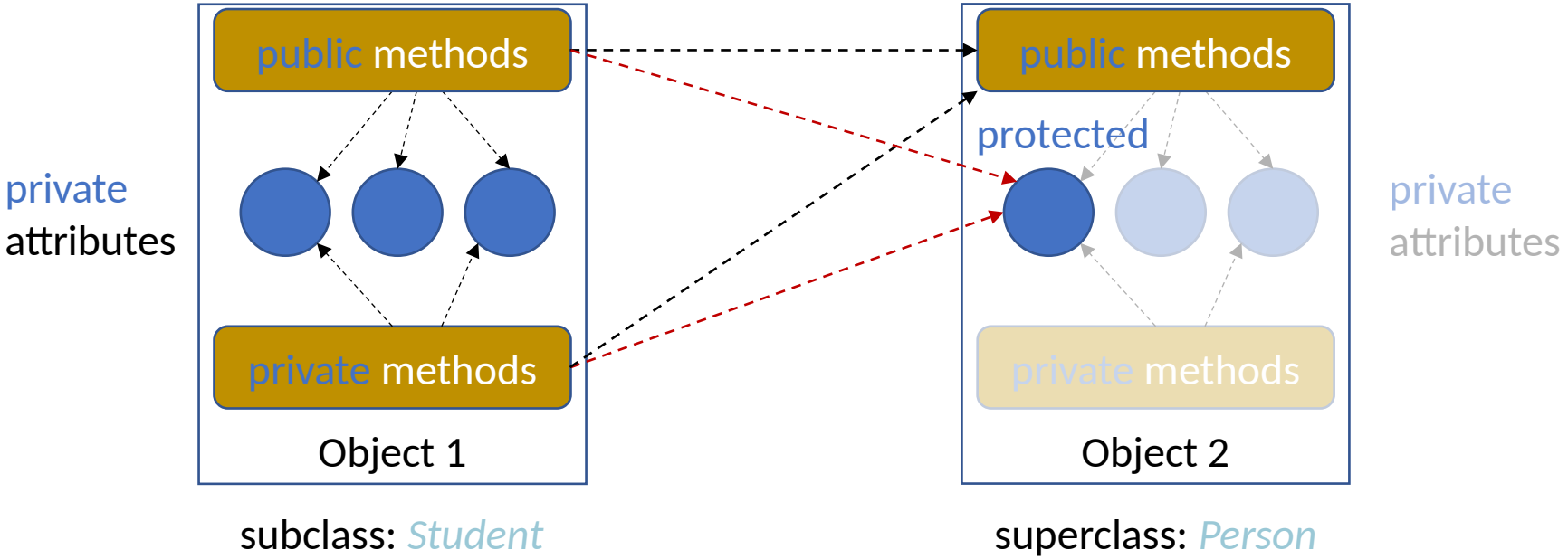
Code: the protected access modifier (interface reminder)

- Objects contain **both** the *attributes* and *behaviours*
- An object should reveal **only** the **interface** that other objects must use to **interact** with it
- Further details should be **hidden**



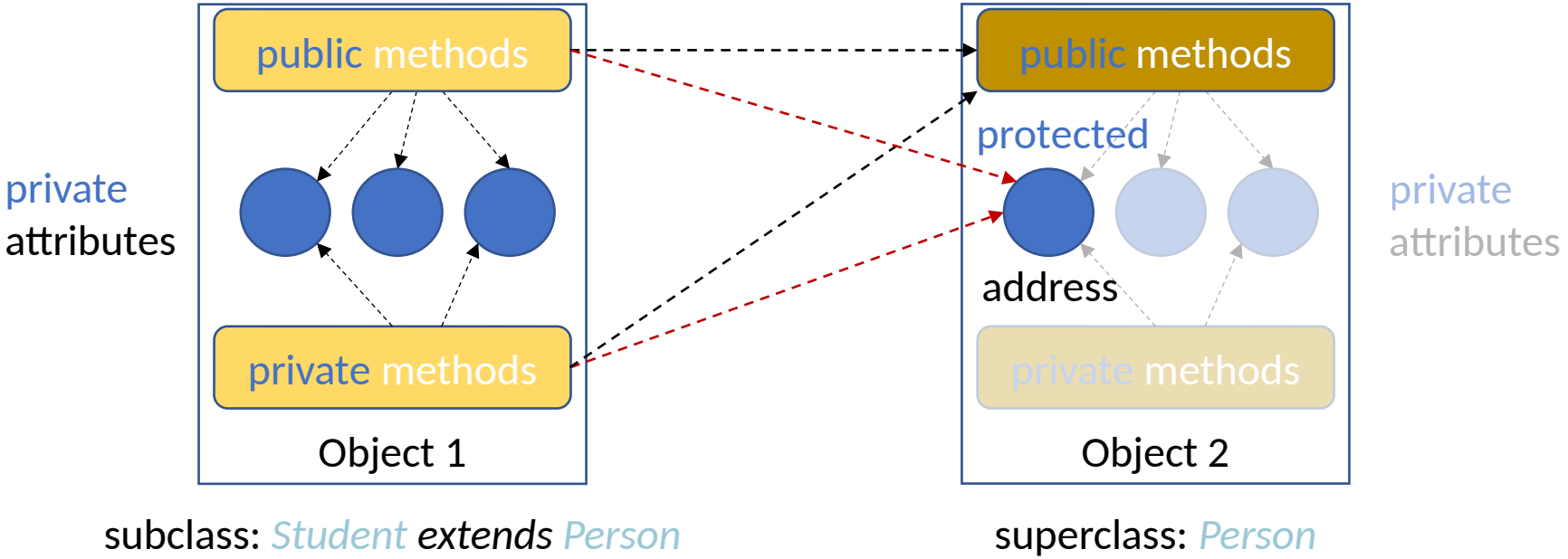
Code: the protected access modifier

- A protected attribute could **weaken encapsulation**



Code: the protected access modifier

- A protected attribute could **weaken encapsulation**
- Example: the change of the **address** attribute from String to *Address* in a *Person* superclass would require changes to a *Student* subclass



Code: the protected access modifier

So far, we have used:

- private: access to members restricted to the **same** class
- public: access to members allowed to **any external** class

protected: members can be accessed from the **same** class, other classes of the **same package**, and from **any subclass**

best practice: use private attributes and superclass constructors; define public or protected *getter* and *setter* methods in the superclass

Outline

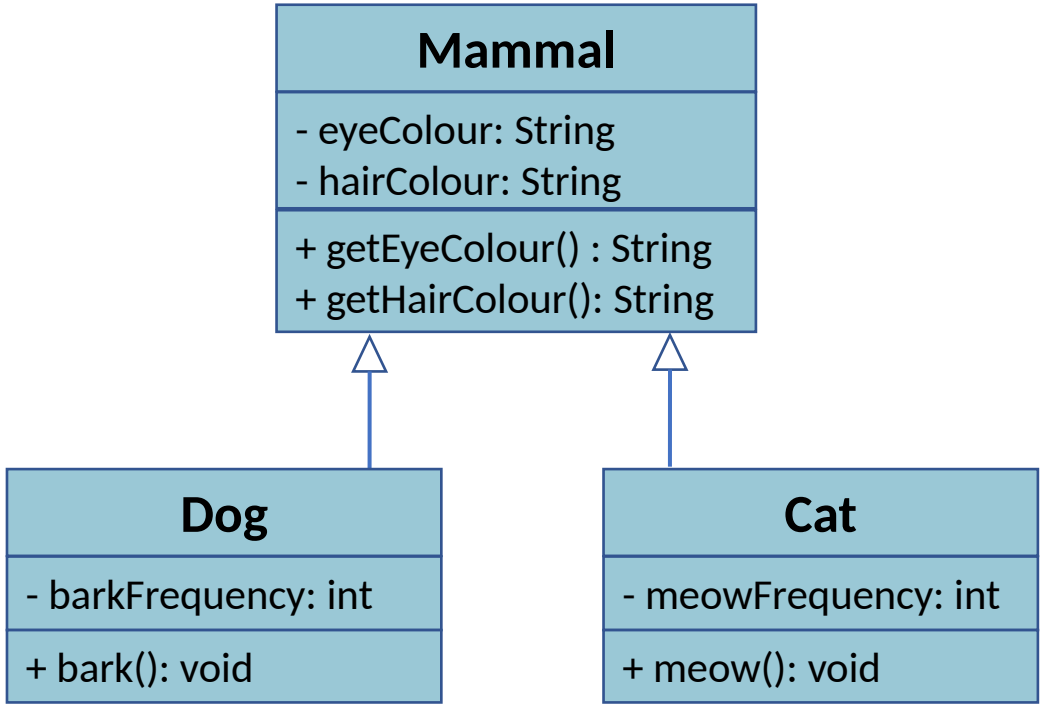
- Object Relationships
  - Summary and Recap on UML Class Diagrams
  - Association, Aggregation and Composition
- Class Relationships: Generalisation and Inheritance
  - Definitions
  - Class Diagrams
  - Code Implementation and the ‘super’ keyword
  - Protected access modifier
  - **Going Further with Inheritance**

Going further with Inheritance

- When there is a **generalisation** relationship, a *subclass* "**is-a-kind-of**" a *superclass*
- The *subclass* **inherits** and can **reuse attributes** and **methods** of the *superclass*

- How can this make **code development** of new classes more **efficient**?

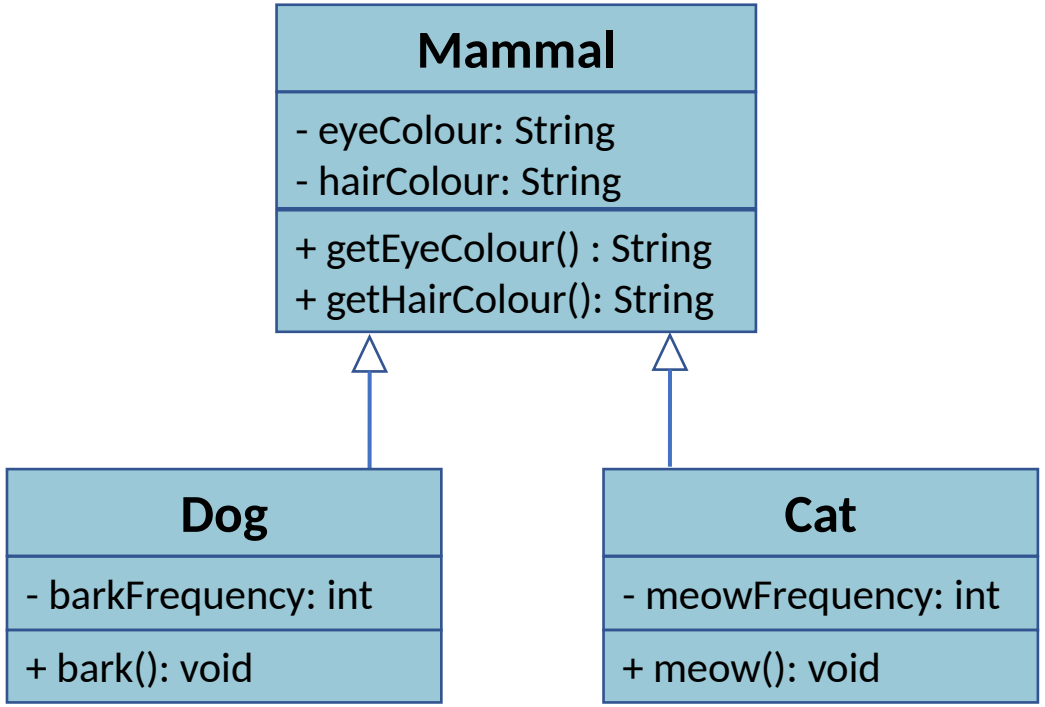
Going further with Inheritance



GoldenRetriever class?

Create it from scratch or…

Going further with Inheritance

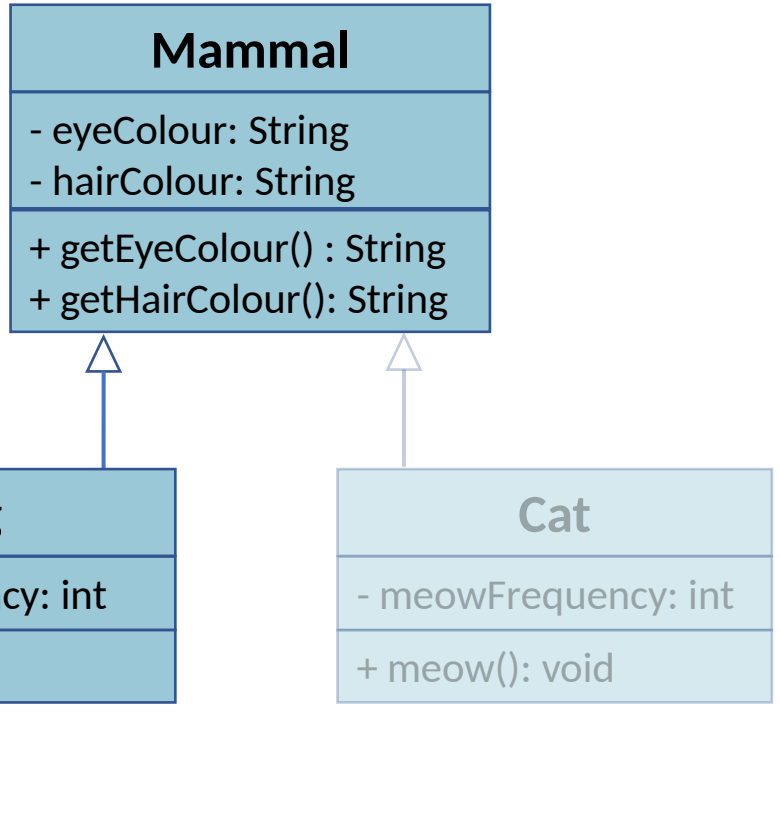


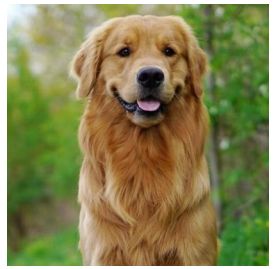
...*GoldenRetriever* is-a-kind-of *Dog* (more specialised)

Generalisation can be applied to create a **hierarchy** of classes

GoldenRetriever further **inherits** *attributes* and *behaviours* from *Dog*

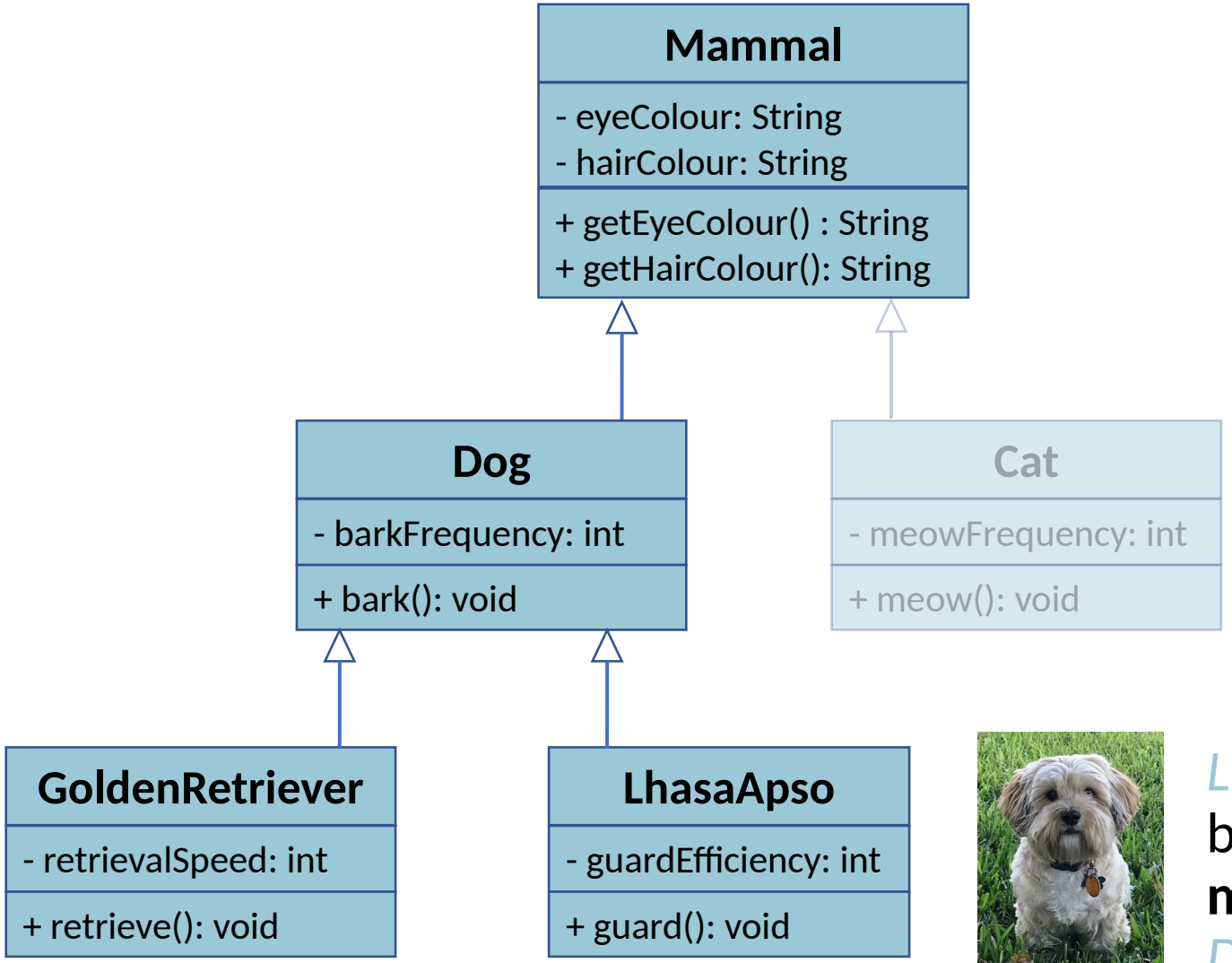
Going further with Inheritance





GoldenRetriever now contains **its** behaviour – **retrieve** – as well as the **more general** behaviours of a *Dog* and of a *Mammal*

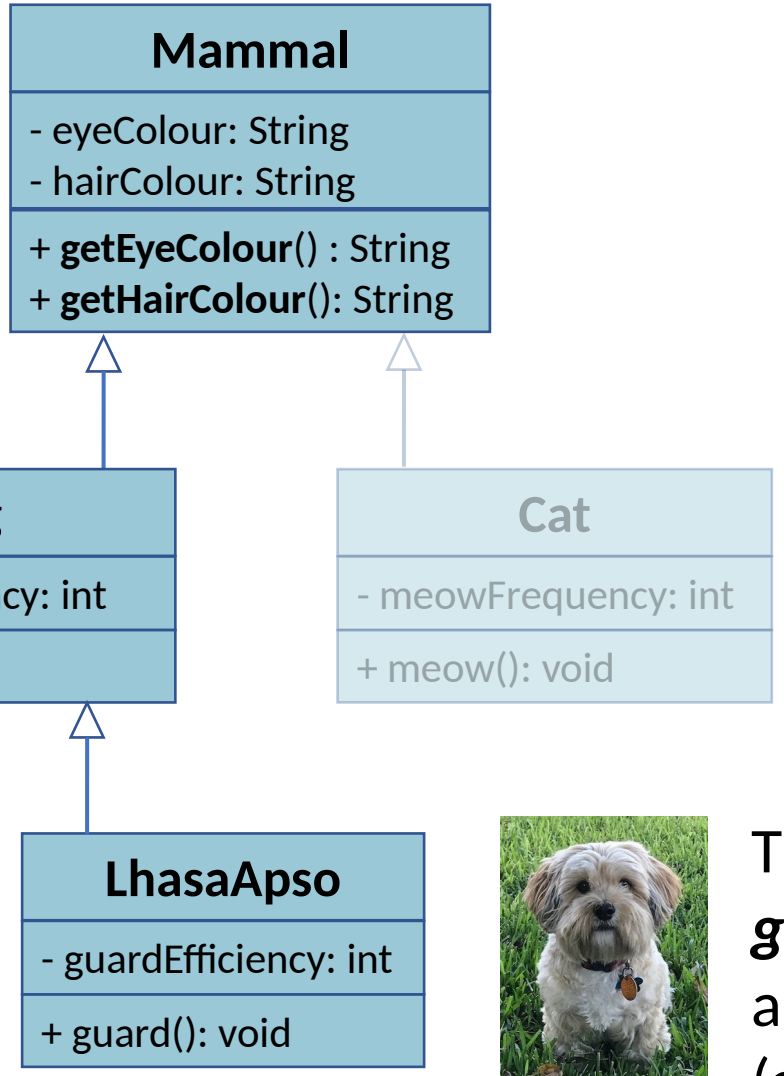
Going further with Inheritance

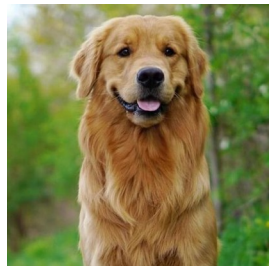


LhasaApso contains **its** behaviour – **guard** – and the **more general** behaviours of a *Dog* and a *Mammal*

Going further with Inheritance: benefits

Less design and coding time

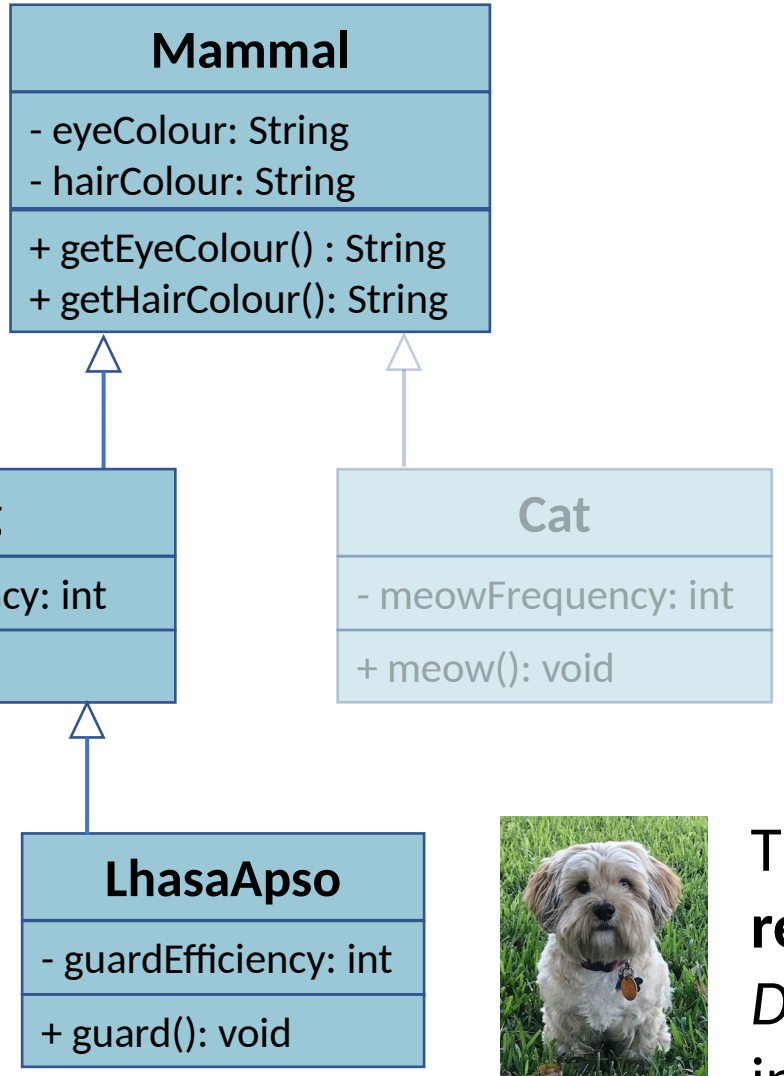




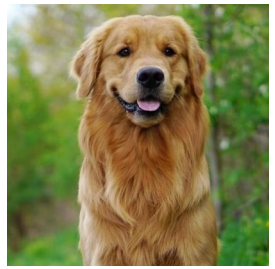
The *inherited* methods ***getEyeColour***, ***getHairColour*** and ***bark*** are effectively **reused** (after **testing**)

Going further with Inheritance: benefits

Less maintenance time



Code changes confined within a single place (e.g., **bark**)



The ***bark*** code is **not replicated**: code changes in *Dog* are inherently **reflected** in **all** the subclasses

Inheritance: summary

- A *child* class ***inherit*** and can ***reuse*** the *attributes* and *methods* defined by a *parent* class (superclass).
- **Benefits**: reuse of existing code
  - **Less** *coding* and *testing* time
  - **Less** *maintenance* time and potential *inconsistencies*

Question

- Identify the right answer on ***Inheritance***

Answer on PollEveryWhere

https://pollev.com/francescotusa

Object-Oriented Programming (OOP) Principles

- Abstraction
- Encapsulation
- **Inheritance**
- Polymorphism

A *child* class *inherit* the *attributes* and *methods* a parent class (superclass) defines, which makes code better **organised**, **reusable** and **easier to maintain**.

Questions