7SENG011W Object Oriented Programming

UML class diagrams, object relationships, generalisation and inheritance

Dr Francesco Tusa

Readings

The topics we will discuss today can be found in the books

- Programming C# 10
  - Chapter 3: Types
- Hands-On Object-Oriented Programming with C#
  - Chapter Object Collaboration: from Association to Inheritance
- UML Distilled
  - Chapter 3: Class Diagrams
- Object-Oriented Thought Process
  - Chapter 7: Mastering Inheritance and Composition

Online

- IBM: UML Class Diagrams
- Inheritance concepts
- Inheritance in C# and .NET

Outline

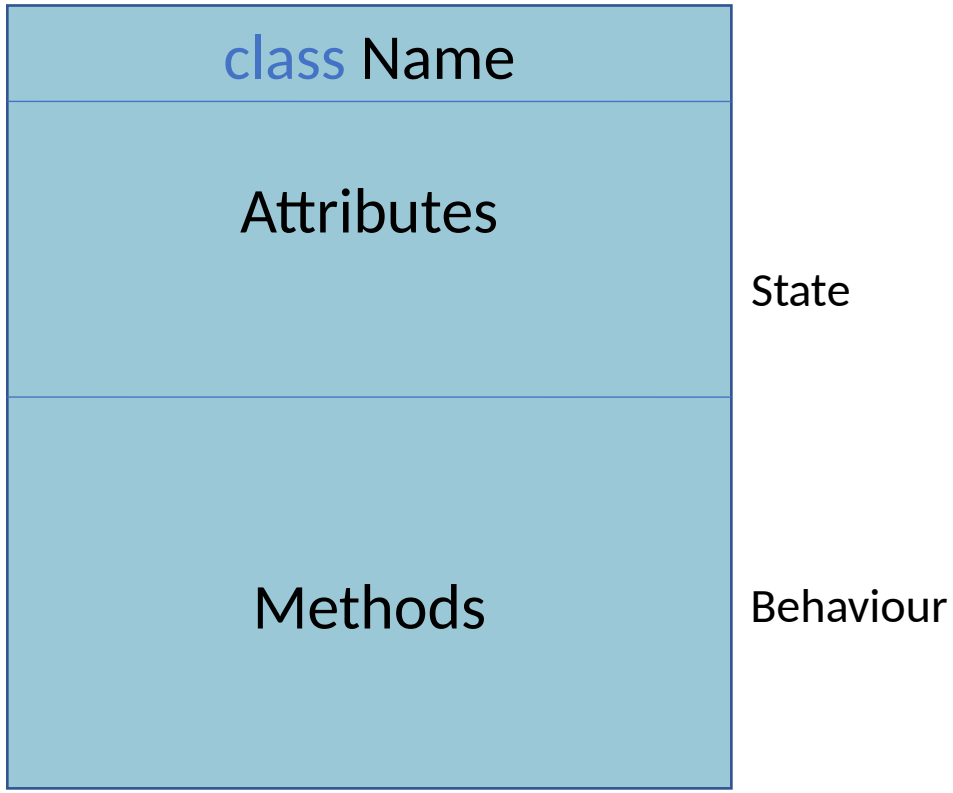
- More on UML Class Diagrams: Object Relationships
- Class Relationships: Generalisation and Inheritance

Outline

- More on UML Class Diagrams: Object Relationships
- Class Relationships: Generalisation and Inheritance

Class Diagrams

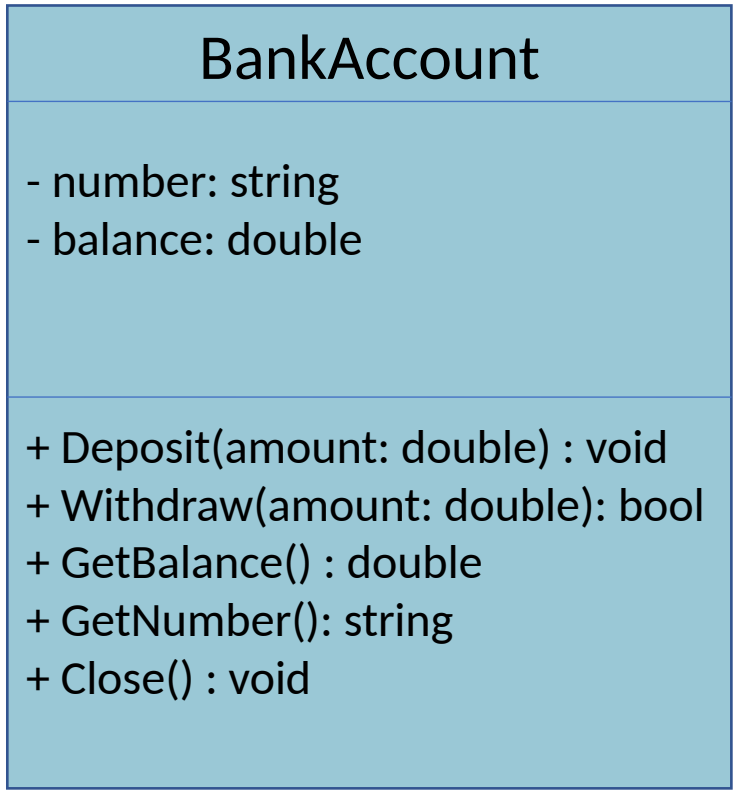
UML (Unified Modelling Language)



Class Diagrams

– symbol: private access modifier

+ symbol: public access modifier

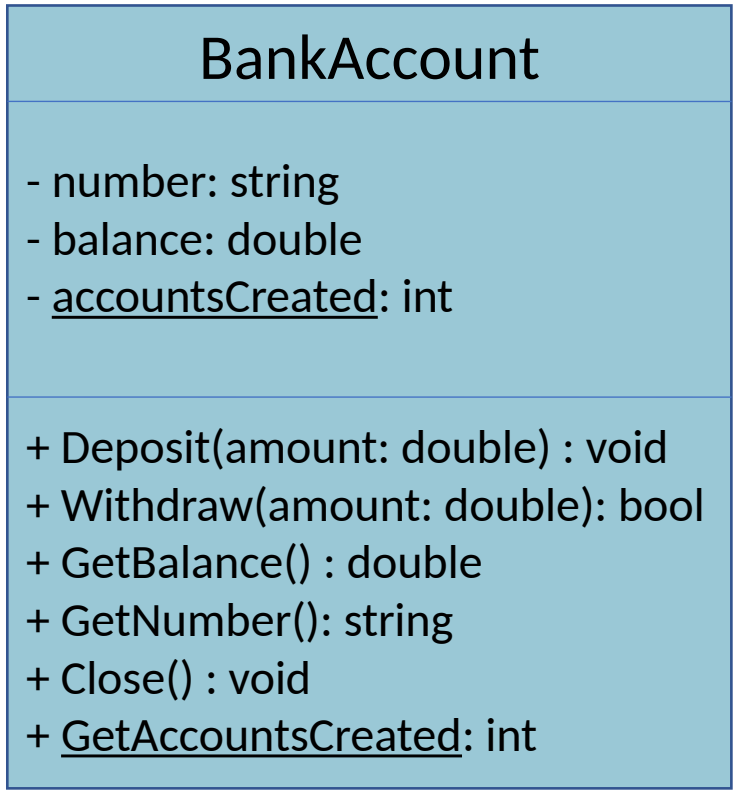


State

Behaviour

Class Diagrams

underlined defines static members

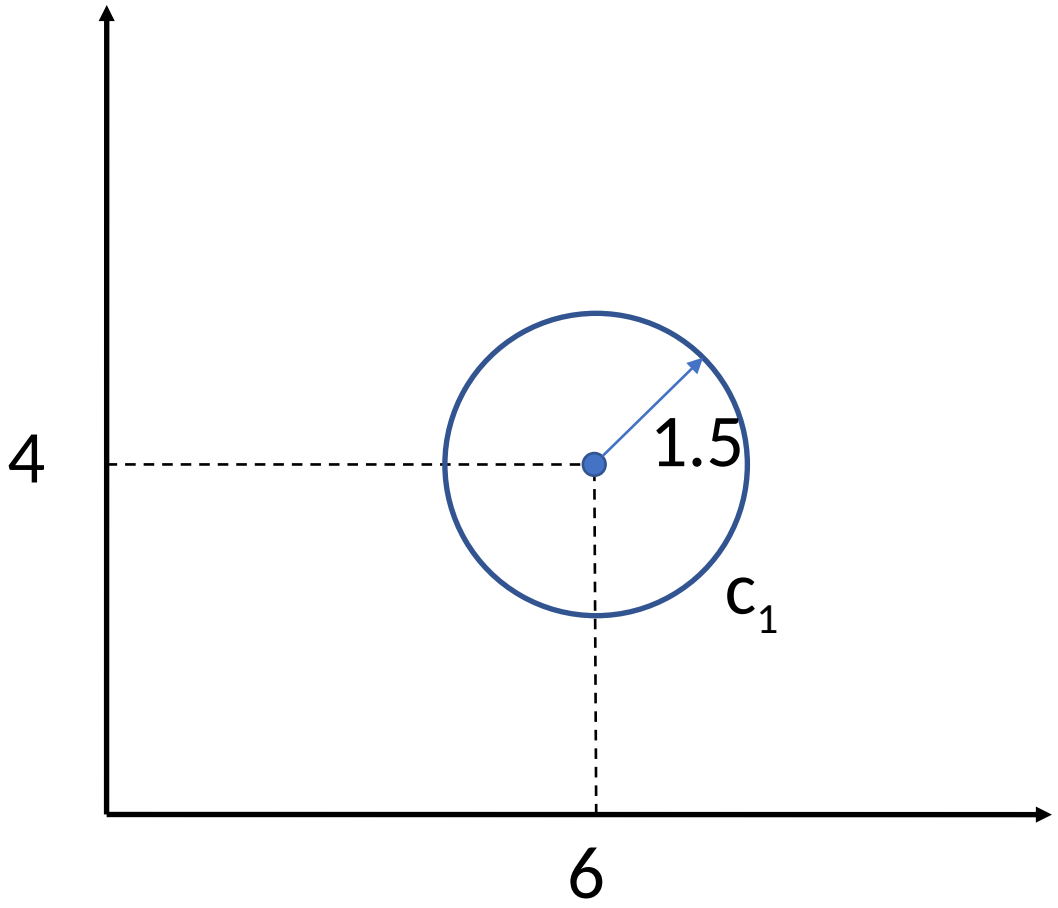


State

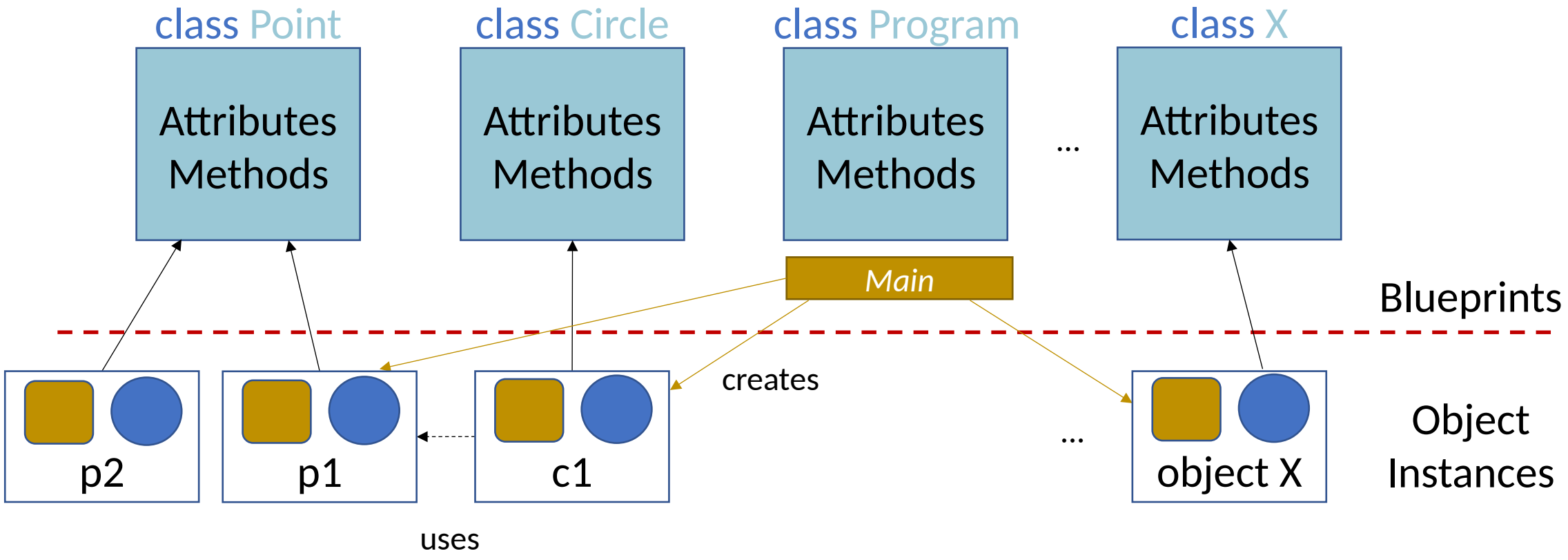
Behaviour

Relationships between Objects

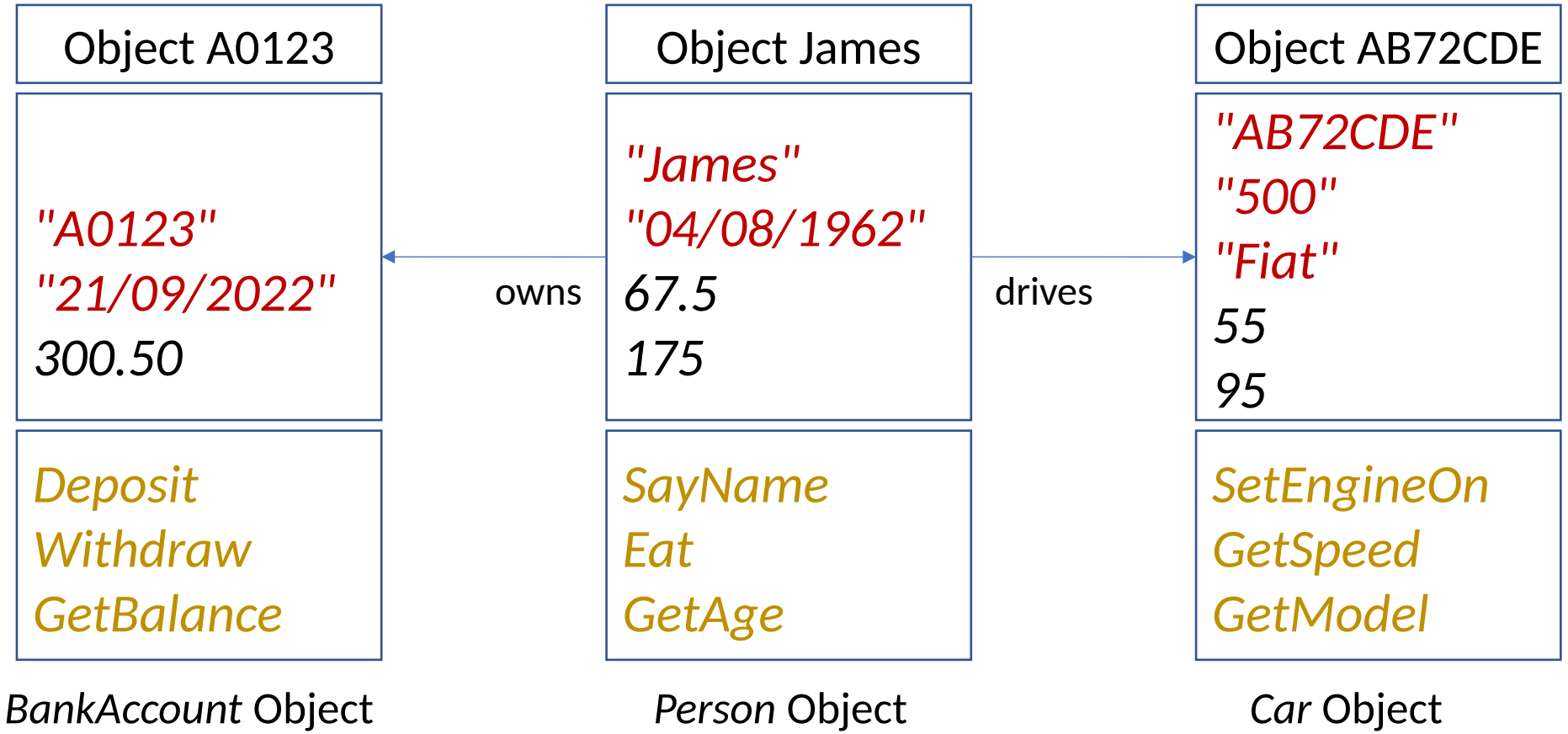
```
Point p1 = new Point(6, 4);
double r1 = 1.5;

Circle c1 = new Circle(p1, r1)
c1.Area();
```



Relationships between Objects



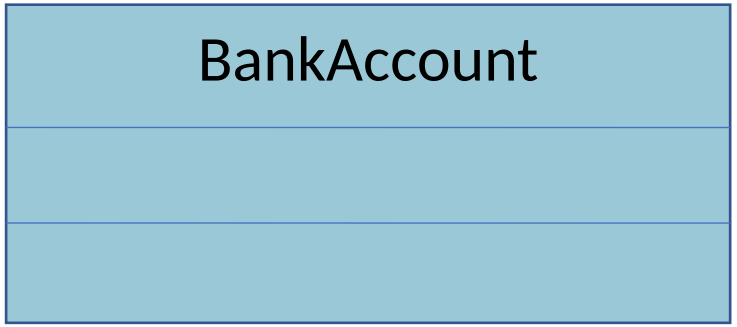
Objects Relationships

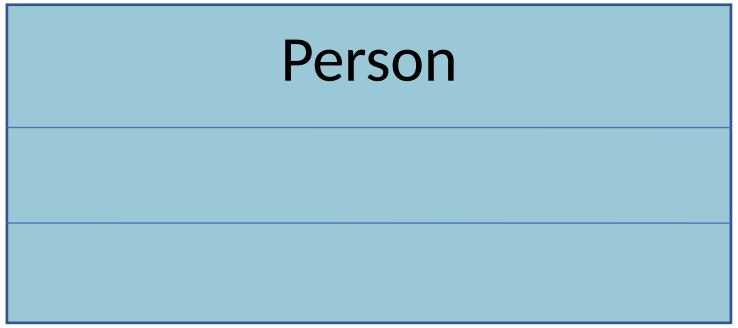


Relationships between Objects

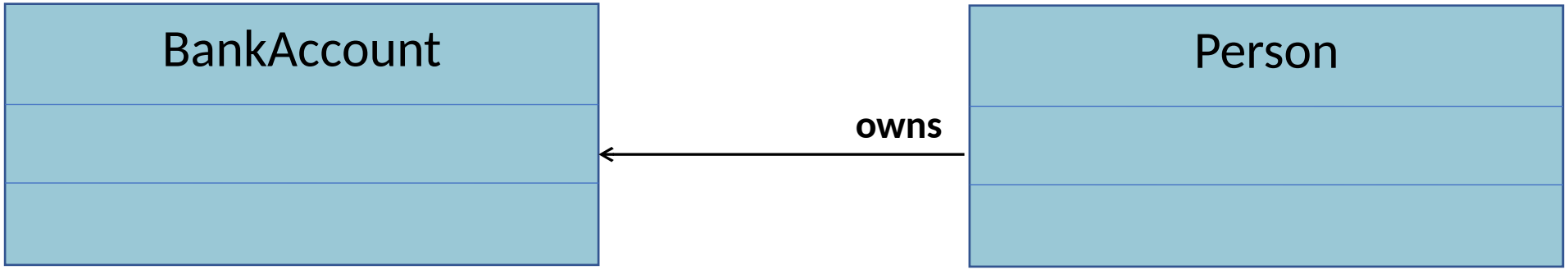
- In the real world one class instance (object) can have a type of relationship with another class instance
- We call it: **association**

Class Diagrams: Association



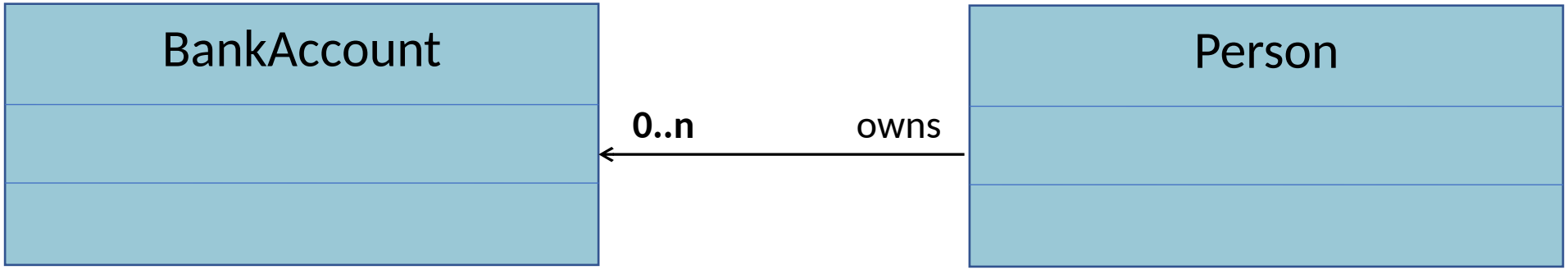


Class Diagrams: Unidirectional Association



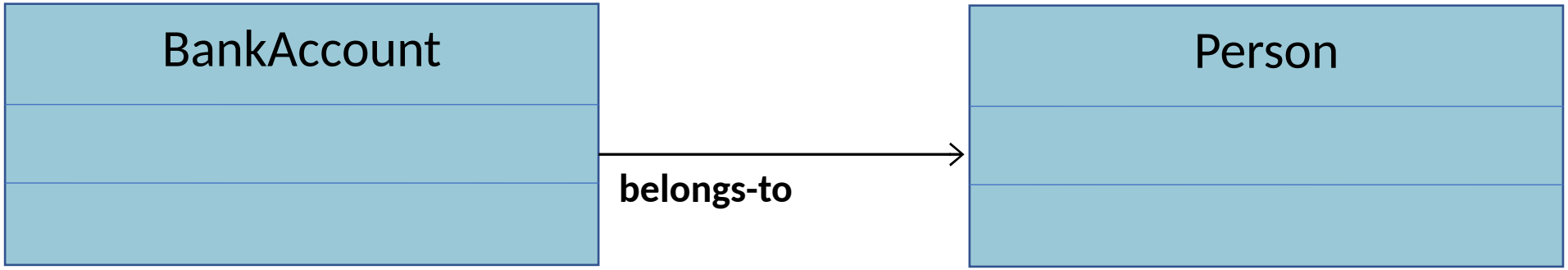
association name

Class Diagrams: Unidirectional Association



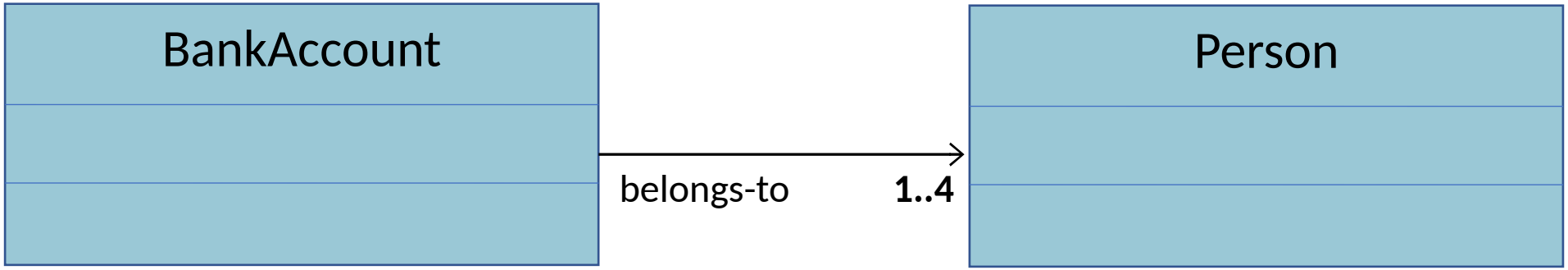
association multiplicity

Class Diagrams: Unidirectional Association



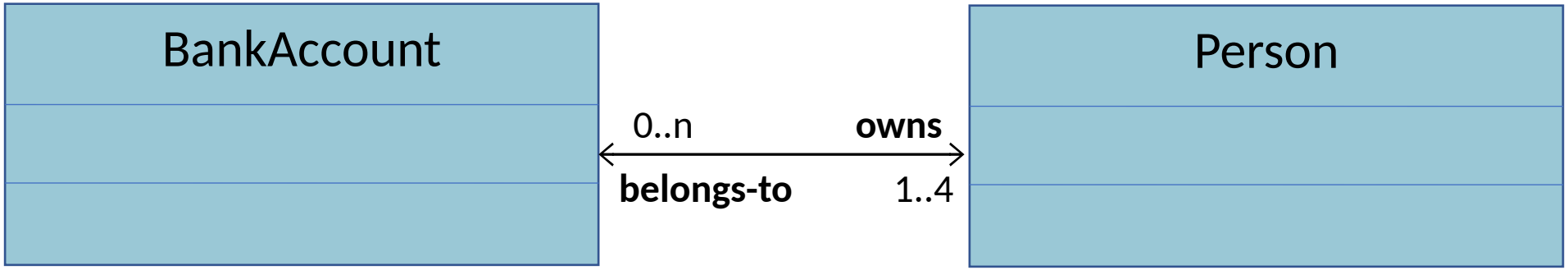
association name

Class Diagrams: Unidirectional Association

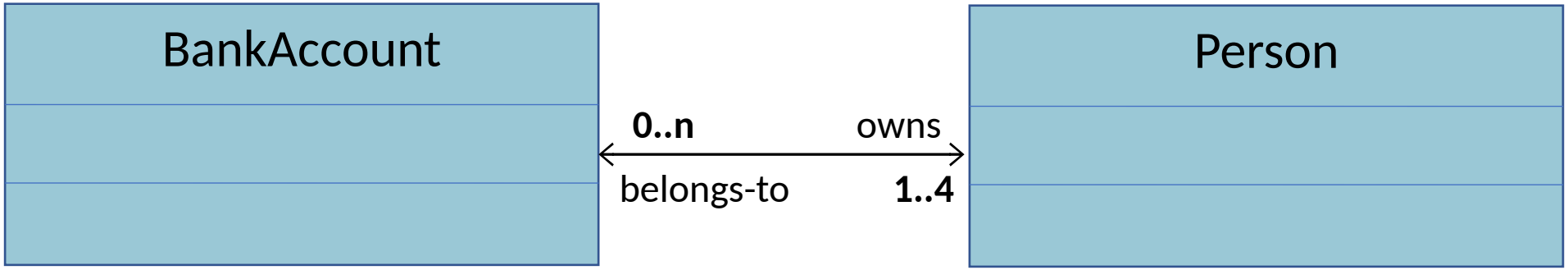


association multiplicity

Class Diagrams: Bidirectional Association



Class Diagrams: Bidirectional Association



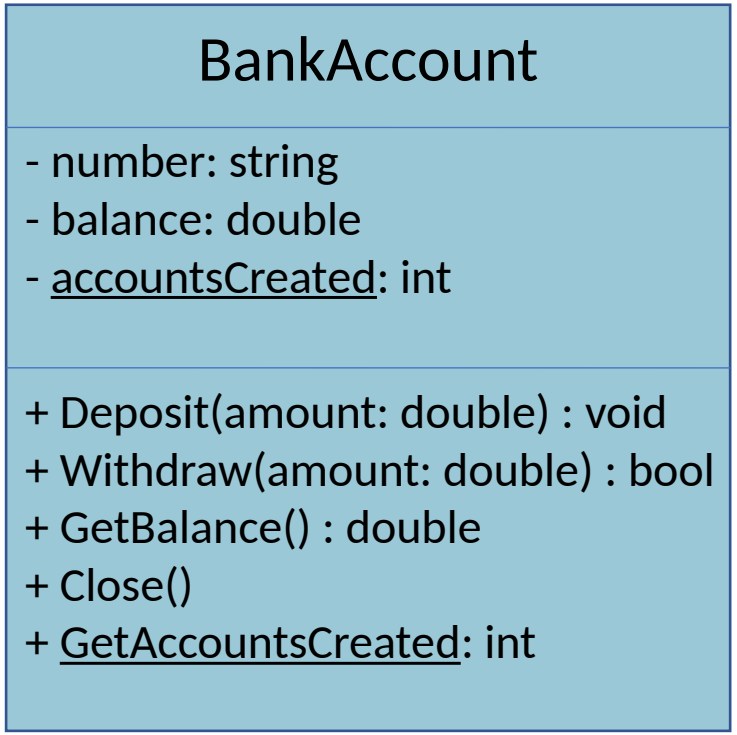
association multiplicity

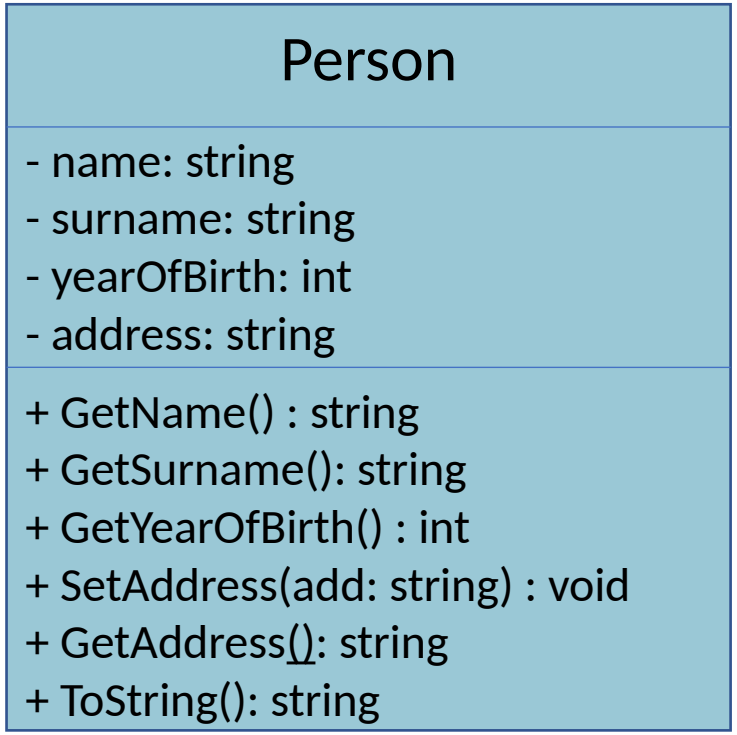
Class Diagrams: Association

- Depending on the problem, an association can be modelled in a different (simplified) way
- Example:

one BankAccount belongs-to one Person

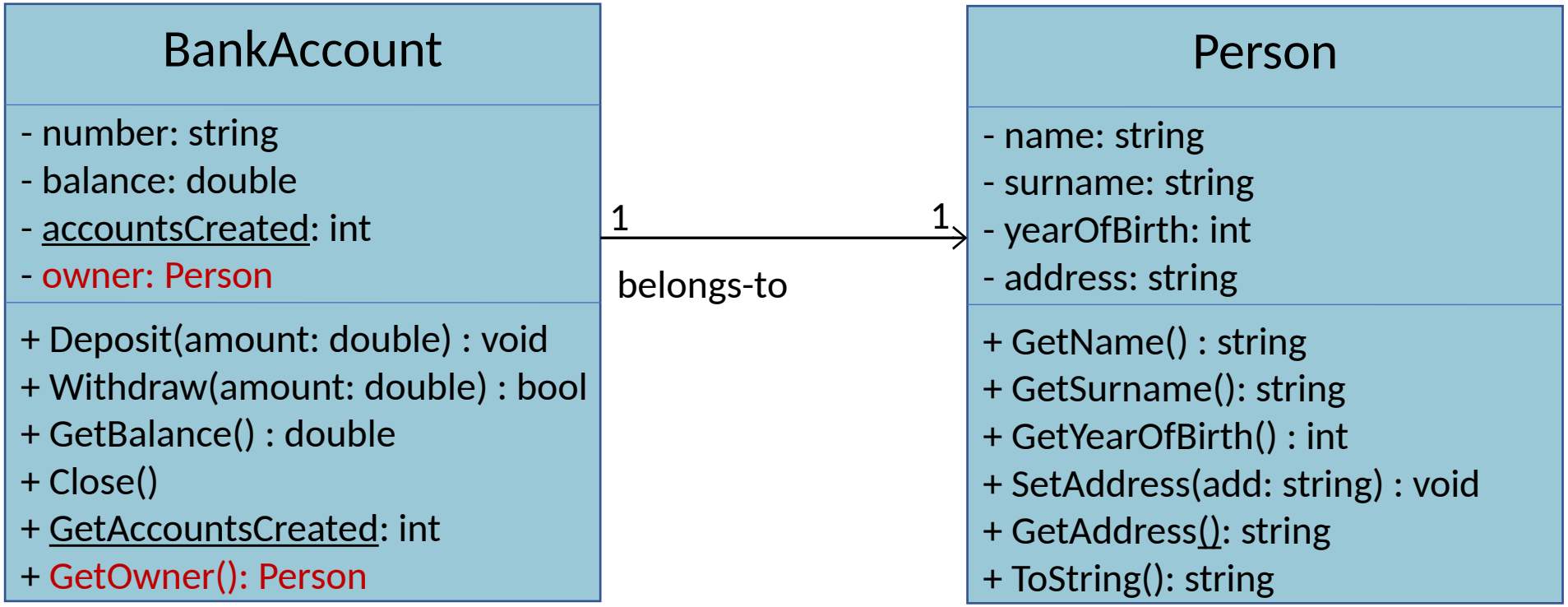
Class Diagrams: Association Example





Association: *one BankAccount belongs-to one Person*

Class Diagrams: Association Example



Association: *one BankAccount belongs-to one Person*

constructors have not been represented in this diagram

Association Example: Implementation

```
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



}
```

Association Example: Implementation

```
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



}
```

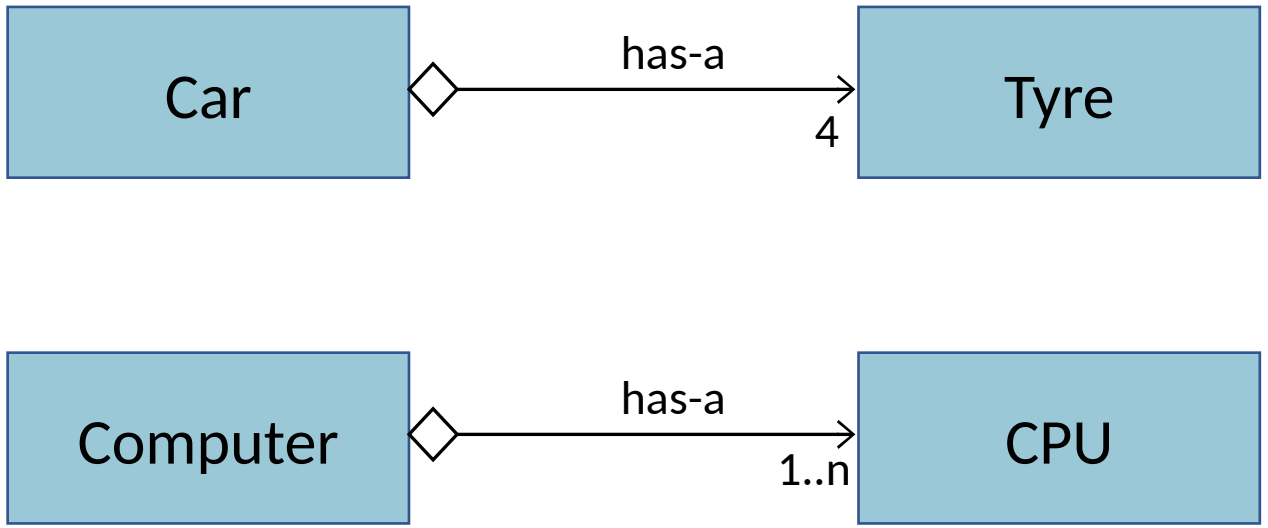
```
class BankAccount
{
    private string number;
    private double balance;
    private Person owner; // implements the association
    private static int accountsCreated = 0;

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

    // other methods of BankAccount

    public string GetOwner() {
        return owner;

}
```

Question

- How do we implement the association:

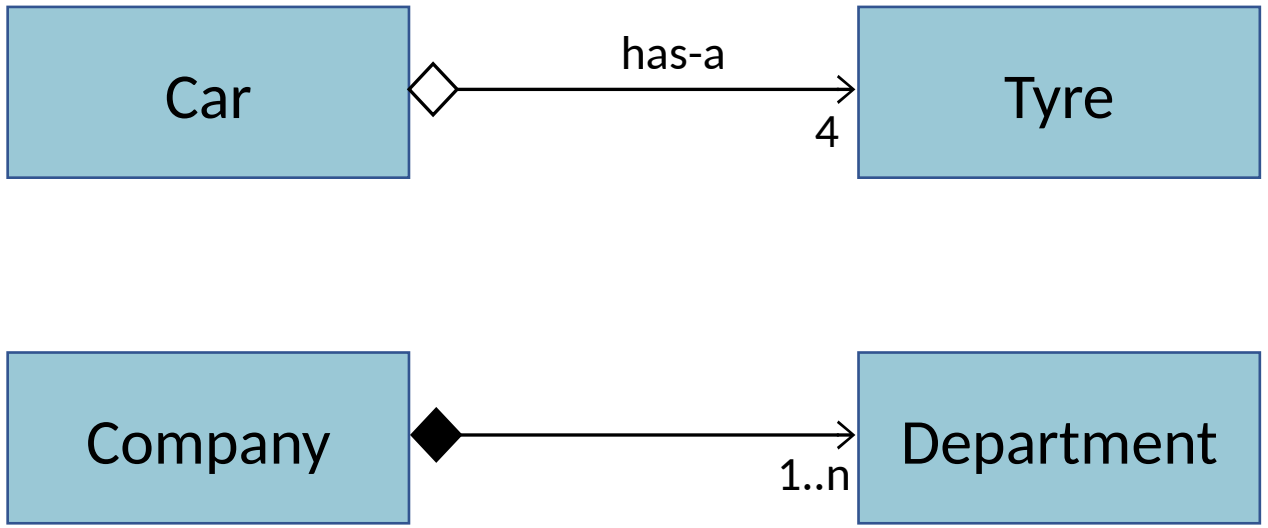
one Person owns one BankAccount

Aggregation: examples



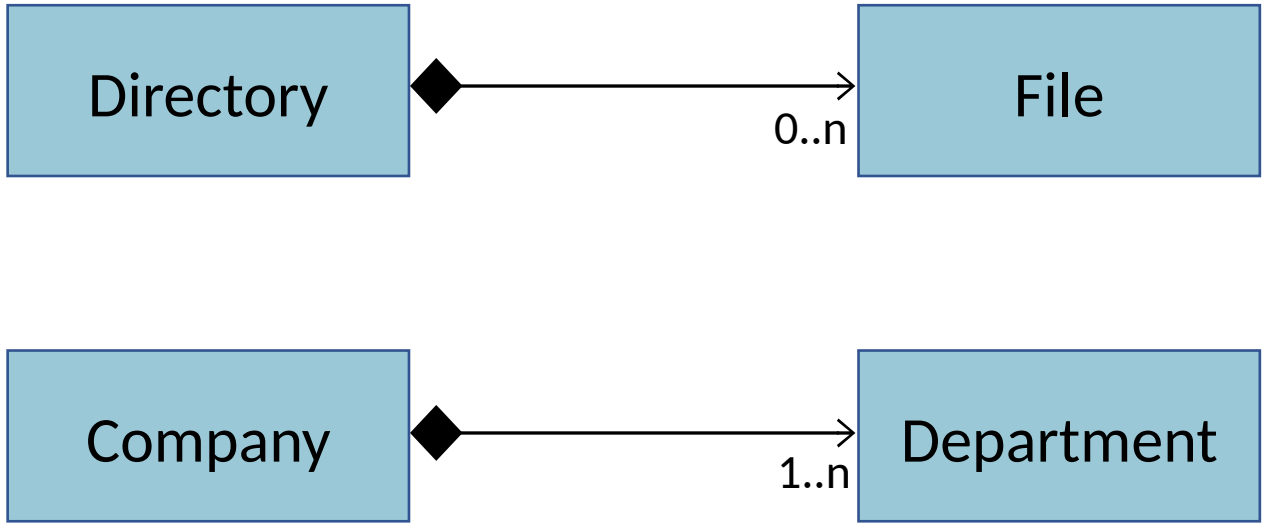
Aggregation: model the *has-a* relationship in which one class represents a larger thing (the whole), which consists of smaller things (the parts)

Question



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

Composition: examples



Composition: is a form of aggregation, with strong ownership and coincident lifetime as part of the whole

Composition: implementation 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 by the Directory object

No references are available elsewhere

The File object lives and dies with the Directory

Outline

- More on UML Class Diagrams: Object Relationships
- Class Relationships: Generalisation and Inheritance

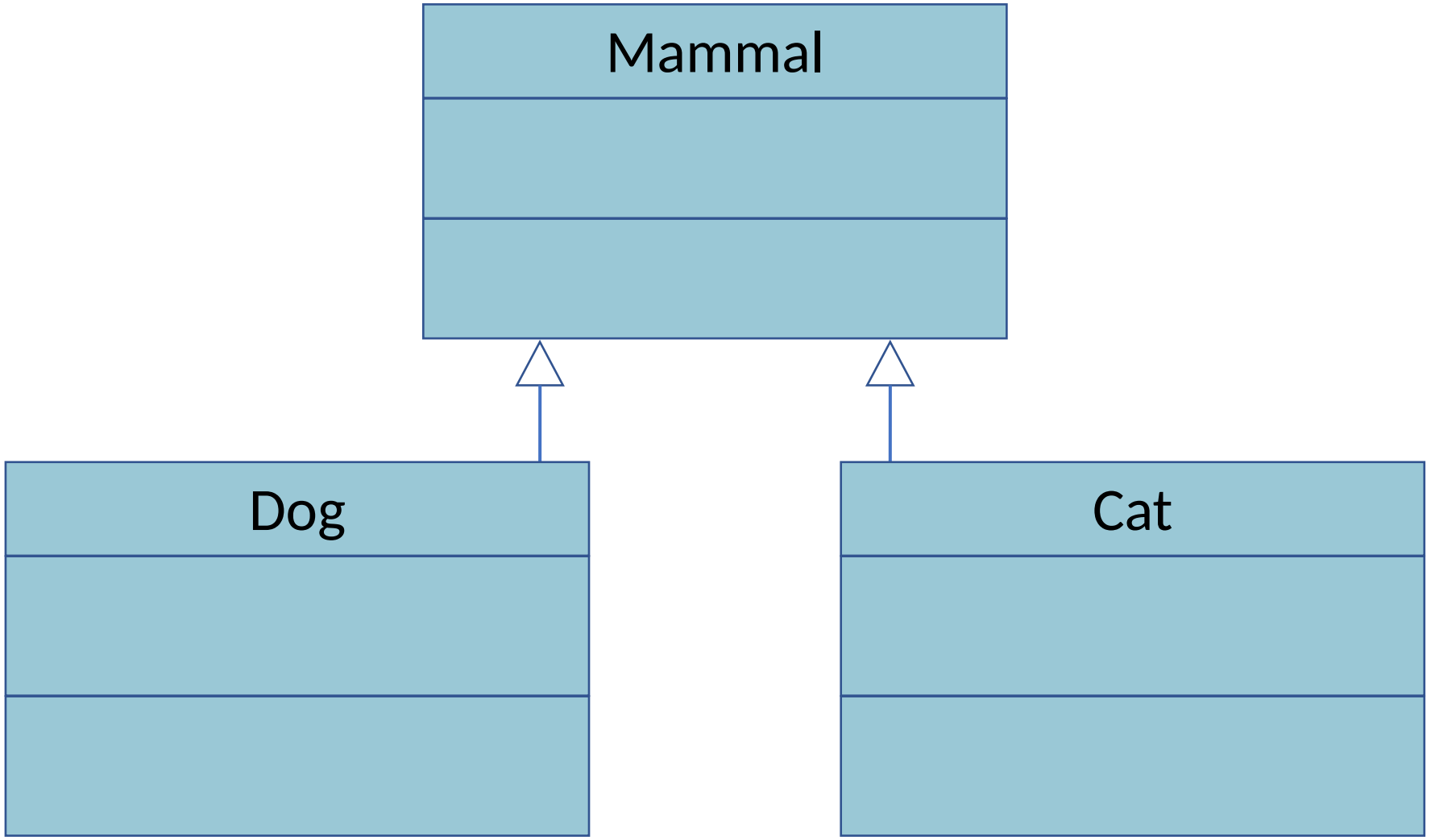
Relationship between classes

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

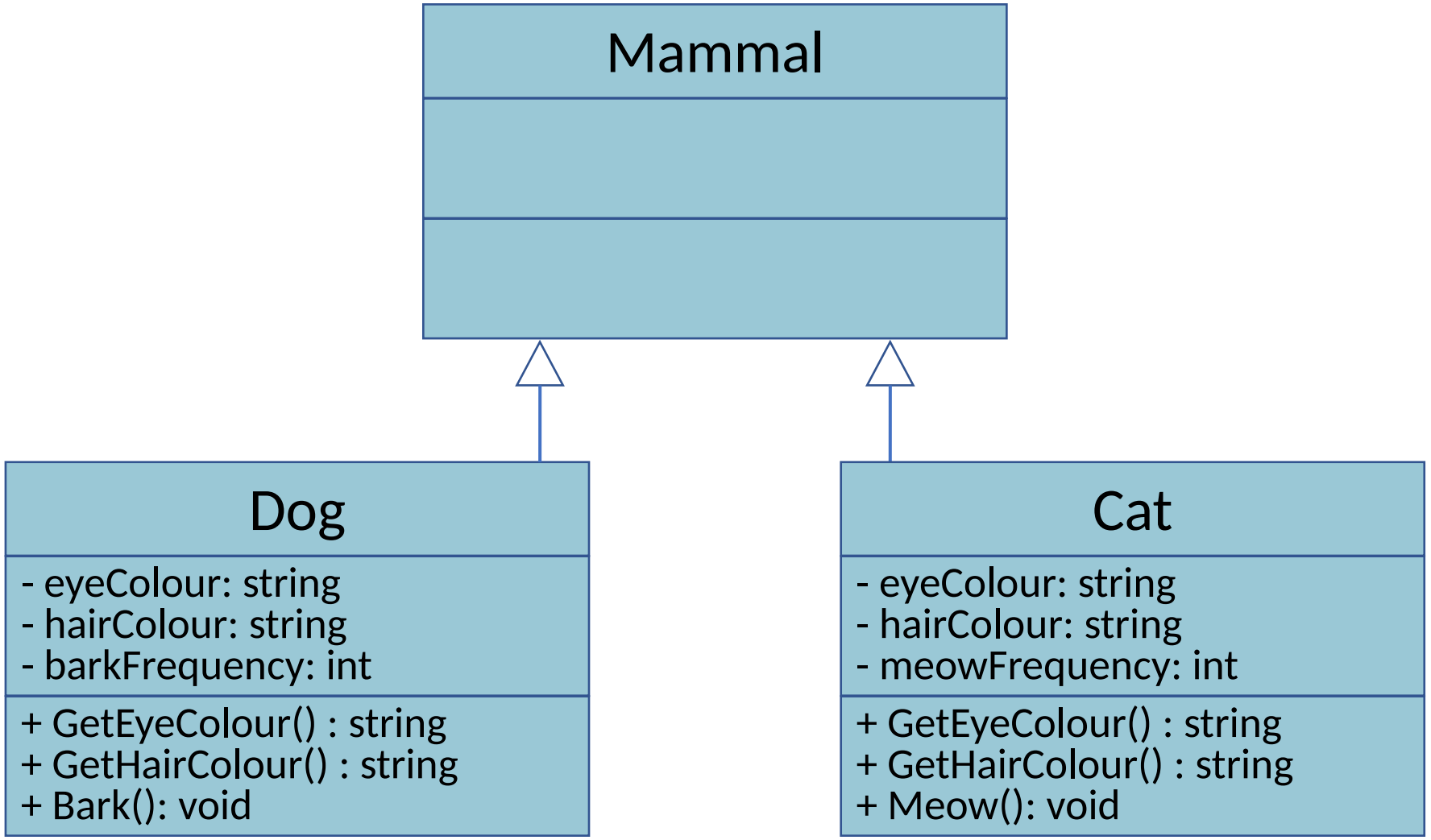
Relationship between classes: example

- 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*

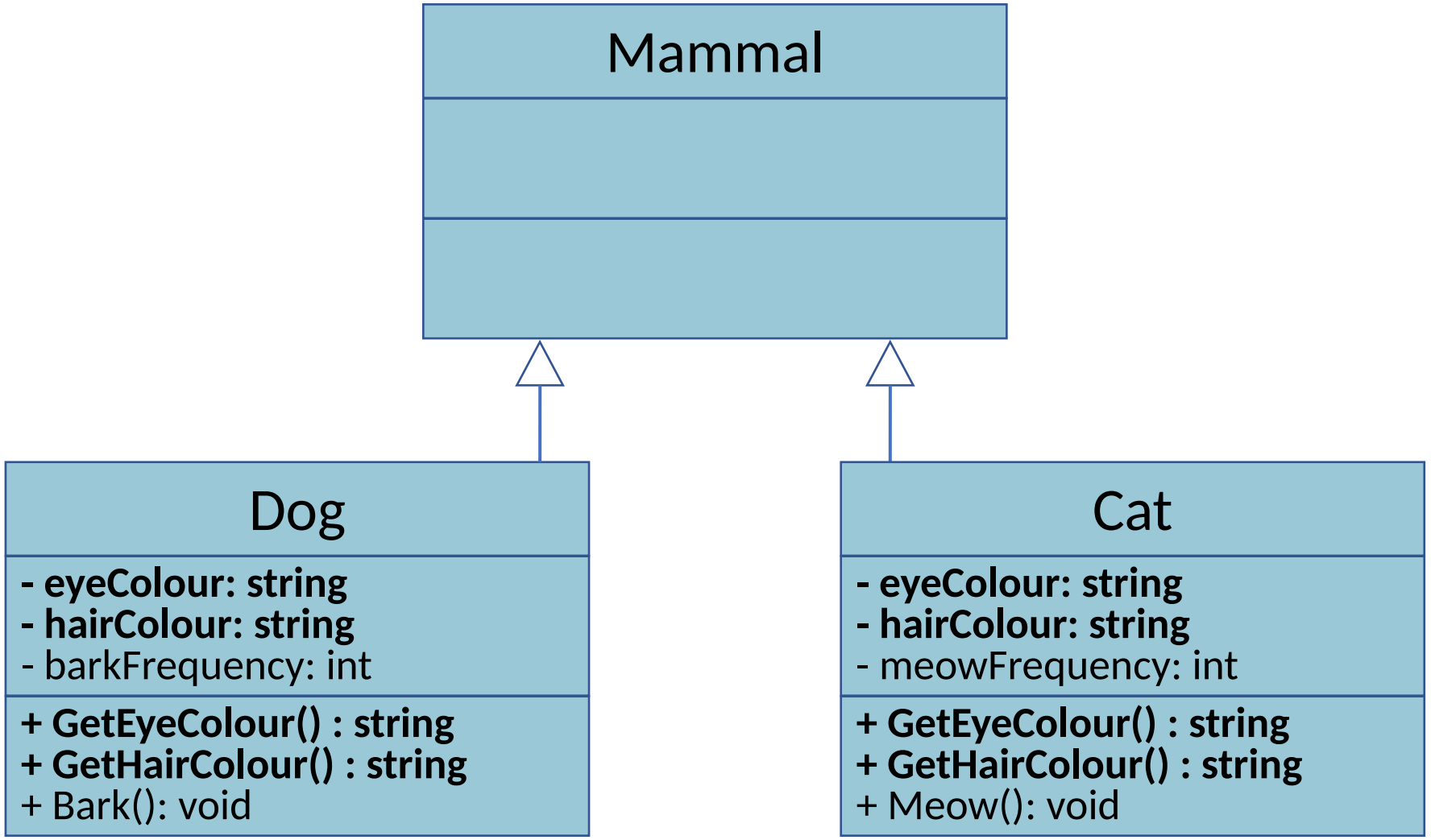
Generalisation relationship



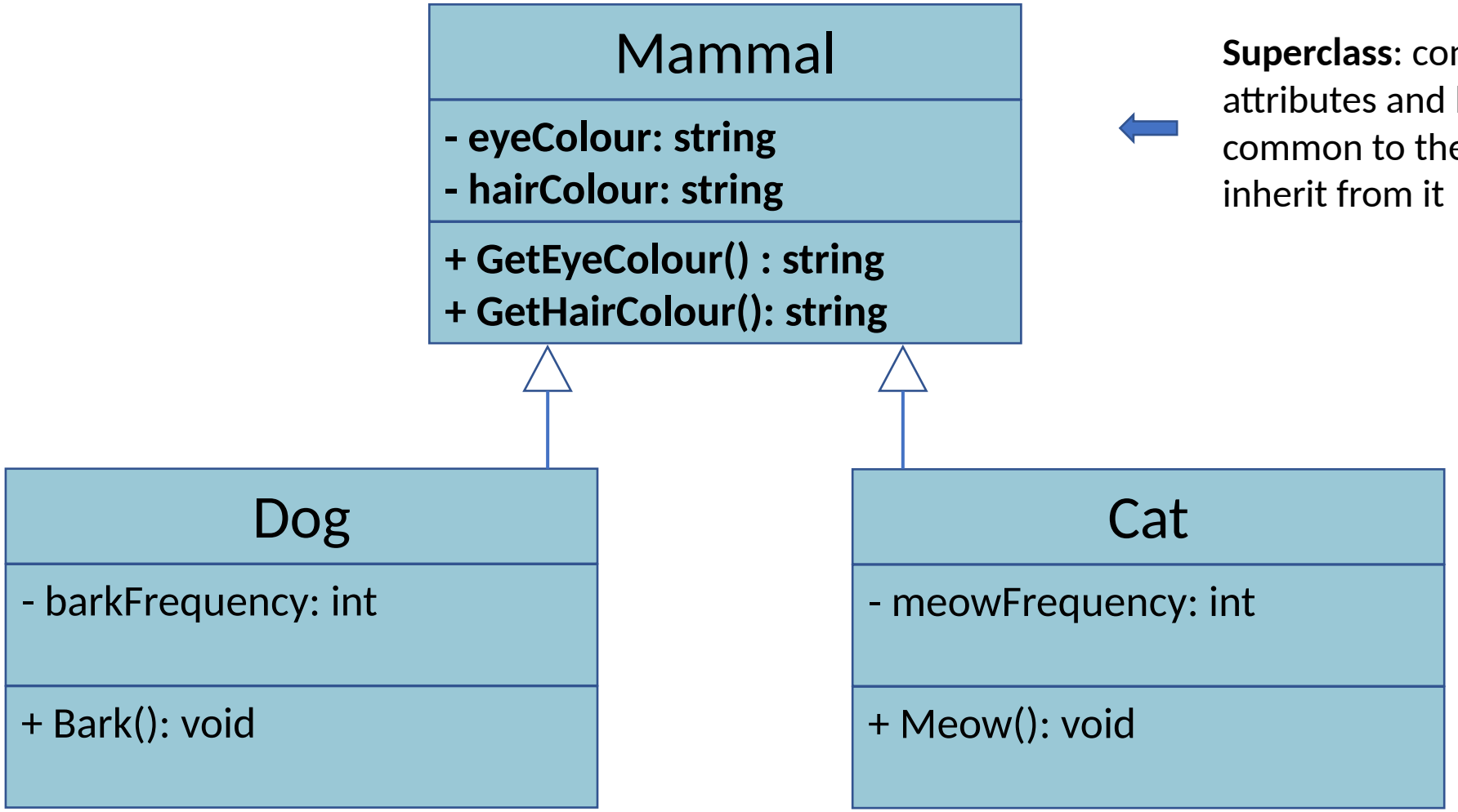
Generalisation relationship: inheritance



Generalisation relationship: inheritance

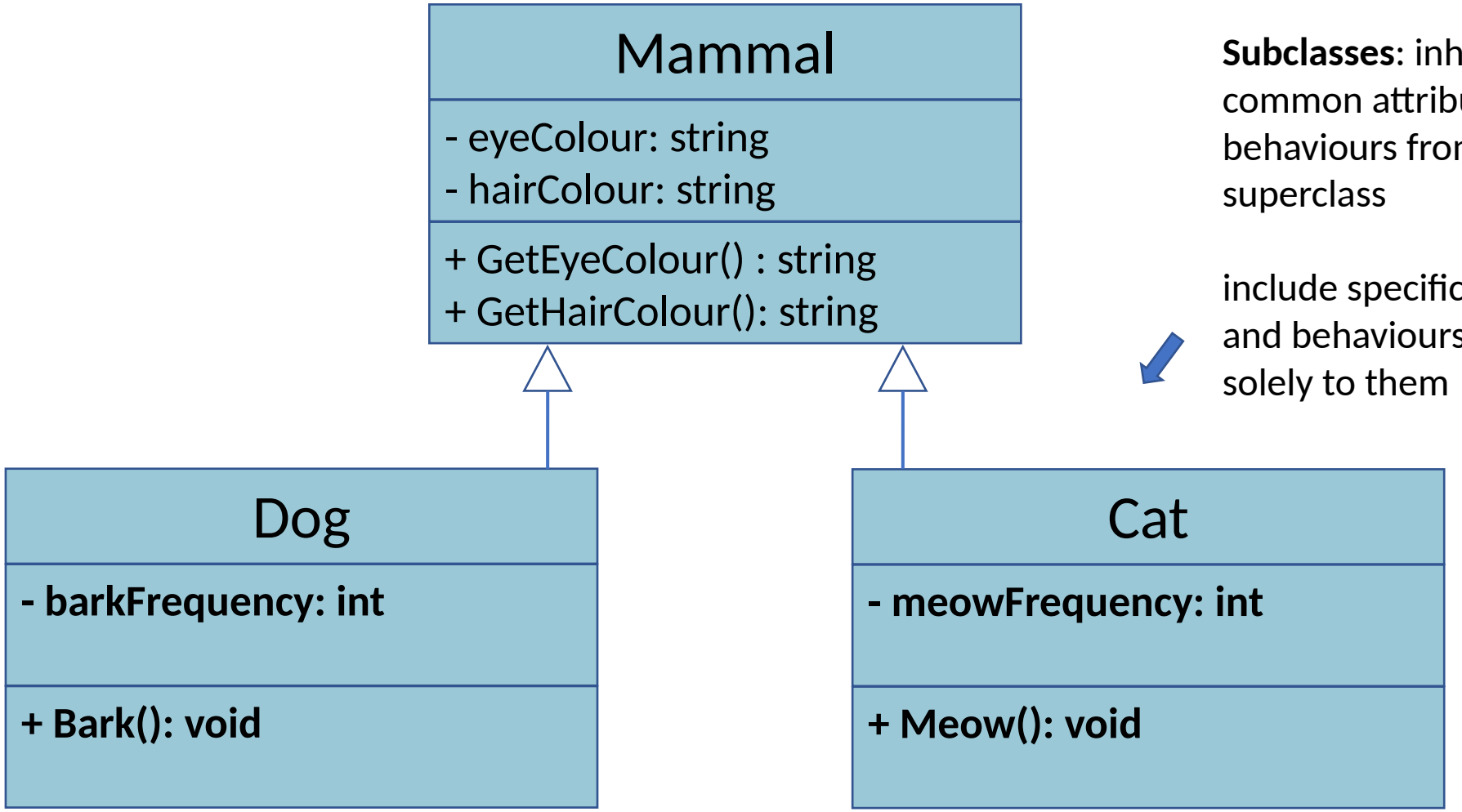


Generalisation relationship: inheritance



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

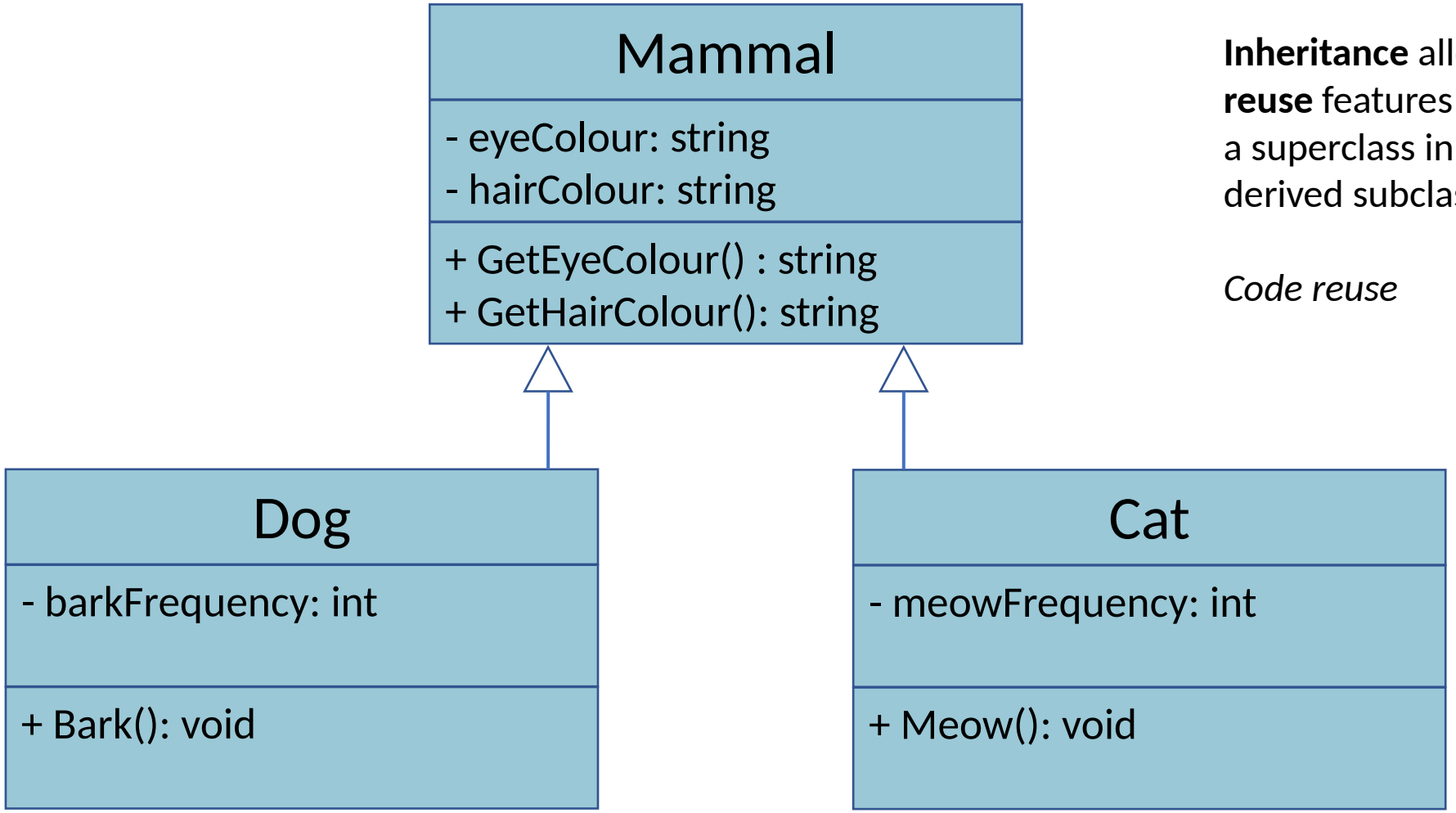
Generalisation relationship: inheritance



Subclasses: inherit common attributes and behaviours from the superclass

include specific attributes and behaviours that pertain solely to them

Generalisation relationship: inheritance



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

Code reuse

Implementation

```
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;
    }
}
```

Implementation

```
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;
    }
}
```

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

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

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

    public void Bark()
    {
        // uses barkFrequency
    }

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

Implementation

```
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 : 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

Implementation

```
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 : 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
}
```

Implementation

```
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 : 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

Private members and constructors

- A subclass **does not inherit**
  - The *private* members of its parent class
  - The *constructors* of the superclass

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

How?

Creating new Objects of the child classes

```
public class Program
{
    public static 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();
    }
}
```

Subclass constructor

```
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 : 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
}
```

Subclass constructor

```
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 : 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

Subclass constructor

```
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 : 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(...)*

Subclass constructor

```
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 : 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 *base(…)*

Subclass constructor

```
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 : Mammal
{
    int barkFrequency;

    public Dog(string ec, string hc, int bf)
        : base(ec, hc)
    {
        barkFrequency = bf;
    }

    public void Bark()
    {
        // uses barkFrequency
    }

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

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

Subclass constructor

```
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 : Mammal
{
    int barkFrequency;

    public Dog(string ec, string hc, int bf)
        : base(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

Subclass constructor

```
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 : Mammal
{
    int barkFrequency;

    public Dog(string ec, string hc, int bf)
        : base(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

Subclass constructor

```
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 : Mammal
{
    int barkFrequency;

    public Dog(string ec, string hc, int bf)
        : base(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

protected access modifier

So far we used:

- private
- internal
- public

protected access modifier

So far we used:

- private: access to members restricted to the **same** class
- internal: access to members allowed to classes of the **same assembly**
- public: access to members allowed to **any external** class

protected access modifier

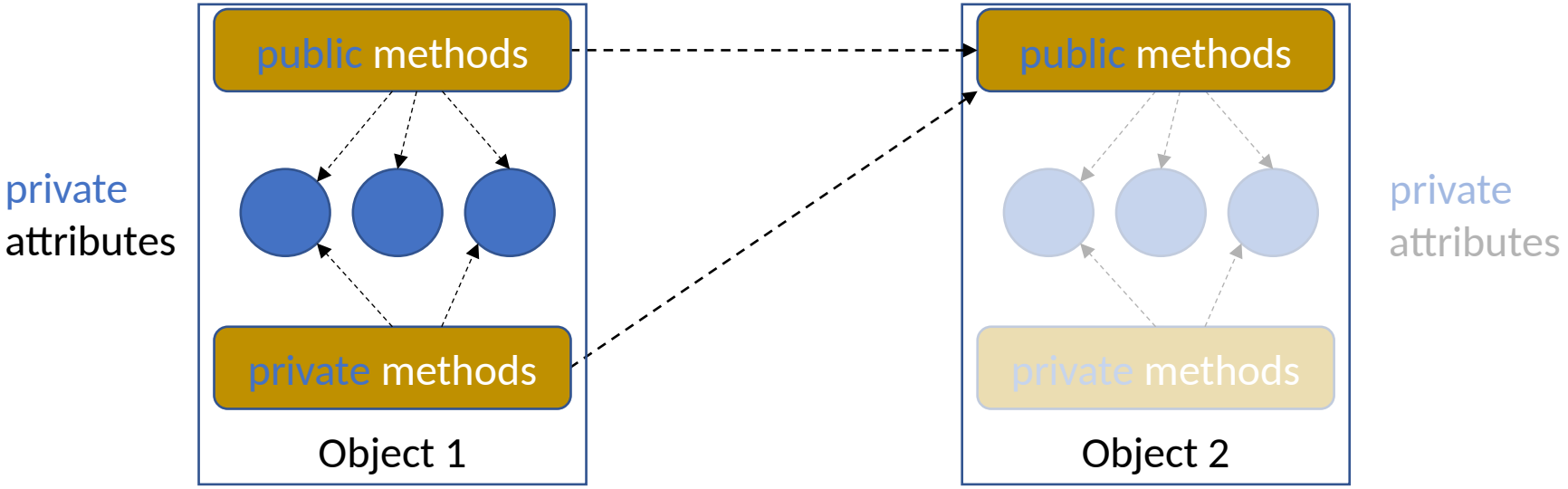
So far we used:

- private: access to members restricted to the **same** class
- internal: access to members allowed to classes of the **same assembly**
- public: access to members allowed to **any external** class

protected: members can be accessed from the **same** class, and from **any subclass**

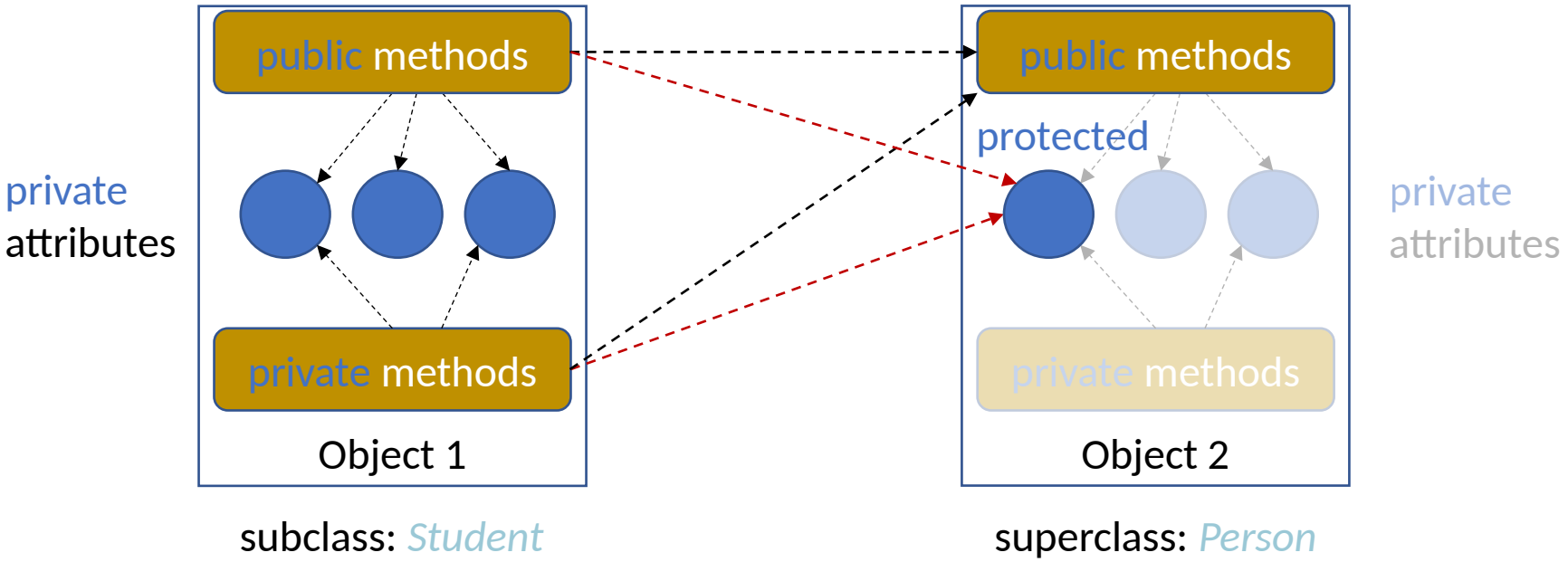
Encapsulation and interfaces (reminder)

- 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**



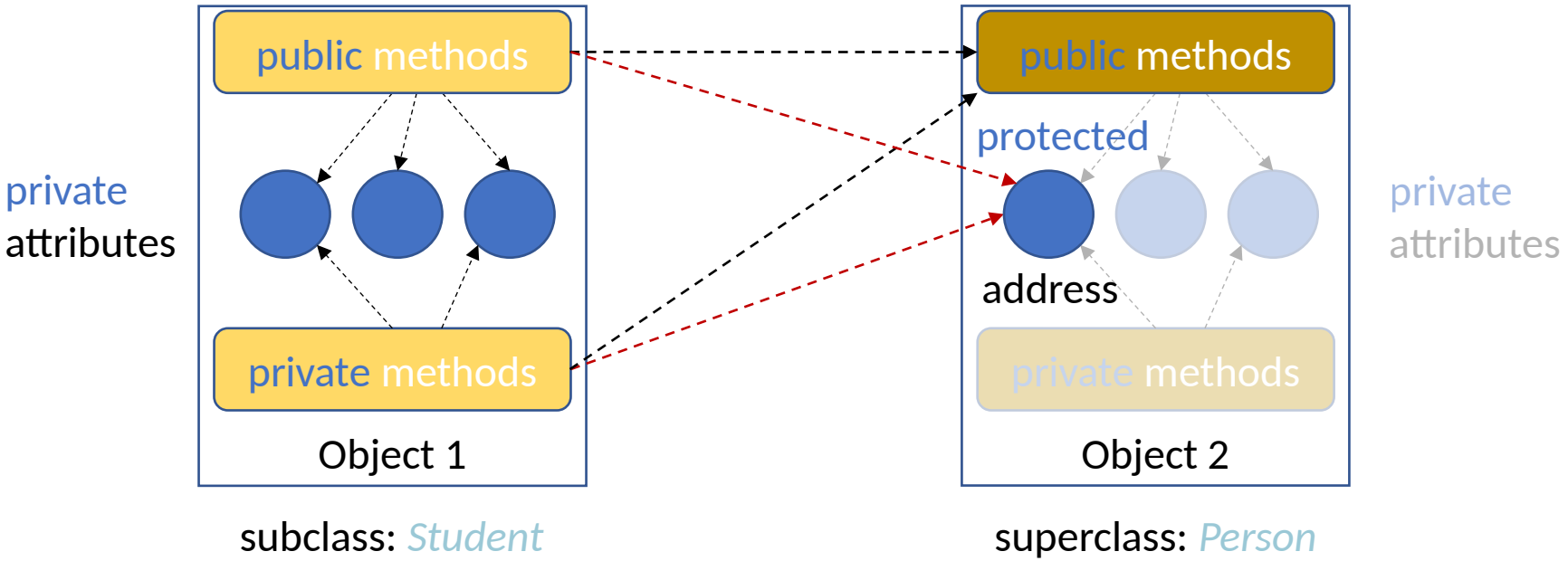
Encapsulation and interfaces: protected

- A protected attribute could make the **encapsulation weaker**



Encapsulation and interfaces: protected

- A protected attribute could make the **encapsulation weaker**
- Example: the change of the **address** attribute from string to *Address* in the *Person* superclass could have required changes to the *Student* subclass



protected access modifier

So far we used:

- private: access to members restricted to the **same** class
- internal: access to members allowed to classes of the **same assembly**
- public: access to members allowed to **any external** class

protected: members can be accessed from the same class, and from any subclass

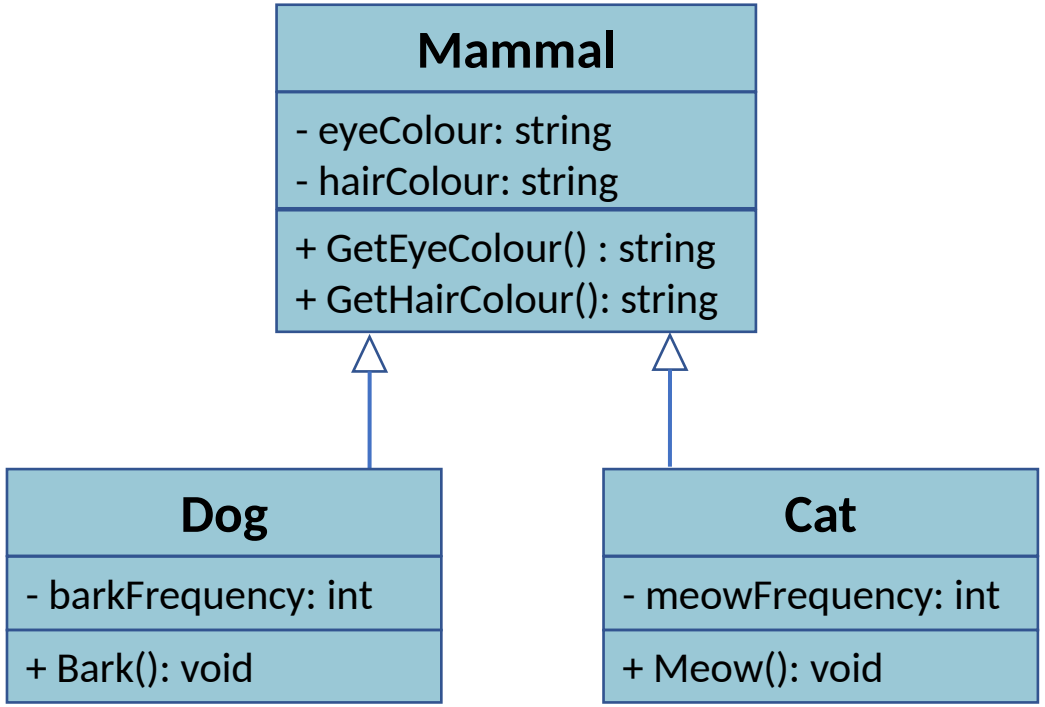
best practice: use private attributes and define public or protected *getter* and *setter* methods

Inheritance

- When there is a **generalisation** relationship: a *subclass* “**is-a-kind-of**” a *superclass*
- The *subclass* **inherits** the **attributes** and **methods** of the *superclass*

- *How can this help with the development of code for new classes*?

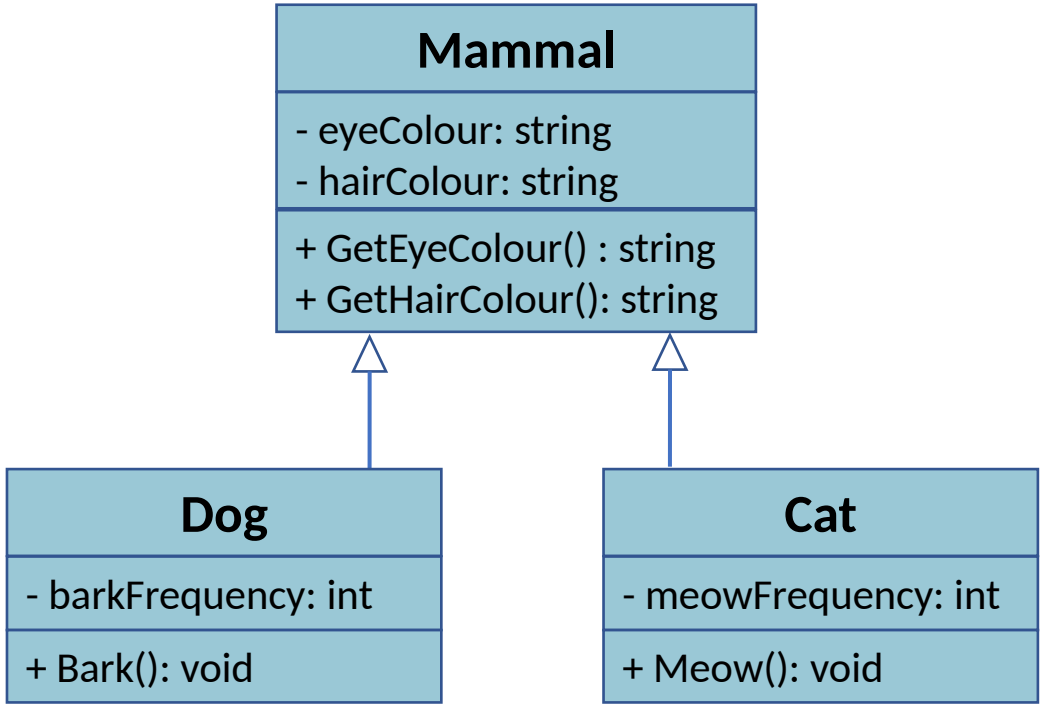
Inheritance



GoldenRetriever class?

Create it from scratch or…

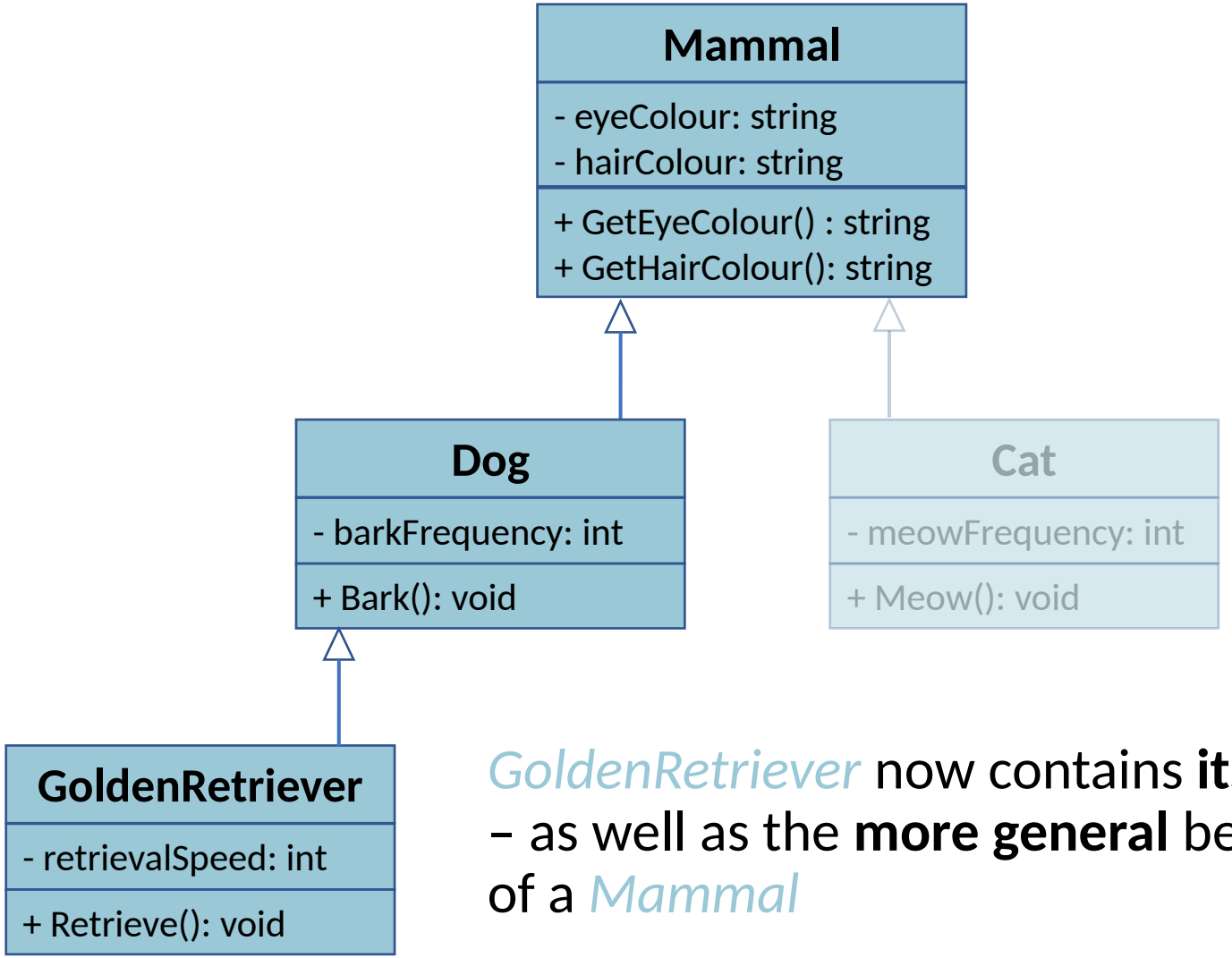
Inheritance



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

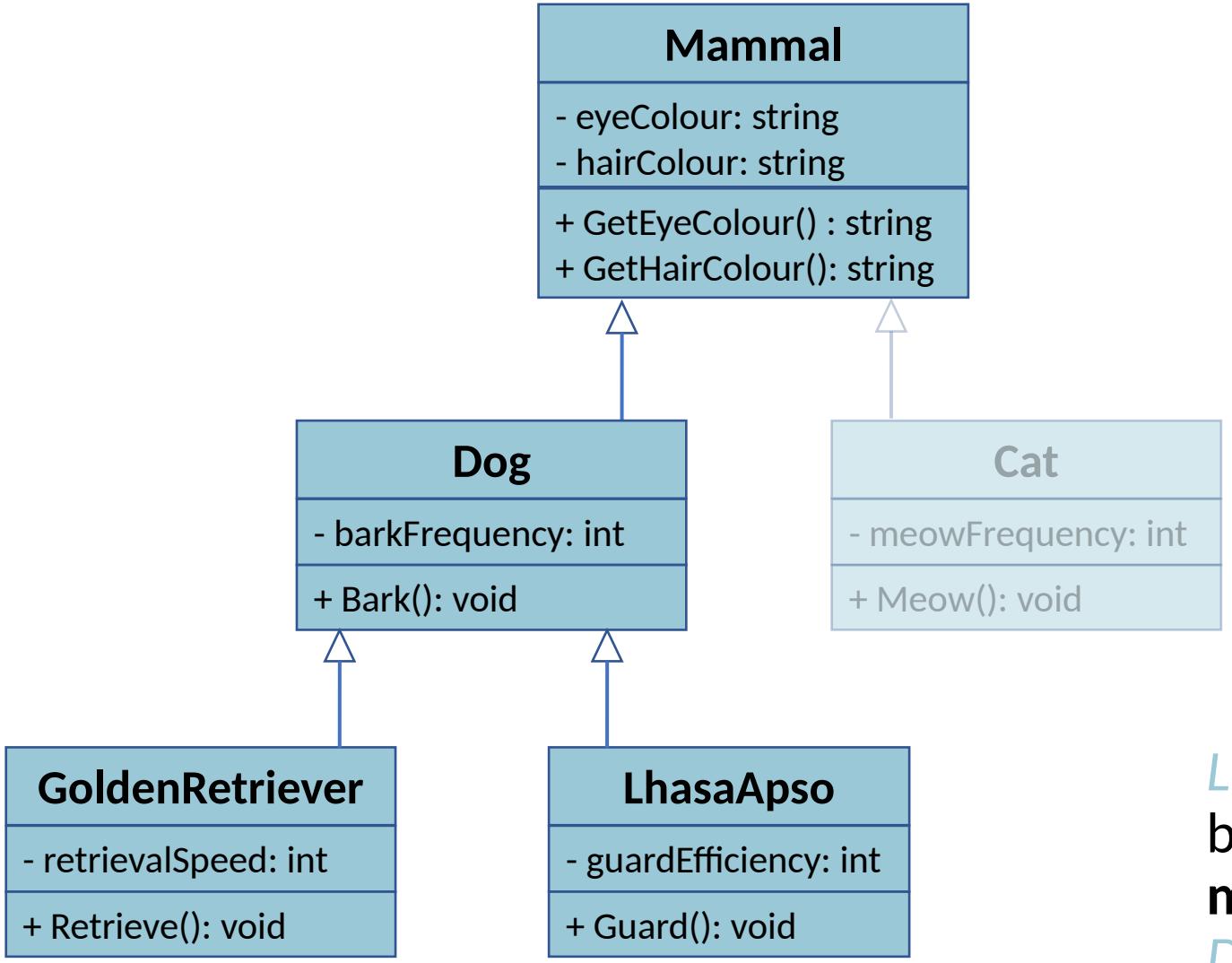
Hence general *attributes* and *behaviours* can be **inherited** from *Dog*

Inheritance



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

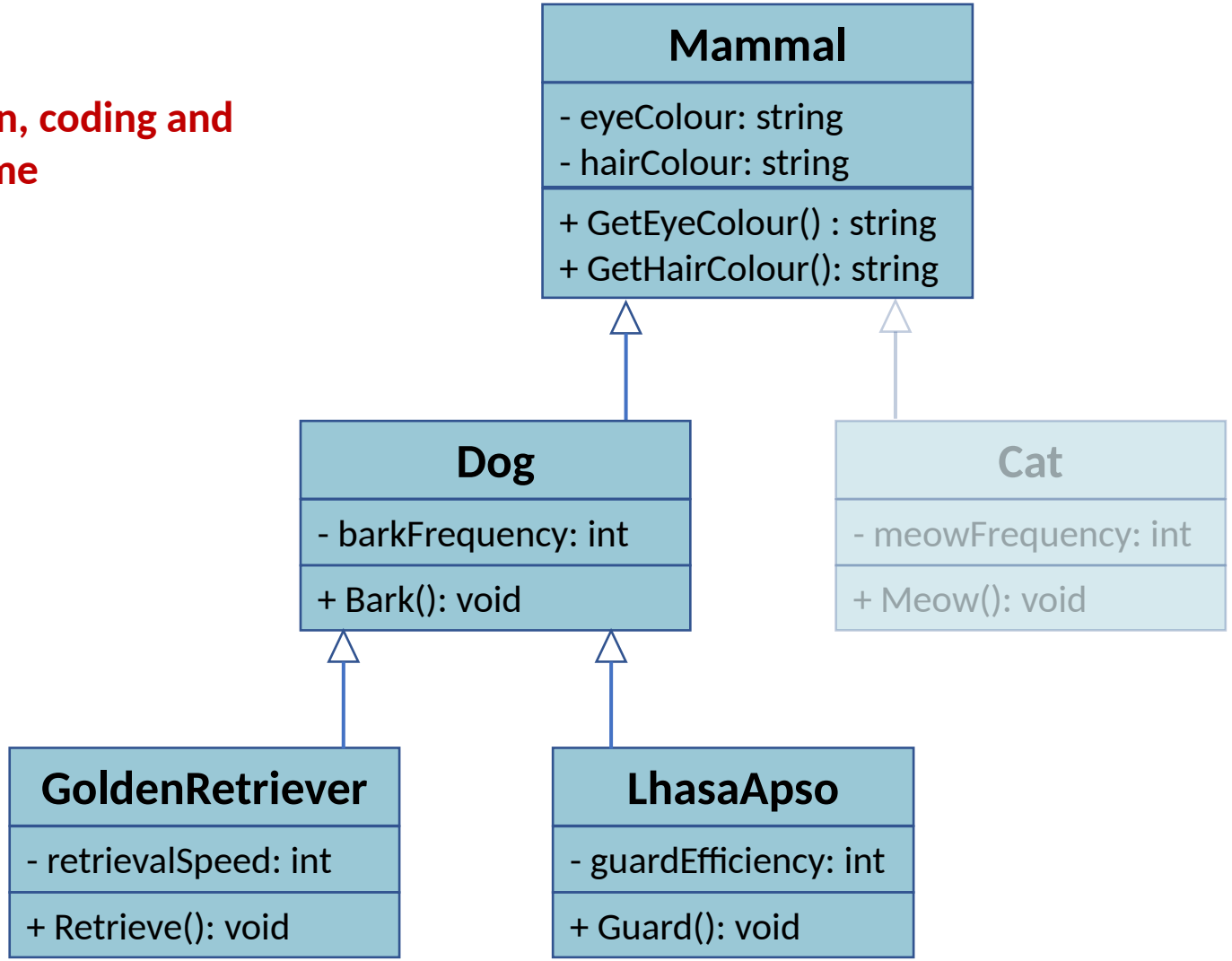
Inheritance



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

Inheritance: benefits

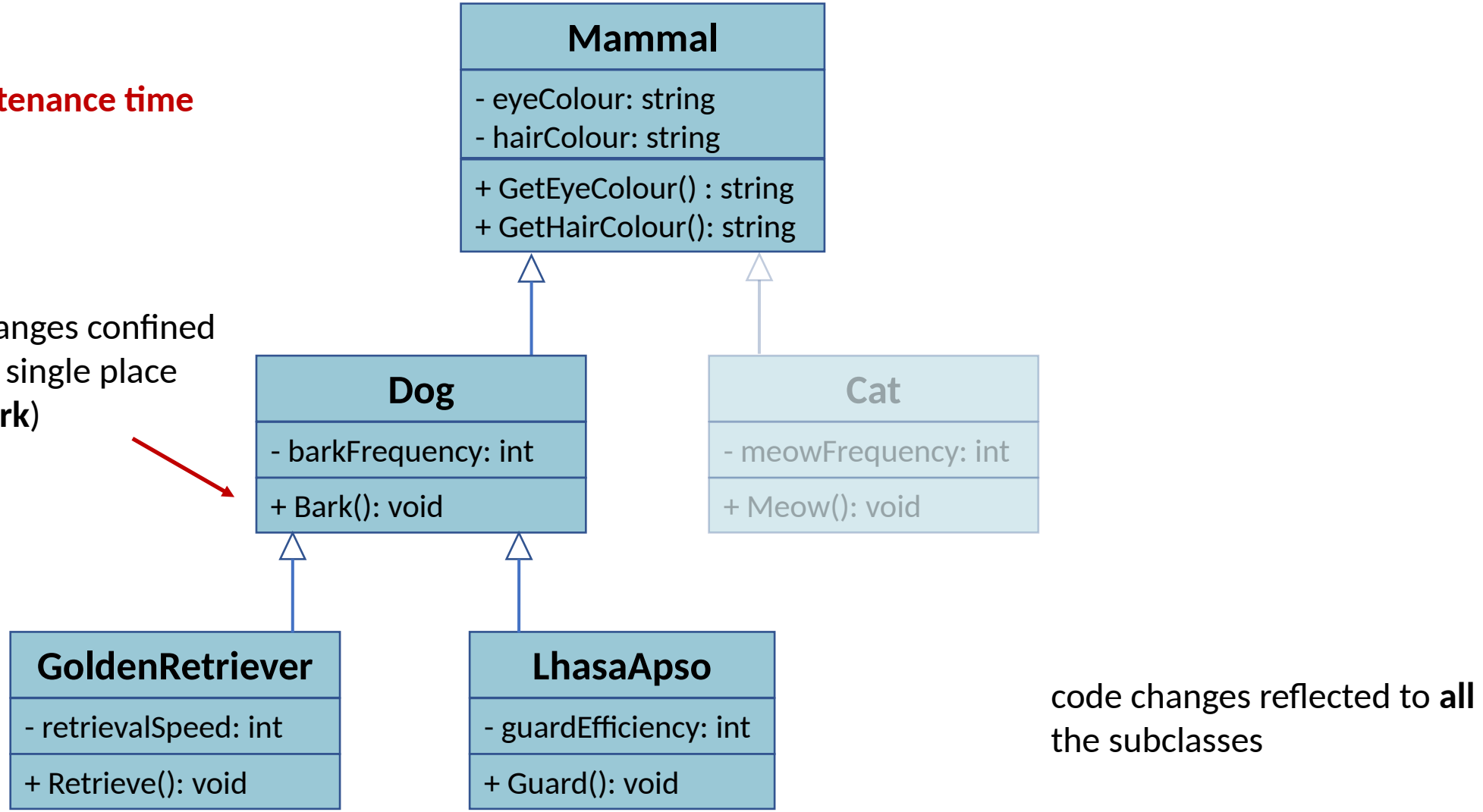
less design, coding and testing time



Inherited methods: *Bark*, *GetEyeColour* and *GetHairColour* are effectively **reused**

Inheritance: benefits

less maintenance time



code changes confined within a single place (e.g., **Bark**)

code changes reflected to **all** the subclasses

Inheritance: summary

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

Object-Oriented Programming (OOP) Principles

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

A *child* class can *inherit* and take advantage of the *attributes* and *methods* a parent class (superclass) defines.