7SENG011W Object Oriented Programming

More on design contracts: interfaces; Object class

Dr Francesco Tusa

Readings

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

- Programming C# 10
  - Chapter 6: Inheritance and Runtime Polymorphism
- Hands-On Object-Oriented Programming with C#
  - Chapter: Object Collaboration
- Object-Oriented Thought Process
  - Chapter 8: Frameworks and Reuse: Designing with Interfaces and Abstract Classes

Online

- Polymorphism
- abstract classes
- Interfaces
- sealed keyword
- Object class
- User-defined exceptions

Outline

- Design Contracts
    - Summary of abstract classes
    - Interfaces
- C# inheritance tree: Object class

Object-Oriented Programming (OOP) Principles

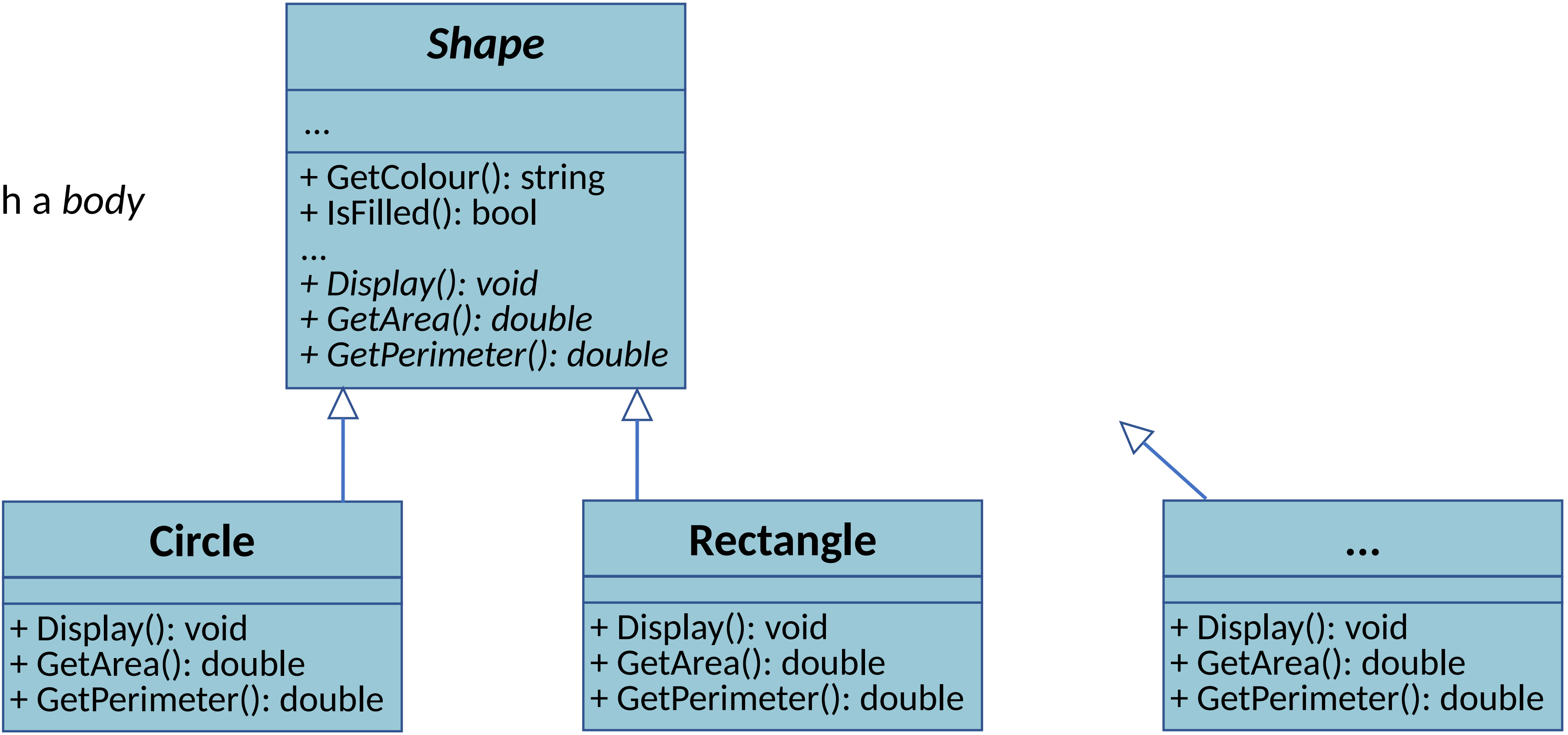
- **Abstraction**
- **Encapsulation**
- **Inheritance**
- **Polymorphism**

When classes are related via a *generalisation* relationship, objects of the *subclasses* can respond to the **same** *"message"* in **different** ways

Abstract classes: contract

Methods with a *body*

Inherited methods



Shape

...

+ GetColour(): string
+ IsFilled(): bool
...
+ Display(): void
+ GetArea(): double
+ GetPerimeter(): double

Circle

+ Display(): void
+ GetArea(): double
+ GetPerimeter(): double

Rectangle

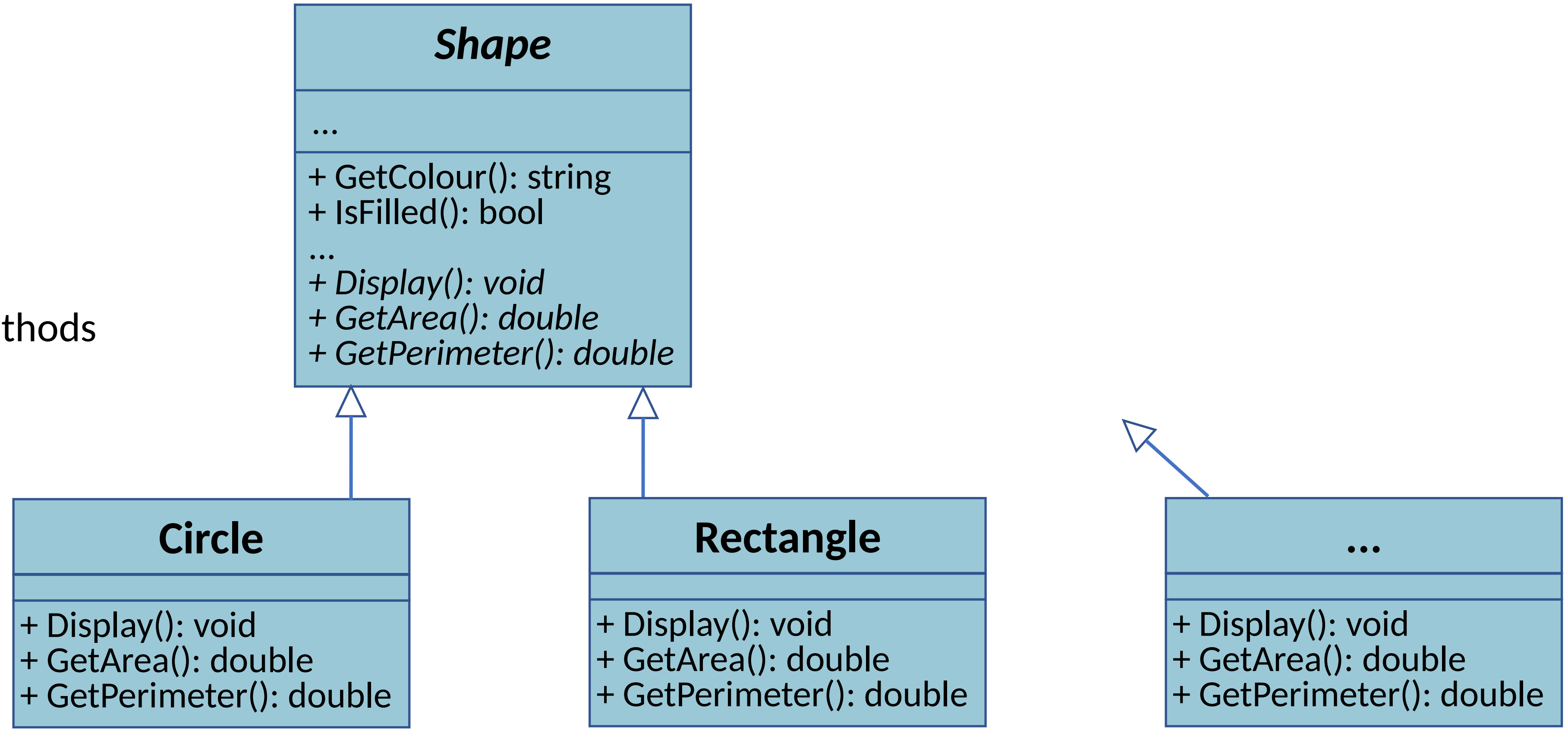
+ Display(): void
+ GetArea(): double
+ GetPerimeter(): double

...

+ Display(): void
+ GetArea(): double
+ GetPerimeter(): double

Classes attributes not represented in the diagram

Abstract classes: contract



Shape

...

+ GetColour(): string
+ IsFilled(): bool
...
+ Display(): void
+ GetArea(): double
+ GetPerimeter(): double

Circle

+ Display(): void
+ GetArea(): double
+ GetPerimeter(): double

Rectangle

+ Display(): void
+ GetArea(): double
+ GetPerimeter(): double

...

+ Display(): void
+ GetArea(): double
+ GetPerimeter(): double

Abstract methods

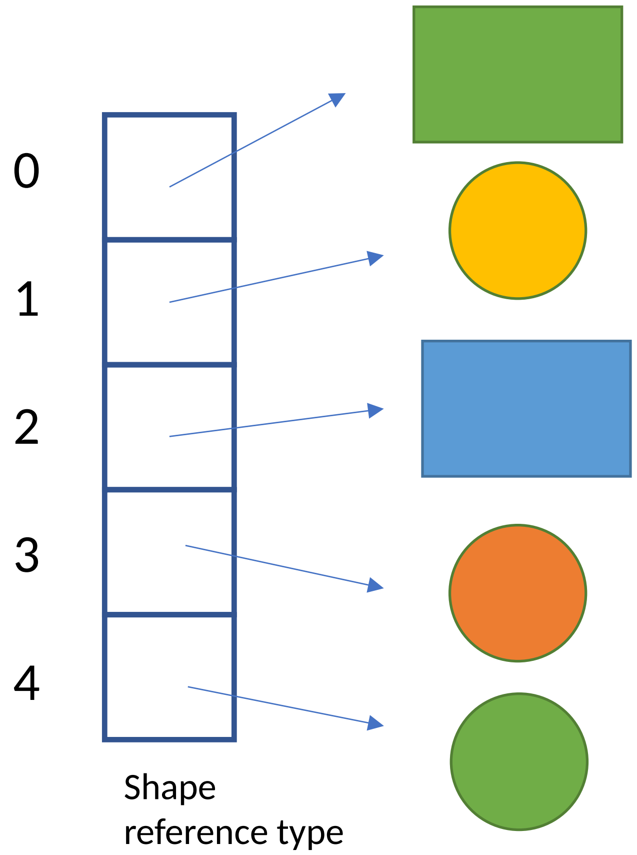
Overridden methods

Classes attributes not represented in the diagram

Polymorphism in action

```
public class ShapeTest
{
    public static void Main()
    {
        Shape[] shapes = new Shape[5];

        /* different shapes are created, e.g.,
           shapes[0] = new Rectangle( … );
           shapes[1] = new Circle ( … );
           […]
         */

        foreach (Shape s in shapes)
            s.Display();
    }

}
```

… the actual version of Display() called at run-time depends on the kind of shape, i.e., *Circle*, *Rectangle*, etc.—**late binding**



Abstract classes: contract

```
public abstract class Shape
{
    private string name;
    private bool filled;
    private string colour;

    public Shape(string c, bool f) { … }

    public void SetColour(string c) { … }
    public string GetColour() { … }
    protected void SetName(string n) { … }
    …

    public abstract void Display();
    public abstract double GetArea();
    public abstract double GetPerimeter();
}
```

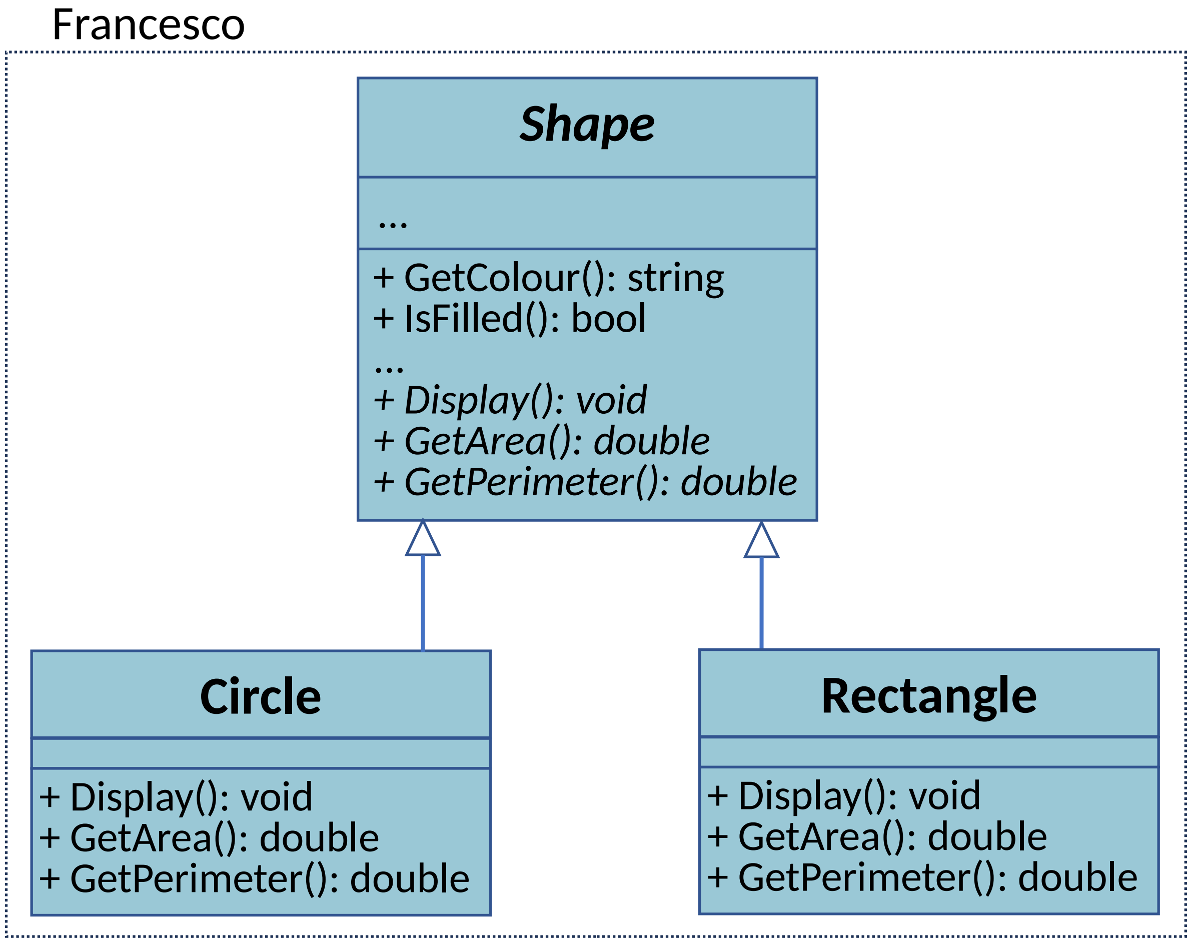
These are abstract methods that the subclasses **must** implement: a **contract**

A **polymorphic** system is implemented via the **contract**

Abstract classes: contract

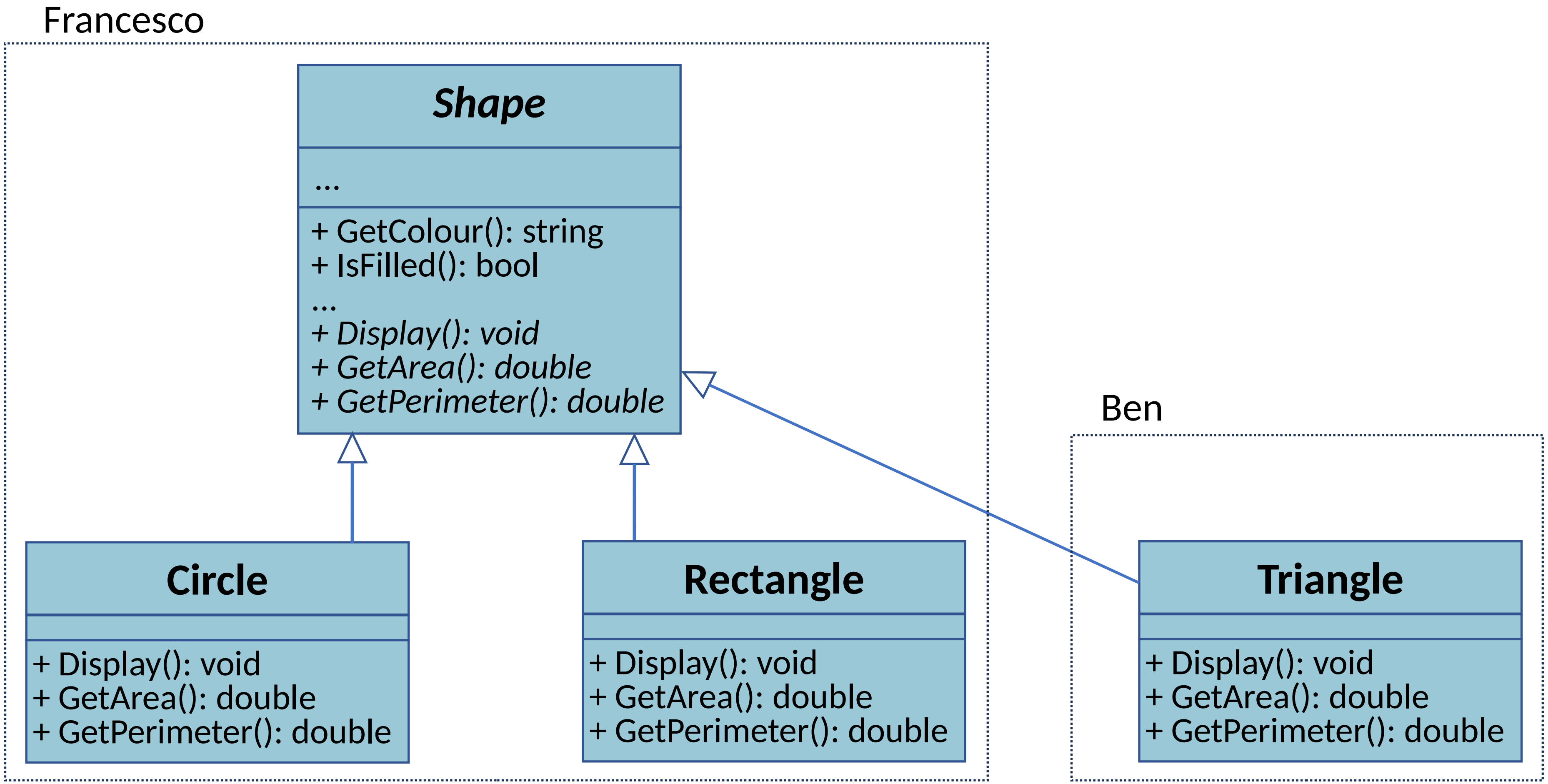
- Group development task based on the *Shape* *contract*

Abstract classes: contract



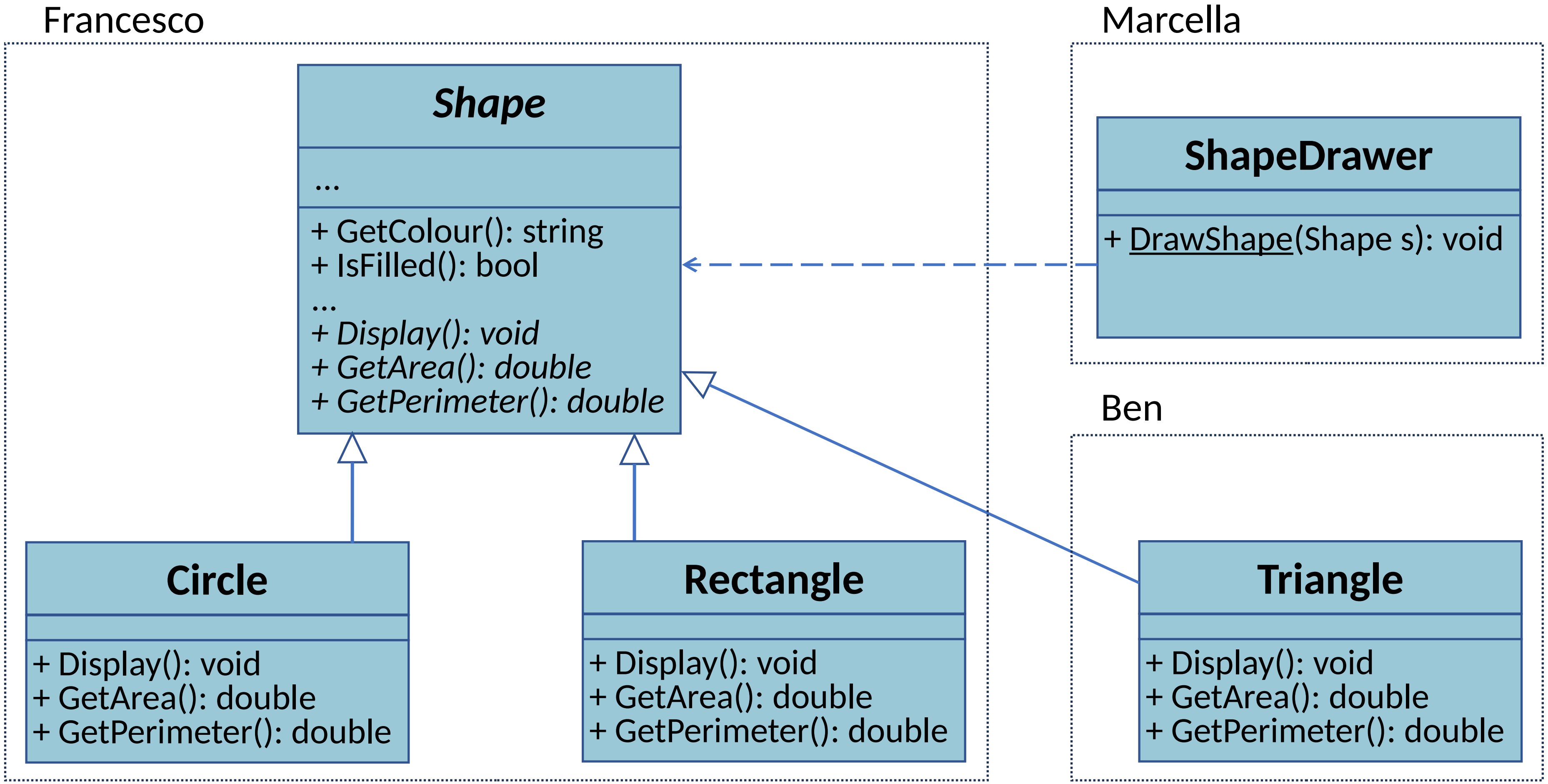
Francesco developed the *shape system* with *Shape*, *Circle* and *Rectangle*

Abstract classes: contract



Ben is given the *Shape* abstract class—the design contract to create a *Triangle* class

Abstract classes: contract



Marcella is given the *Shape* abstract class—the design contract to create a *ShapeDrawer* class that prints any shape

Abstract classes: contract

```
class Triangle
{
    // attributes
    …

    public Triangle( … ) { … }

    public override void Display() {
     // specific triangle implementation
    }

    public override double GetArea() {
     // specific triangle implementation
    }

    public override double GetPerimeter() {
    // specific triangle implementation
    }

}
```

Ben only needs to know the **contract** specification of *Shape*, **not** how other shapes are **implemented**—**abstraction**

Abstract classes: contract

```
static class ShapeDrawer
{
    public static void DrawShape(Shape s)
    {
        s.Display();
        s.GetArea();
        s.GetPerimeter();
    }
     ...
}
```

When implementing *DrawShape*, Marcella only needs to know the **contract** specification of *Shape* and invoke *any method* of that *contract*.

No need to know how those shapes are implemented—**abstraction**

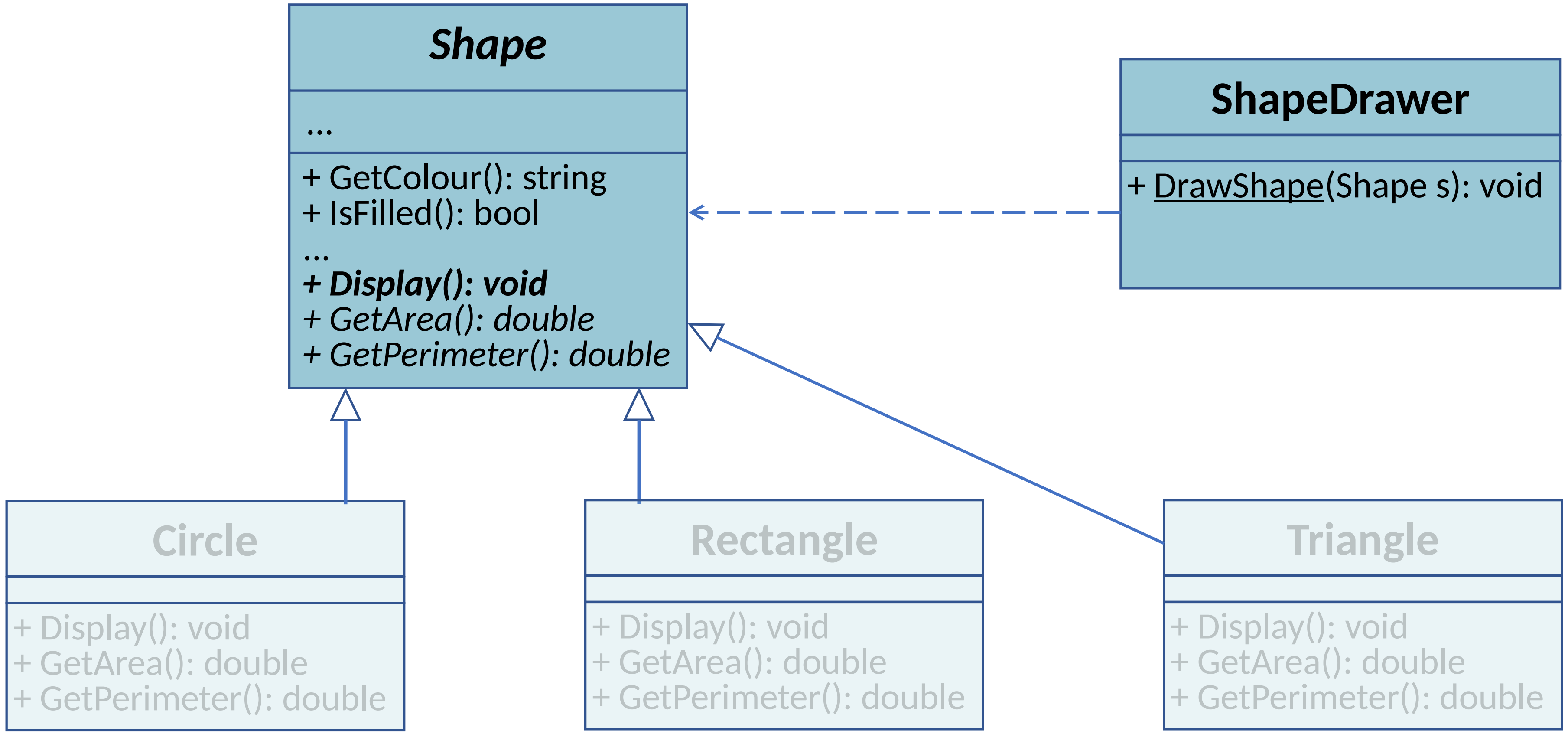
Object-Oriented Programming (OOP) Principles

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

A class should provide an **abstract view** of a “service” through its *public methods* and **hide** the **implementation** details.

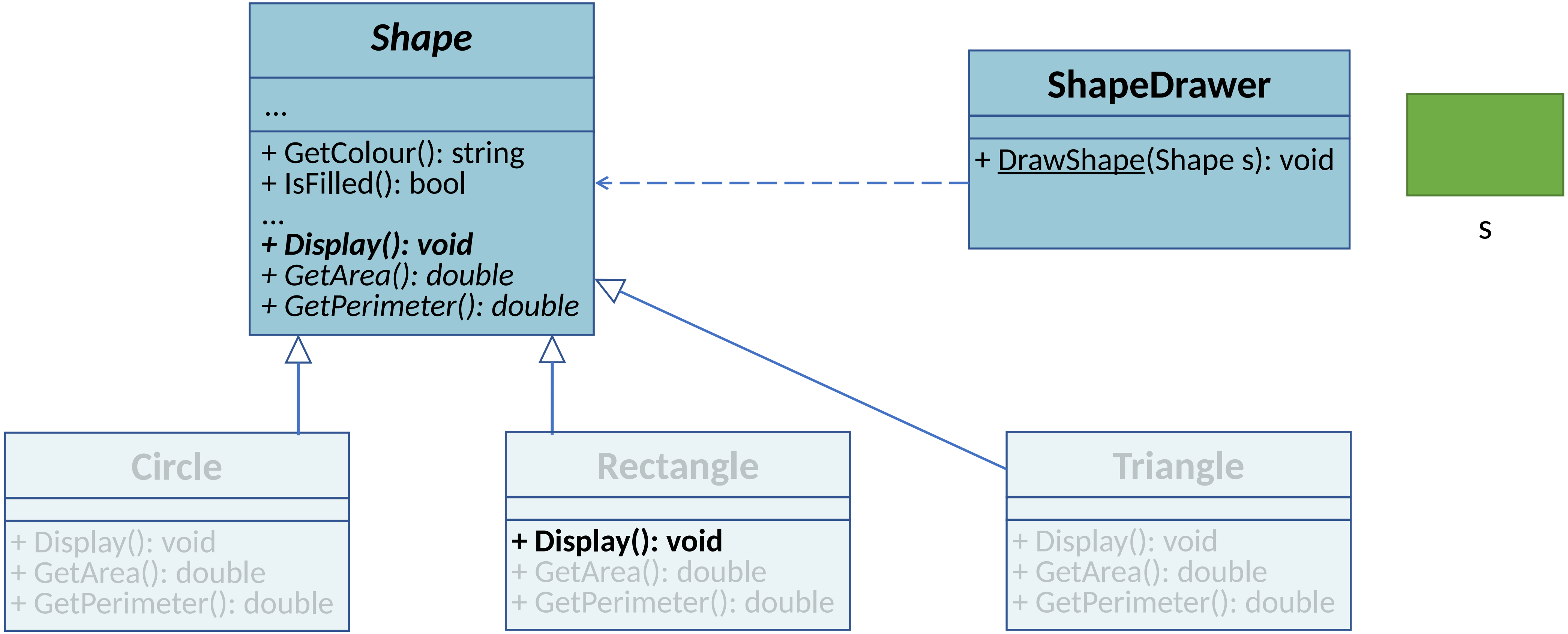
Abstraction is related to *polymorphism*, *encapsulation* and *implementation hiding*.

Abstract classes: contract



DrawShape uses the reference passed via *s* to call *Display*(), part of the *Shape* contract

Abstract classes: contract



Shape

…

+ GetColour(): string
+ IsFilled(): bool
…
+ Display(): void
+ GetArea(): double
+ GetPerimeter(): double

ShapeDrawer

+ DrawShape(Shape s): void

s

Circle

+ Display(): void
+ GetArea(): double
+ GetPerimeter(): double

Rectangle

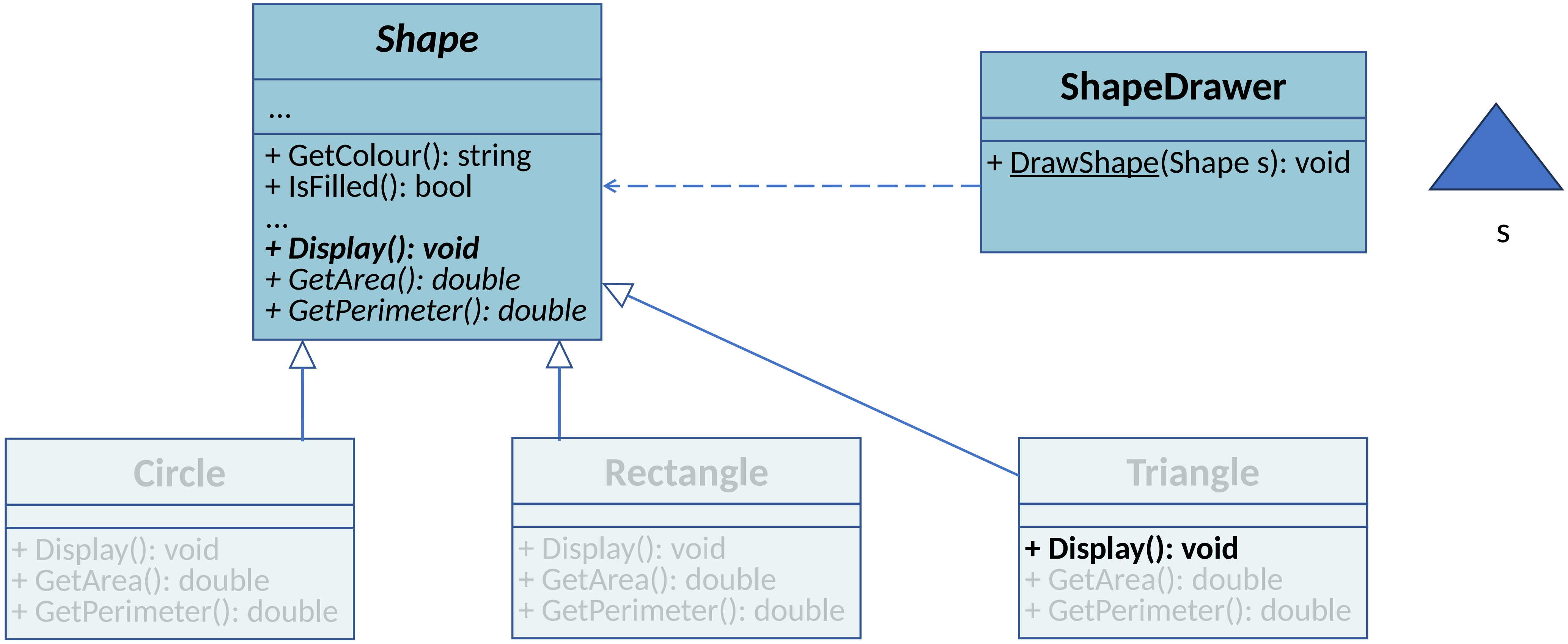
+ Display(): void
+ GetArea(): double
+ GetPerimeter(): double

Triangle

+ Display(): void
+ GetArea(): double
+ GetPerimeter(): double

Polymorphic behaviour: the code specific to a shape is executed at runtime—*Rectangle*

Abstract classes: contract



Shape

...

+ GetColour(): string
+ IsFilled(): bool
...
+ Display(): void
+ GetArea(): double
+ GetPerimeter(): double

ShapeDrawer

+ DrawShape(Shape s): void

s

Circle

+ Display(): void
+ GetArea(): double
+ GetPerimeter(): double

Rectangle

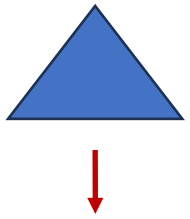
+ Display(): void
+ GetArea(): double
+ GetPerimeter(): double

Triangle

+ Display(): void
+ GetArea(): double
+ GetPerimeter(): double

Polymorphic behaviour: the code specific to a shape is executed at runtime—*Triangle*

Abstract classes: contract



```
class Shape
{
    ...
    public abstract void Display();
    public abstract double GetArea();
    public abstract double GetPerimeter();
    ...
}
```

```
static class ShapeDrawer
{
    public static void DrawShape(Shape s)
    {
        s.Display();
        // s.GetArea() or s.GetPerimeter()
    }
     ...
}
```

```
class Triangle
{
    ...
    public override void Display() {
     // specific triangle implementation
    }

    public override double GetArea() { ... }

    public override double GetPerimeter()
{ ... }
}
```

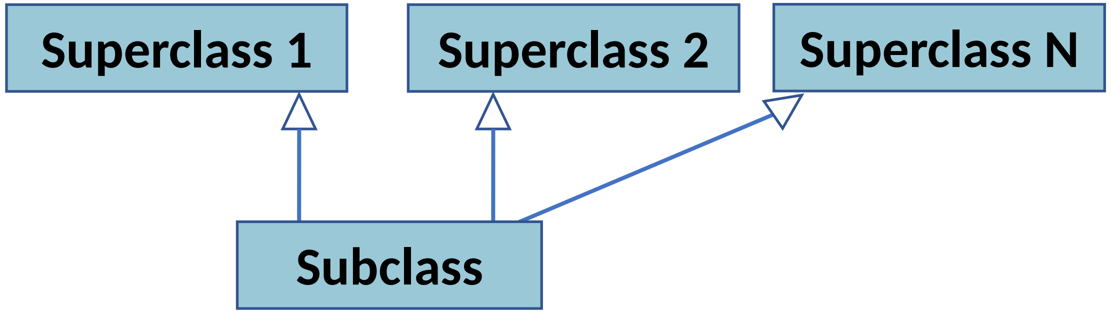
```
class Rectangle
{
    ...
    public override void Display() {
     // specific rectangle implementation
    }

    public override double GetArea() { ... }

    public override double GetPerimeter()
{ ... }
}
```

Outline

- Design Contracts
  - Summary of abstract classes
  - Interfaces
- C# inheritance tree: Object class

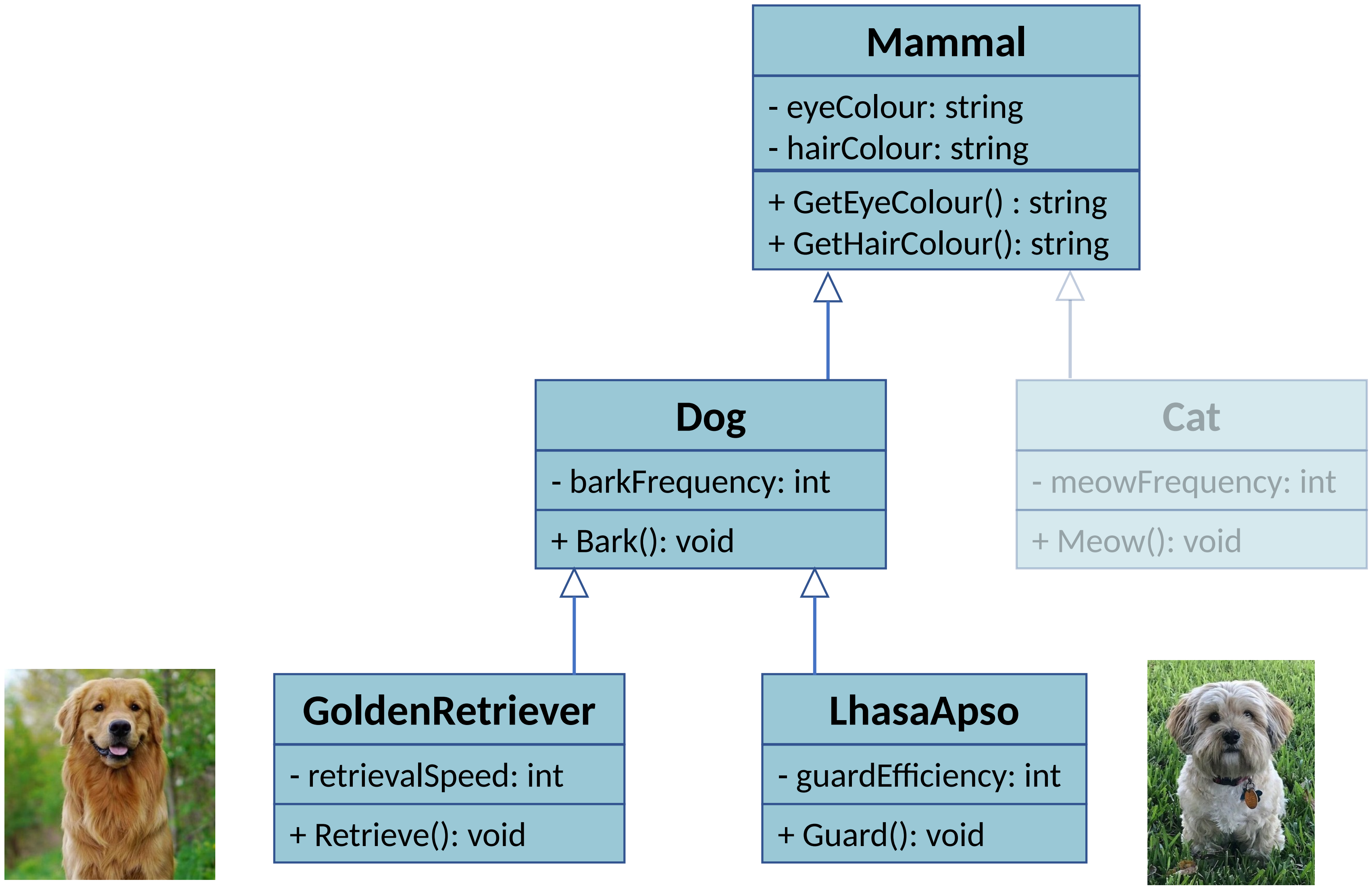
Multiple inheritance

- An abstract class allows for defining **design contracts**
- **Idea**: a subclass can inherit from **multiple** abstract superclasses, each describing a *part* of a *larger* contract



- Let's see how *multiple inheritance* would work in general with an example

Inheritance: Mammal Hierarchy



Mammal

- eyeColour: string
- hairColour: string

+ GetEyeColour() : string
+ GetHairColour(): string

Dog

- barkFrequency: int

+ Bark(): void

Cat

- meowFrequency: int

+ Meow(): void

GoldenRetriever

- retrievalSpeed: int

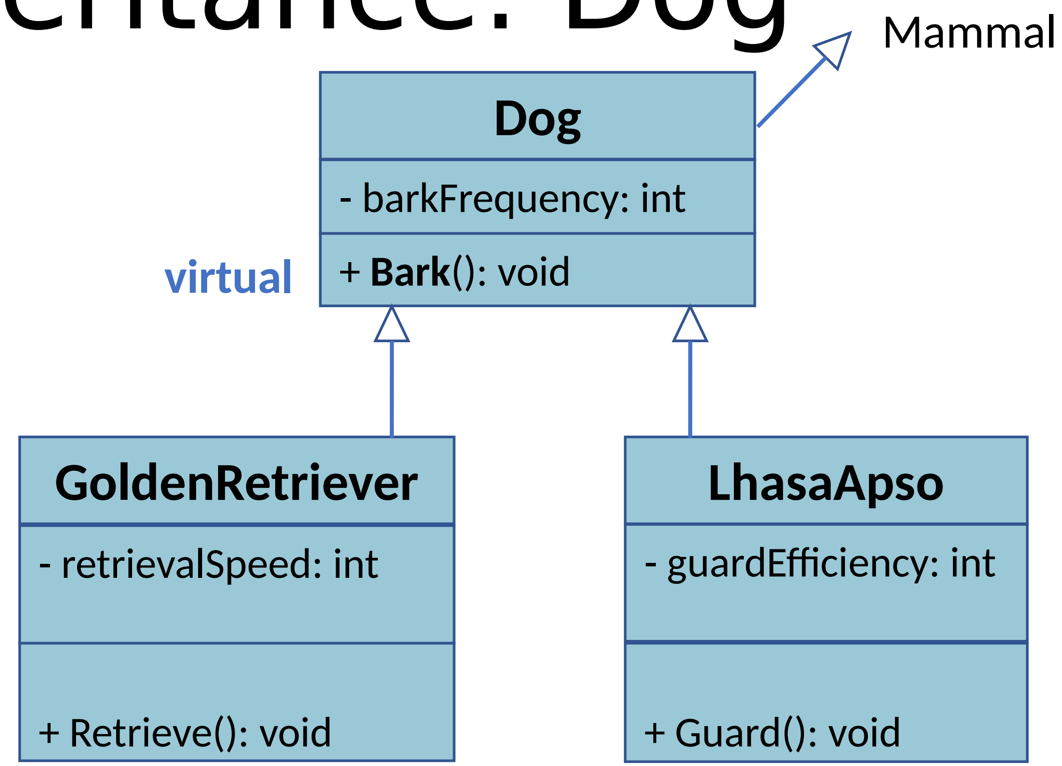
+ Retrieve(): void

LhasaApso

- guardEfficiency: int

+ Guard(): void

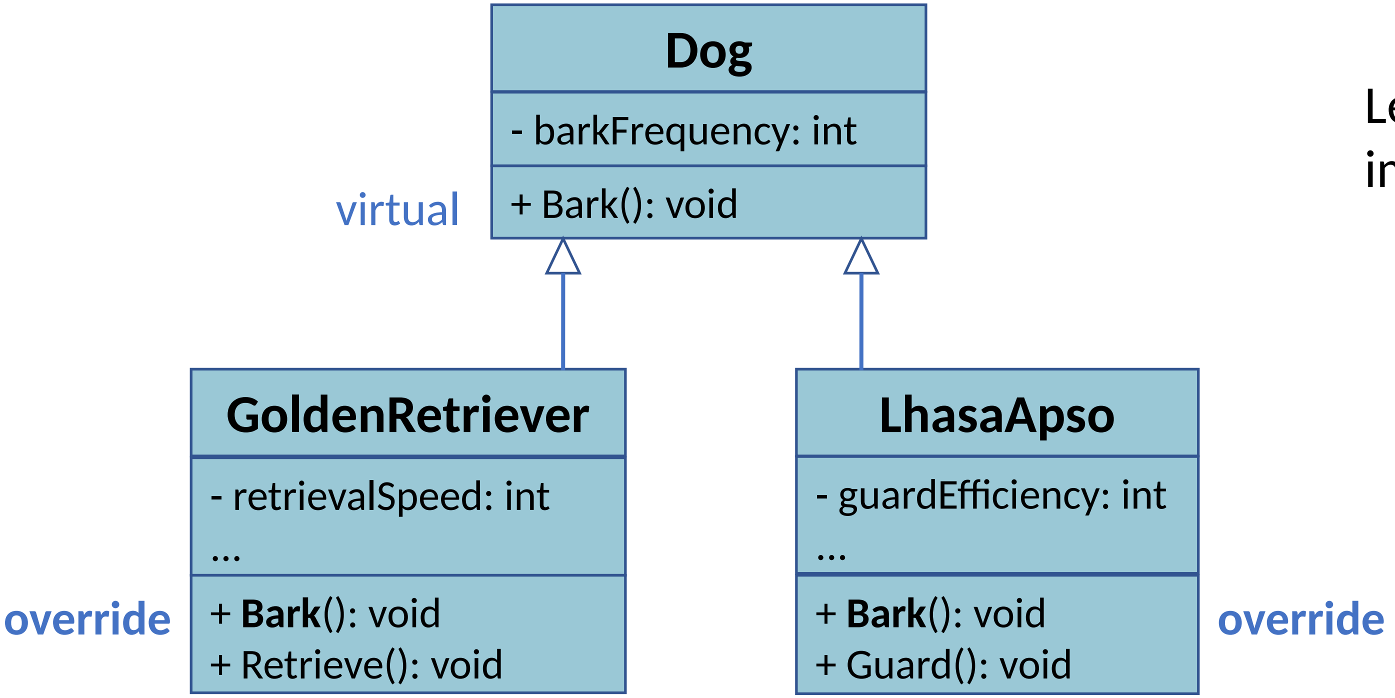
Inheritance: Dog



Let's use *polymorphism* to implement different versions of *Bark*

After testing, we realised *Bark* should not be the same in the subclasses

Inheritance: Dog



Dog

- barkFrequency: int

+ Bark(): void

virtual

GoldenRetriever

- retrievalSpeed: int

...

+ **Bark**(): void

+ Retrieve(): void

override

LhasaApso

- guardEfficiency: int

...

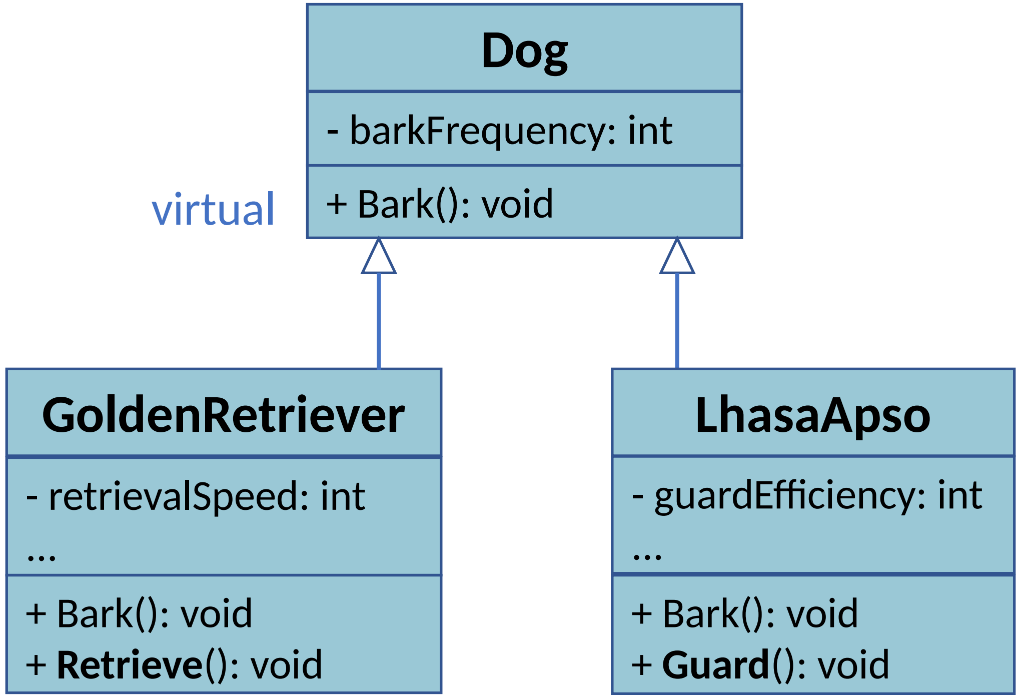
+ **Bark**(): void

+ Guard(): void

override

Let's use *polymorphism* to implement different versions of *Bark*

Inheritance: more specialised dog



virtual

Dog

- barkFrequency: int

+ Bark(): void

GoldenRetriever

- retrievalSpeed: int

...

+ Bark(): void

+ **Retrieve**(): void

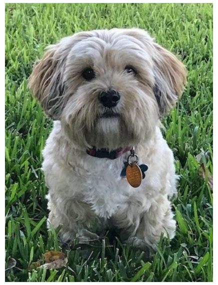
LhasaApso

- guardEfficiency: int

...

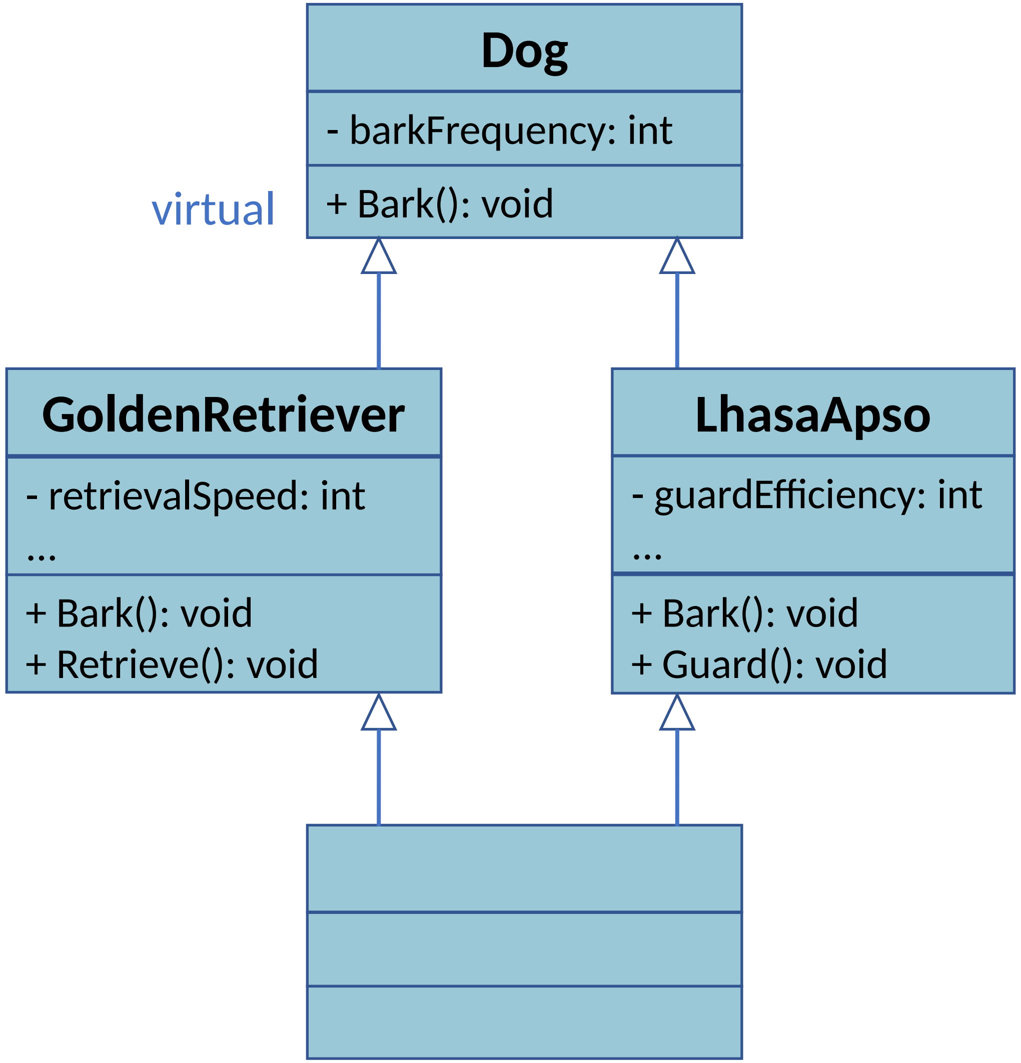
+ Bark(): void

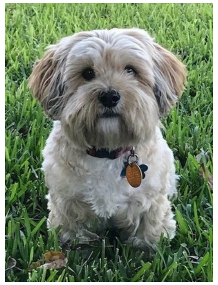
+ **Guard**(): void



We use this *hierarchy* to create a dog that has both *Retrieve* and *Guard*

Inheritance: more specialised dog





A subclass that inherits from **two** parent classes

Inheritance: more specialised dog

Dog

- barkFrequency: int

virtual + Bark(): void

GoldenRetriever

- retrievalSpeed: int

…

+ Bark(): void

+ Retrieve(): void

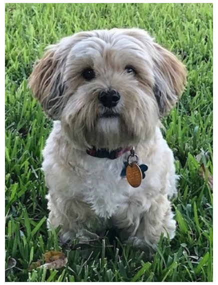
LhasaApso

- guardEfficiency: int

…

+ Bark(): void

+ Guard(): void



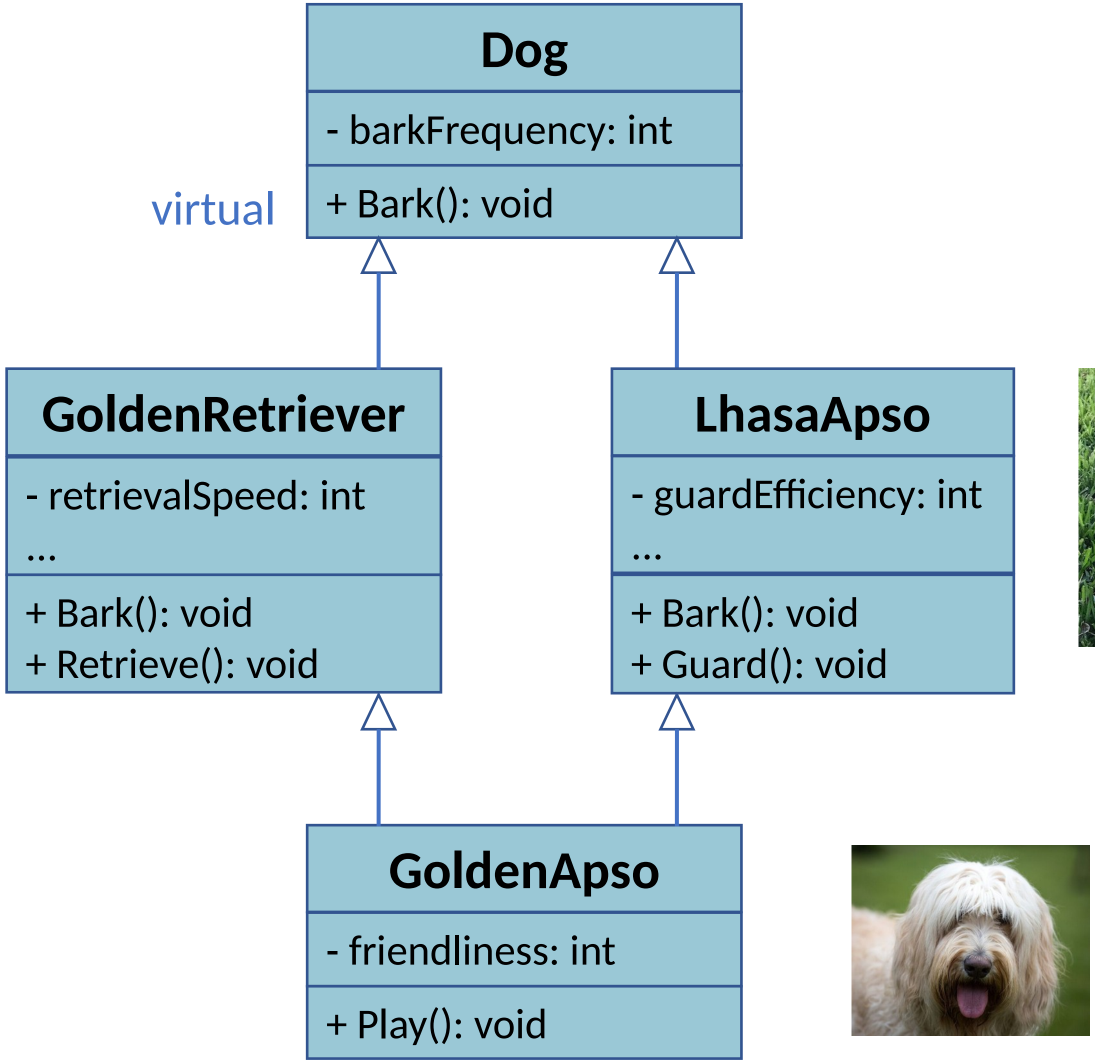
GoldenApso

- friendliness: int

+ Play(): void

GoldenApso **inherits** *Retrieve* from *GoldenRetriever*, *Guard* from *LhasaApso*, and **defines** *Play*

Inheritance: more specialised dog



virtual

Dog

- barkFrequency: int

+ Bark(): void

GoldenRetriever

- retrievalSpeed: int
...

+ Bark(): void
+ Retrieve(): void

LhasaApso

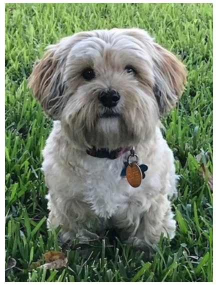
- guardEfficiency: int
...

+ Bark(): void
+ Guard(): void

GoldenApso

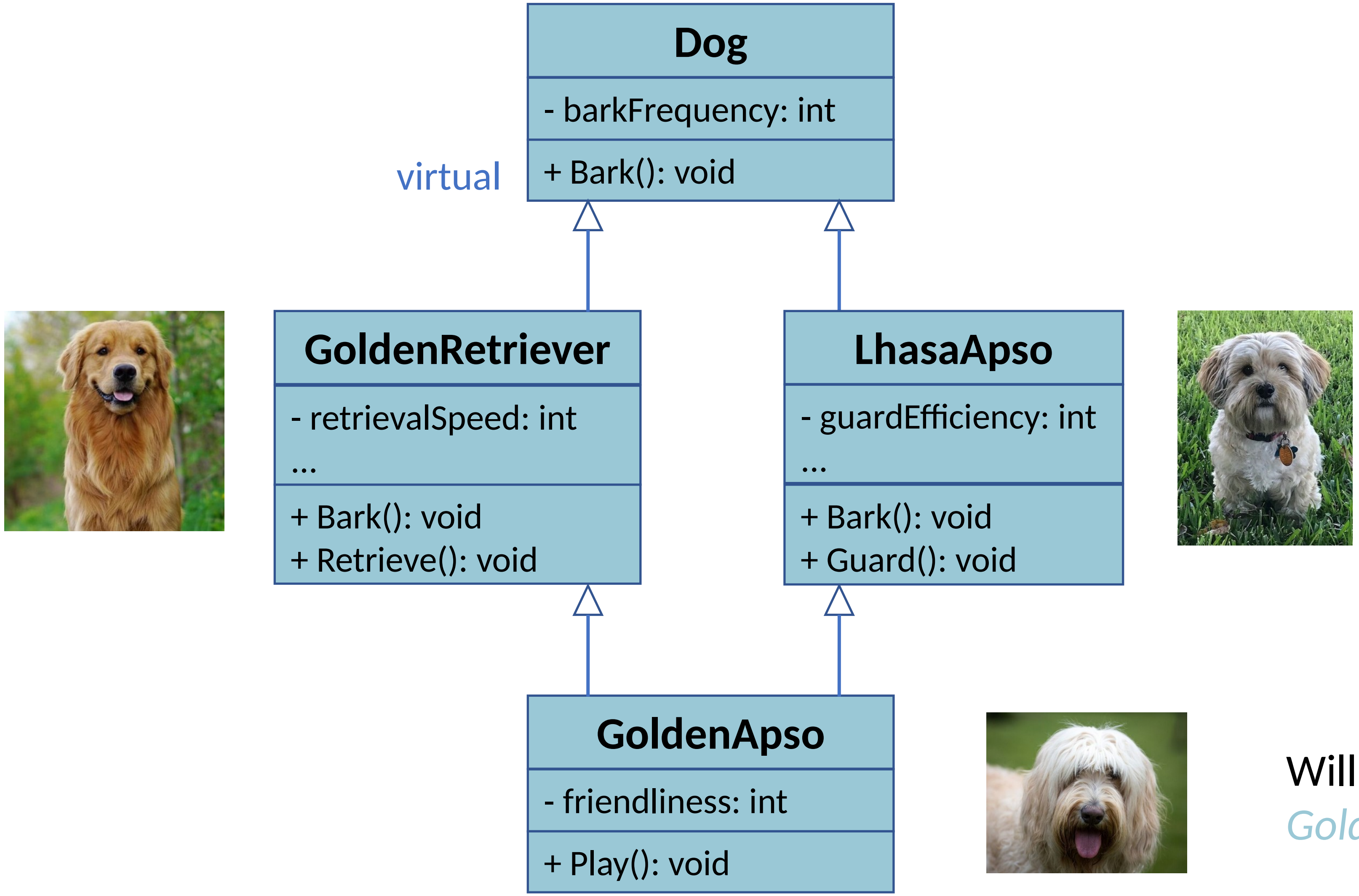
- friendliness: int

+ Play(): void



How does a *GoldenApso* *bark*?

Inheritance: more specialised dog



Dog

- barkFrequency: int

virtual

+ Bark(): void

GoldenRetriever

- retrievalSpeed: int

…

+ Bark(): void

+ Retrieve(): void

LhasaApso

- guardEfficiency: int

…

+ Bark(): void

+ Guard(): void

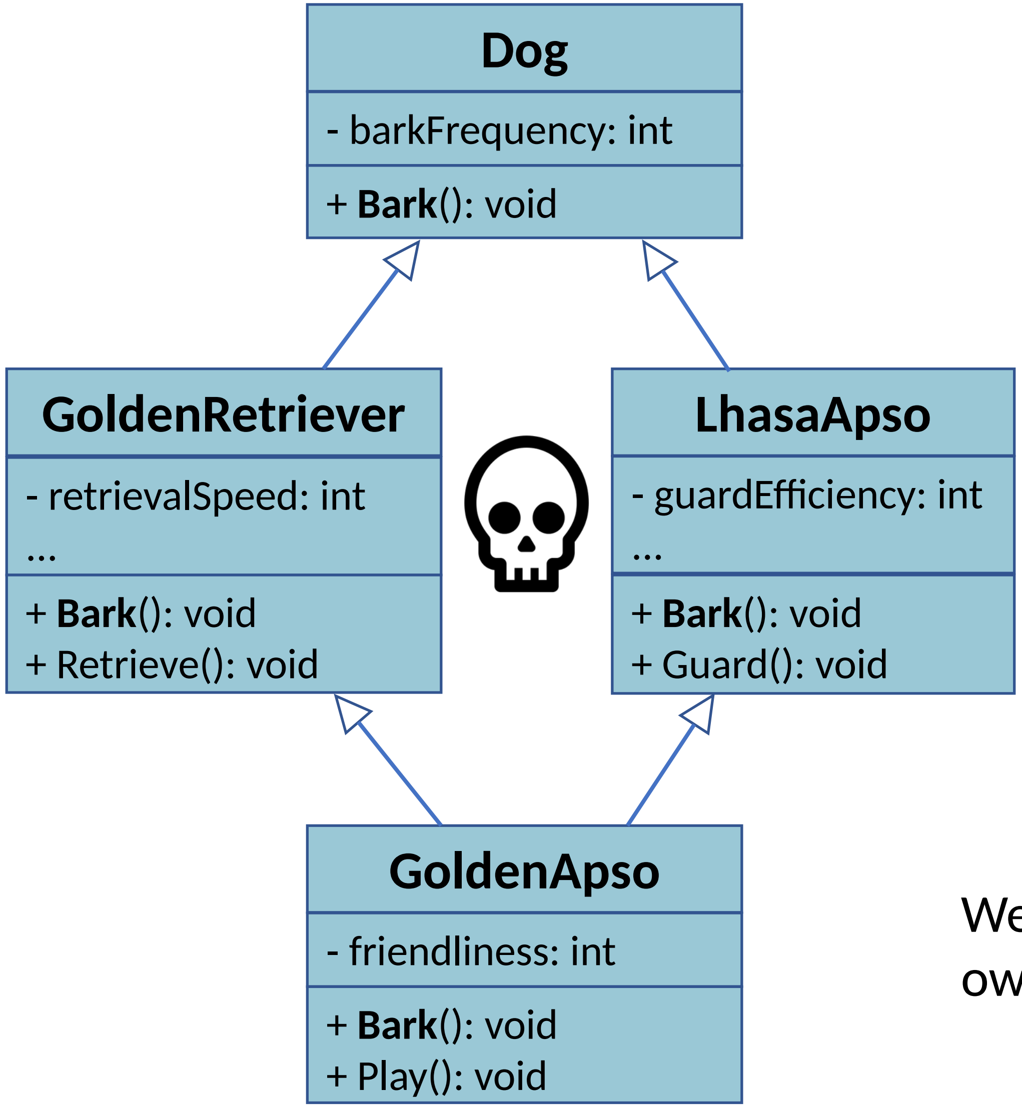
GoldenApso

- friendliness: int

+ Play(): void

Will *Bark* be inherited from *GoldenRetriever* or *LhasaApso*?

Inheritance: more specialised dog



Dog

- barkFrequency: int

+ **Bark**(): void

GoldenRetriever

- retrievalSpeed: int

…

+ **Bark**(): void

+ Retrieve(): void

LhasaApso

- guardEfficiency: int

…

+ **Bark**(): void

+ Guard(): void

💀

GoldenApso

- friendliness: int

+ **Bark**(): void

+ Play(): void

“Deadly Diamond of Death”

We do not know! *GoldenApso* should provide its own *Bark* overridden method

Multiple inheritance

- An abstract class allows for defining **design contracts**
- ~~**Idea**: a subclass can inherit from **multiple** abstract superclasses, each describing a *part* of a *larger* contract~~
- **Reality**: this is **not supported** by *C#* or *Java*
- C++ supports it but can lead to the “Deadly Diamond of Death”

How can we solve the problem?

Interfaces

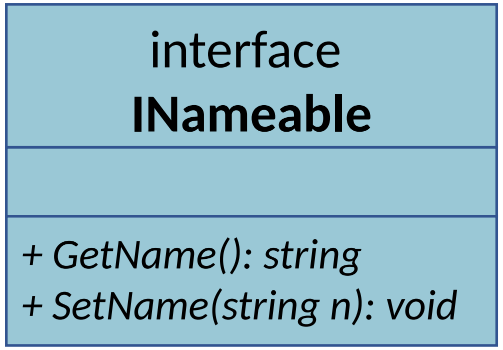
- So far, we have used the term **interface** to describe the set of public methods that an object exposes
- In many OOP languages, such as *C#* and *Java*, **interface** also indicates an alternative to abstract classes for defining **design contracts**

Interfaces

- An abstract class can have attributes, methods and abstract methods

- An interface **only includes methods without implementation** (body) that are *implicitly* abstract and public (< C# 8)
- *Inheriting* from interfaces allows for defining **design contracts** while preventing the *Deadly Diamond of Death*

Interfaces



interface
INameable

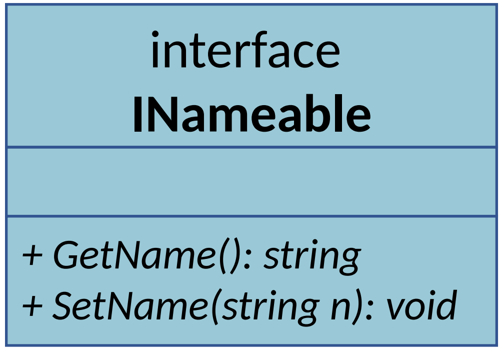
+ GetName(): string
+ SetName(string n): void

C# convention, capital *I*

```
interface INameable
{
    // no attributes (only constants)

   // no constructors

    public void SetName(string name);
    public string GetName();
}
```

INameable defines a behaviour associated with *nameable* entities

Interfaces



interface
INameable

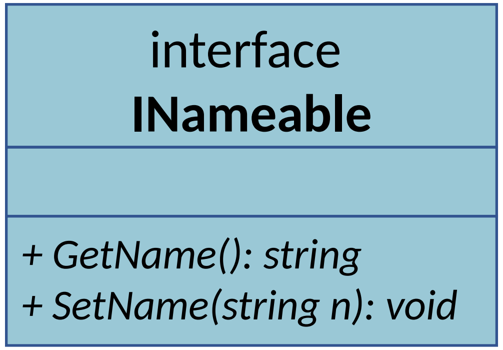
+ GetName(): string
+ SetName(string n): void

```
interface INameable
{
    // no attributes (only constants)

   // no constructors

    public void SetName(string name);
    public string GetName();
}
```

No state / attributes cannot be instantiated

INameable defines a behaviour associated with *nameable* entities

Interfaces



interface
INameable

+ GetName(): string
+ SetName(string n): void

```
interface INameable
{
   // no attributes (only constants)

  // no constructors

   public void SetName(string name);
   public string GetName();
}
```

methods are implicitly public
and have **no body** (abstract)

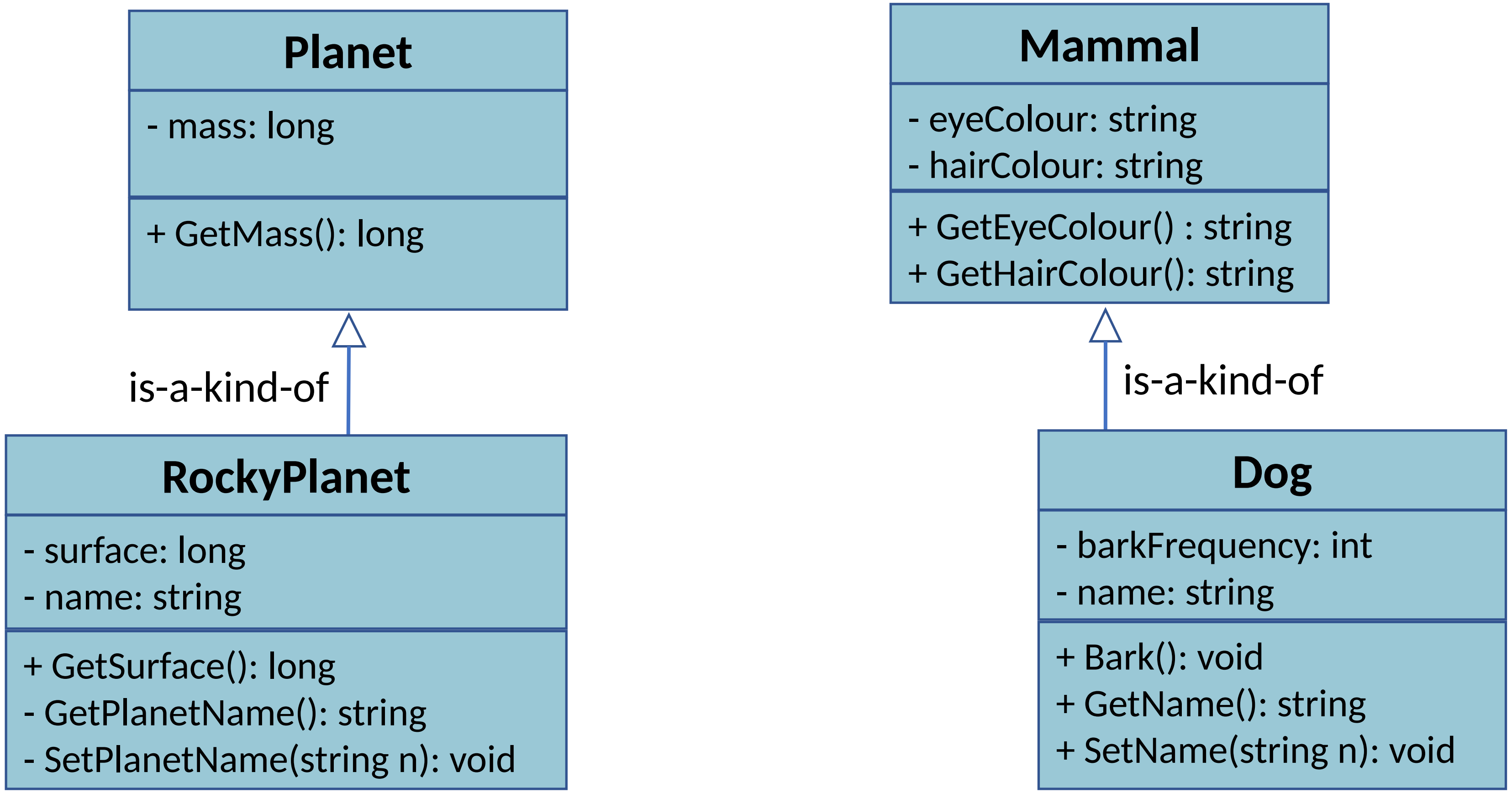
INameable defines a behaviour associated with *nameable* entities

Interfaces

- A class can **inherit** directly from **only one** *superclass*
- It can **implement many** *interfaces*

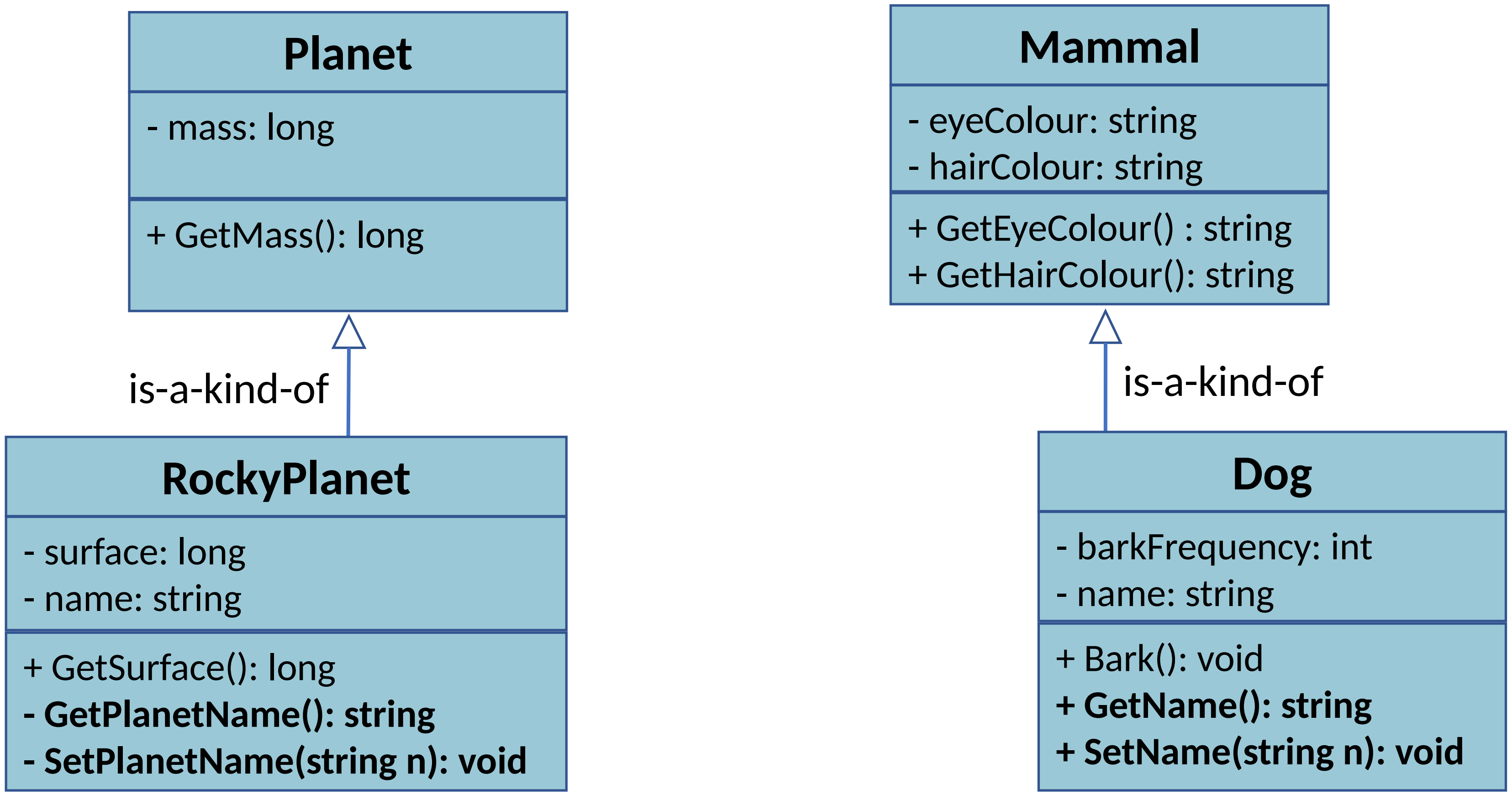
Example: a space exploration game

Interfaces



Planet
- mass: long
+ GetMass(): long
is-a-kind-of
RockyPlanet
- surface: long
- name: string
+ GetSurface(): long
- GetPlanetName(): string
- SetPlanetName(string n): void
Mammal
- eyeColour: string
- hairColour: string
+ GetEyeColour() : string
+ GetHairColour(): string
is-a-kind-of
Dog
- barkFrequency: int
- name: string
+ Bark(): void
+ GetName(): string
+ SetName(string n): void

Interfaces



Planet

- mass: long

+ GetMass(): long

is-a-kind-of

RockyPlanet

- surface: long
- name: string

+ GetSurface(): long

- GetPlanetName(): string

- SetPlanetName(string n): void

Mammal

- eyeColour: string
- hairColour: string

+ GetEyeColour() : string
+ GetHairColour(): string

is-a-kind-of

Dog

- barkFrequency: int
- name: string

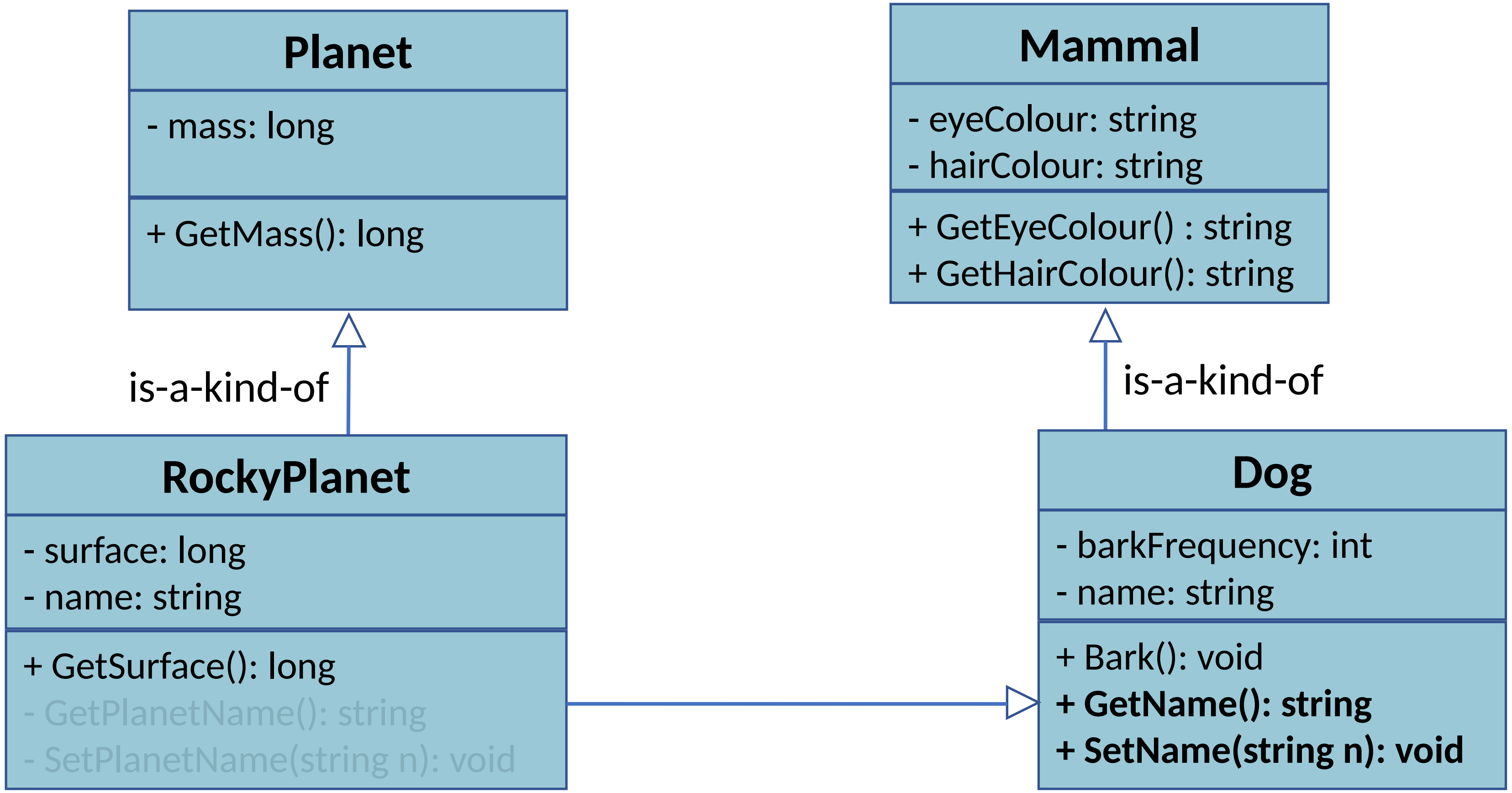
+ Bark(): void

+ GetName(): string

+ SetName(string n): void

A well-thought design should *standardise* those behaviours

Interfaces

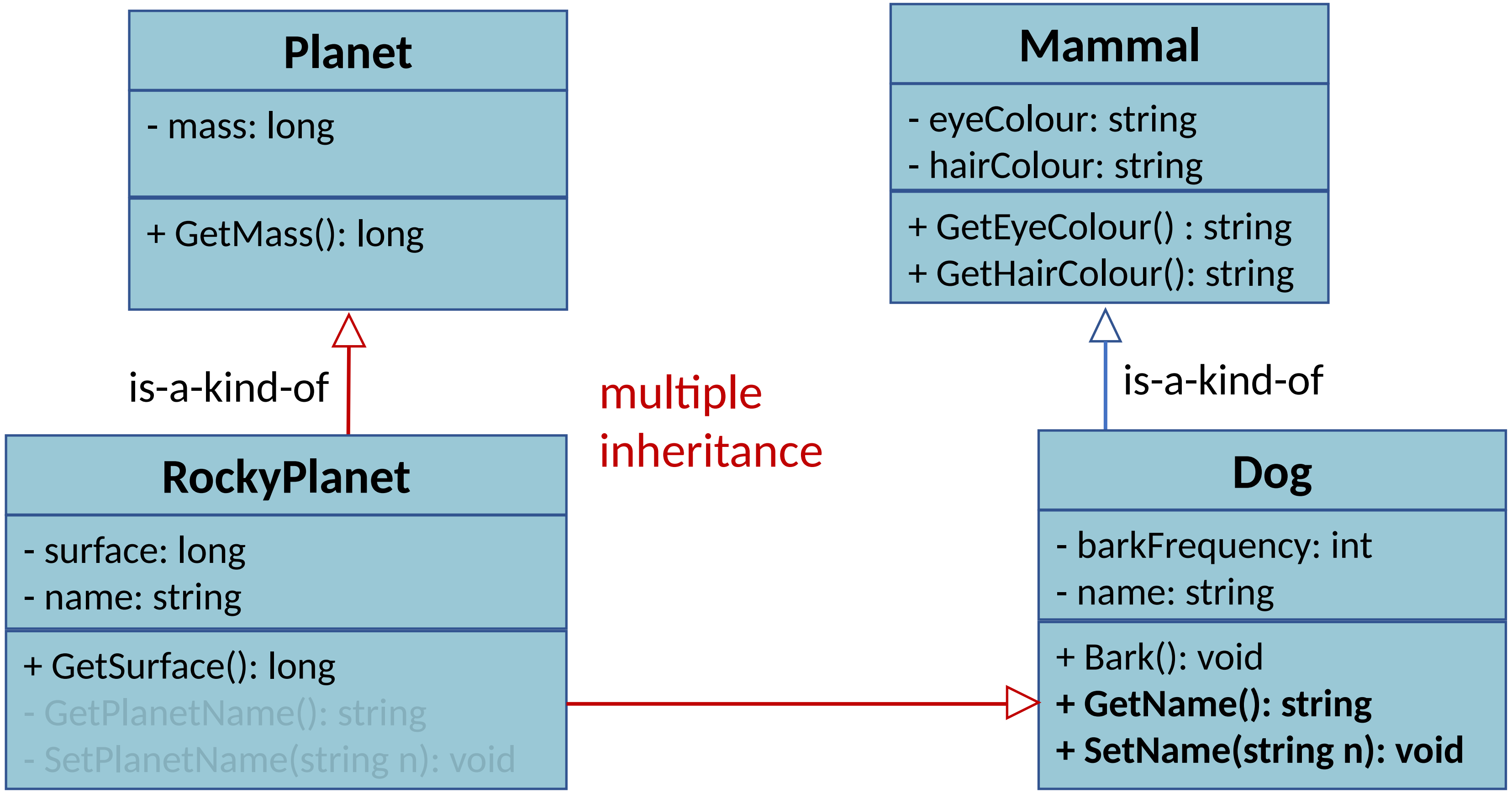


RockyPlanet could inherit them from *Dog*—would this make sense?

Inheritance: drawbacks

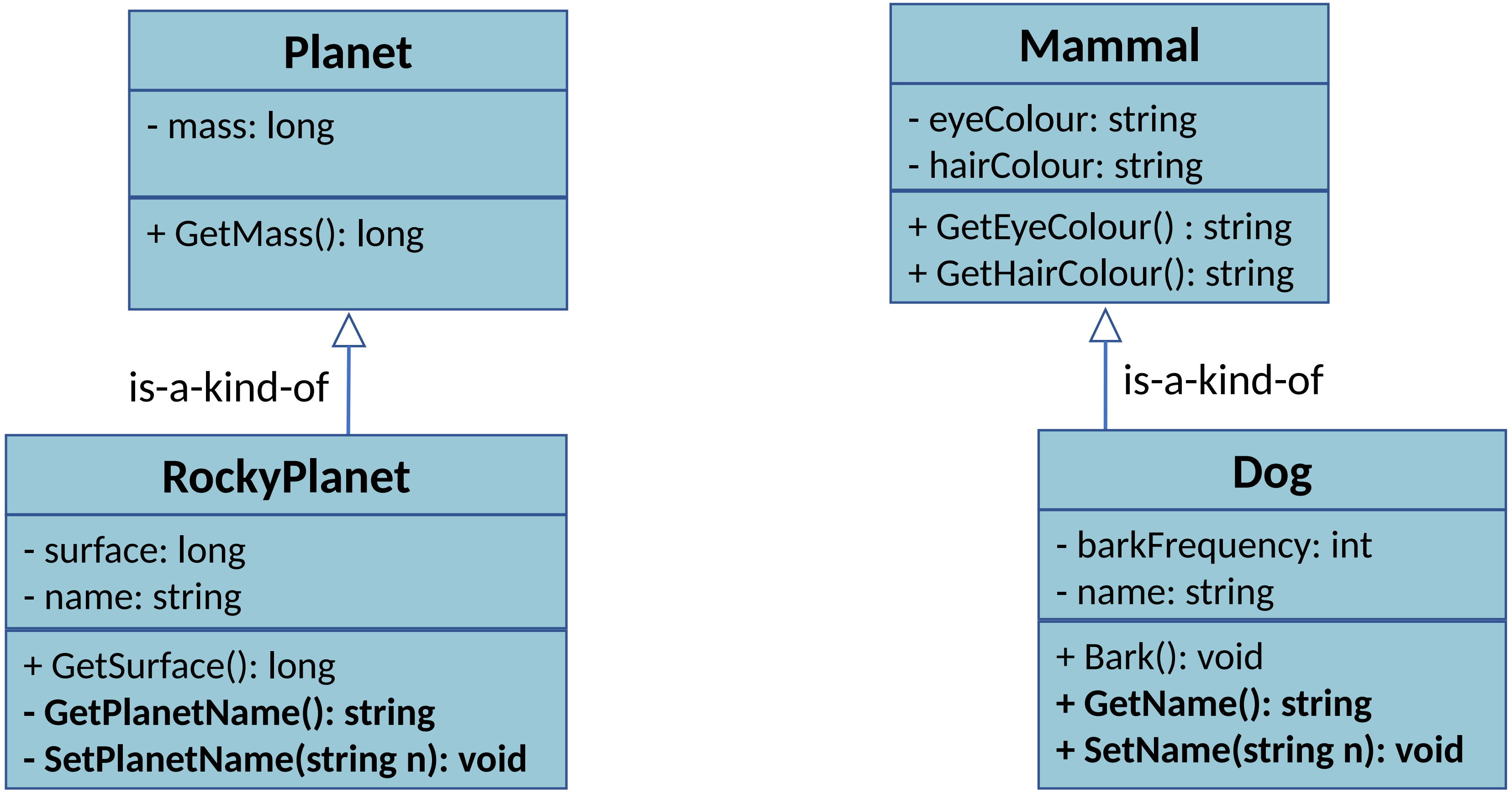
- **Rigid** and not **flexible** inheritance hierarchy: *RockyPlanet* is not a *Dog*
- Class inheritance **breaks** encapsulation: *issues* if *not a true is-a-kind-of* —**changes** to a superclass can have a **ripple effect** on subclasses

Interfaces



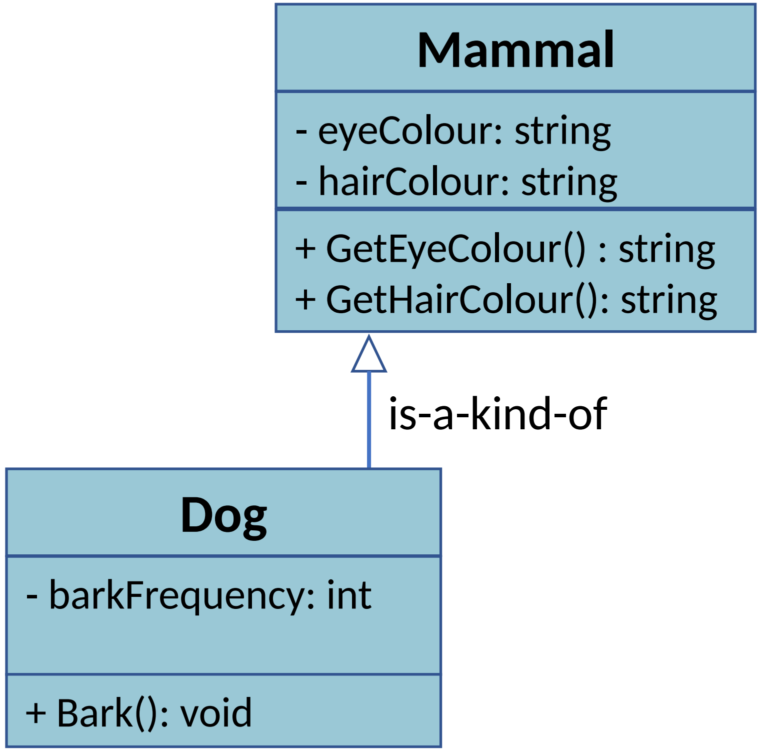
Also, *RockyPlanet* would **inherit** from *Dog* and Planet at the **same time**

Interfaces



Can **interfaces** be used for this design? Let's see how

Interfaces



Mammal

- eyeColour: string
- hairColour: string

+ GetEyeColour() : string
+ GetHairColour(): string

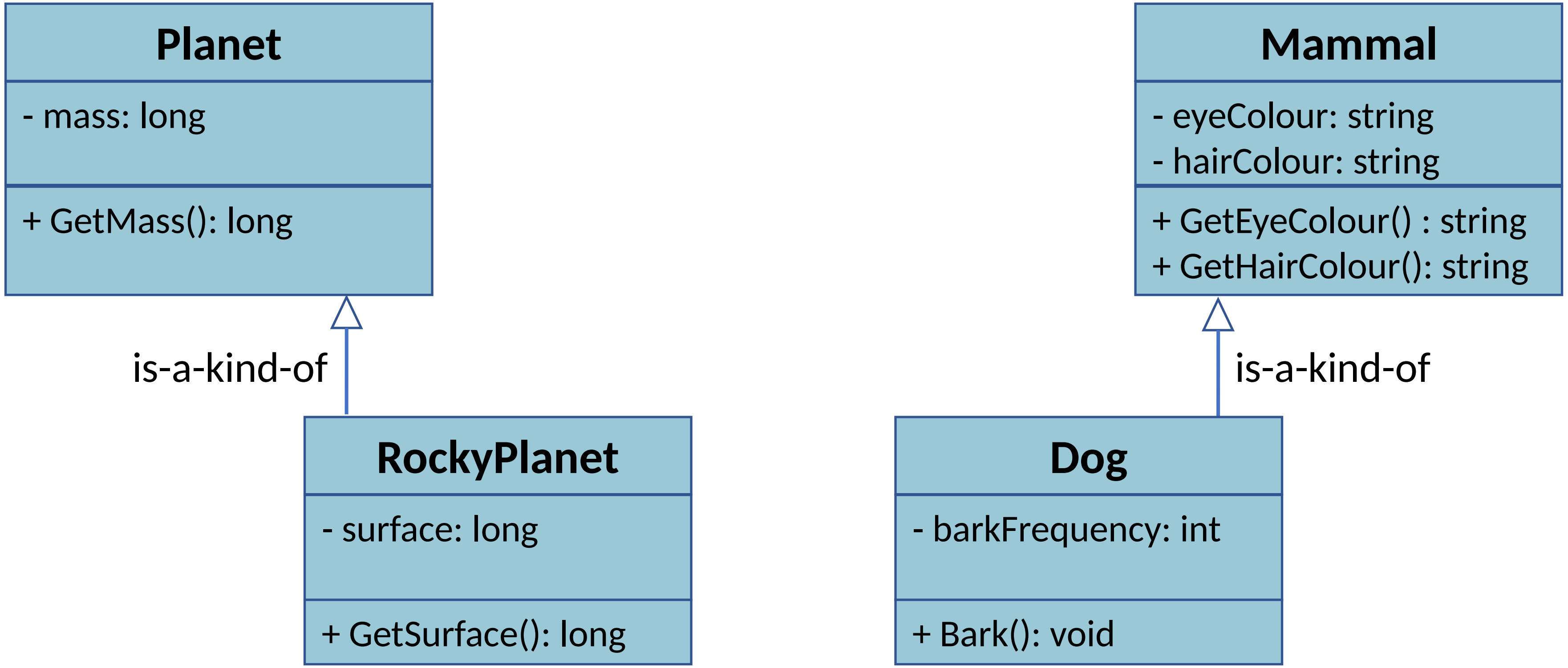
is-a-kind-of

Dog

- barkFrequency: int

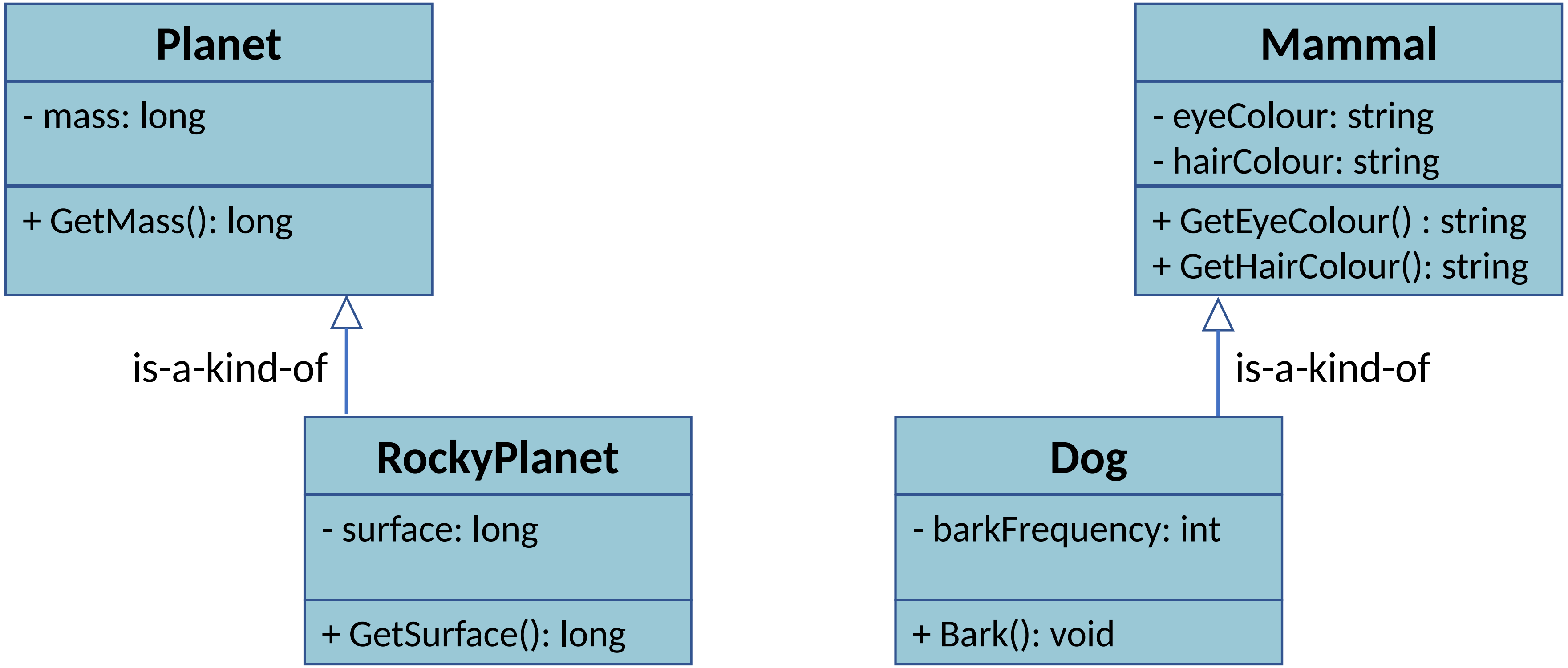
+ Bark(): void

Interfaces



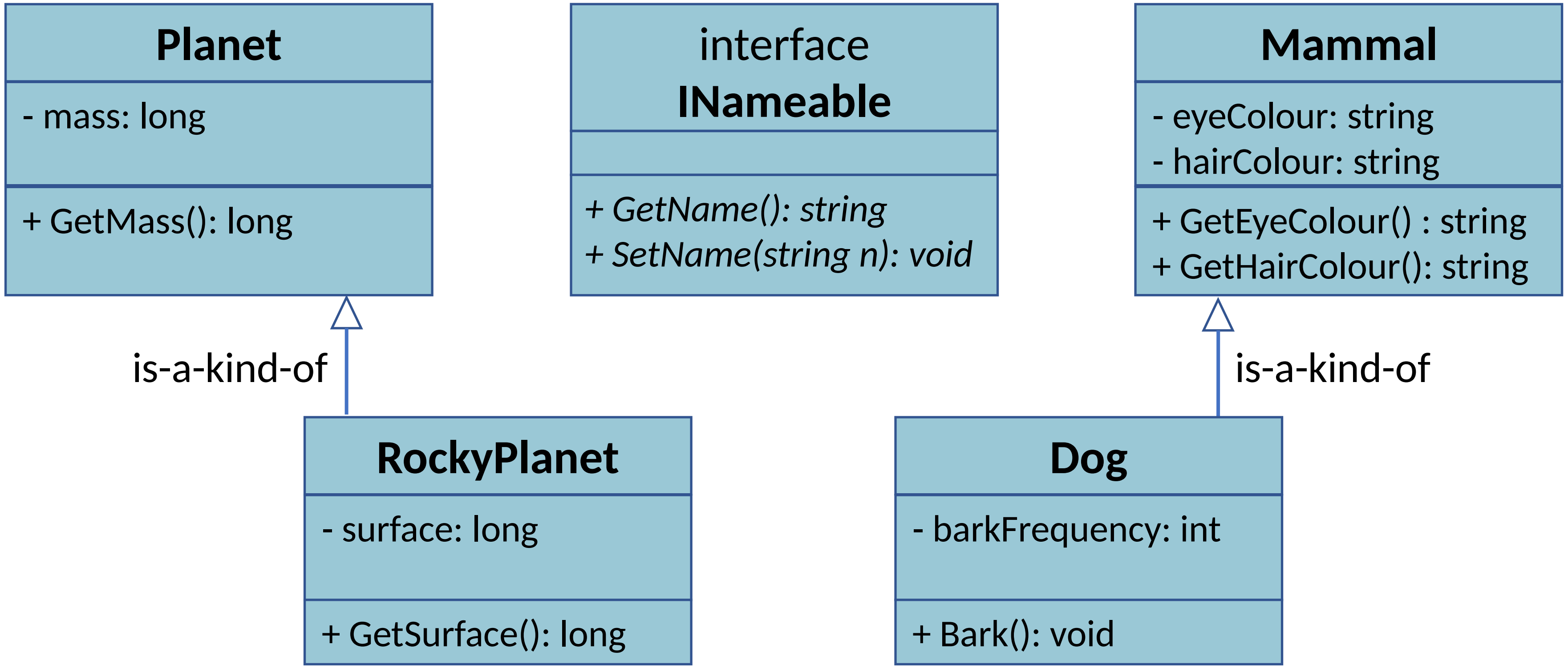
Completely **different** inheritance structure

Interfaces

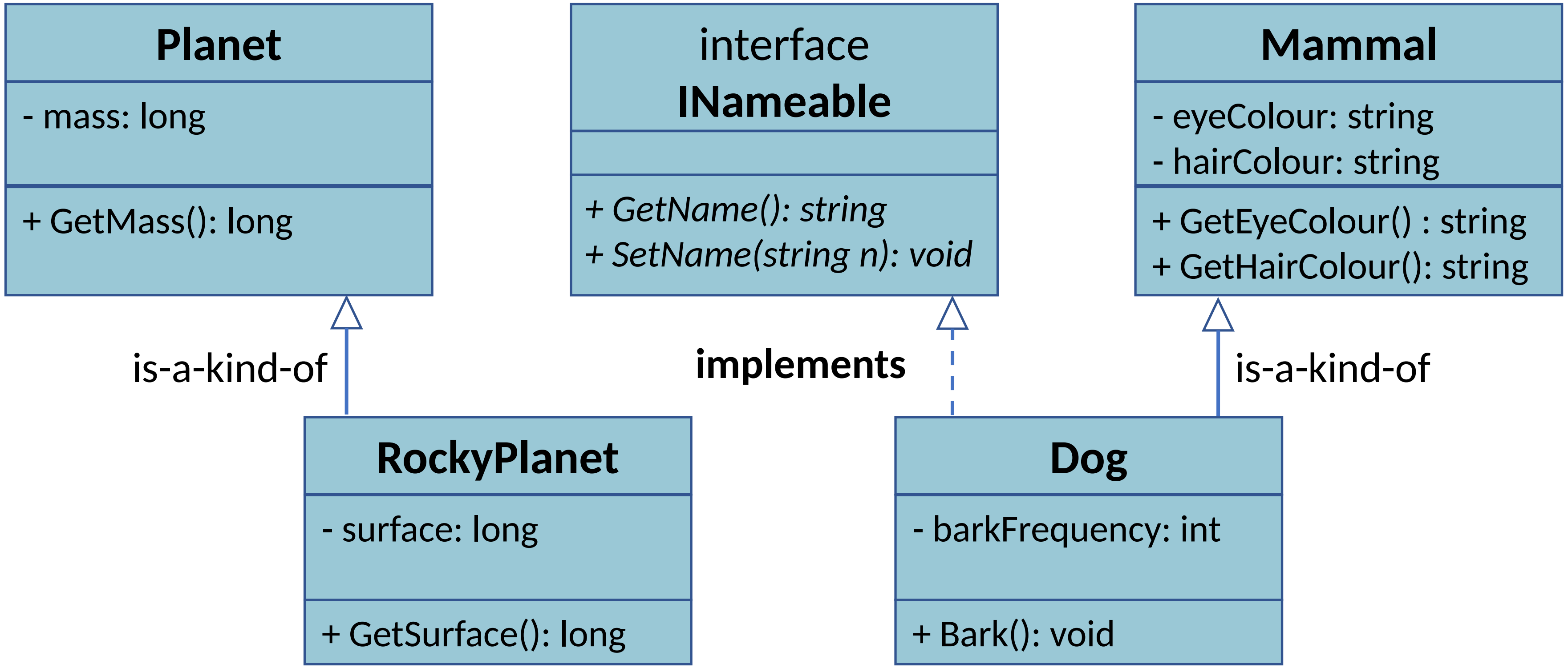


Can **INameable** be used for this design?

Interfaces


Planet
- mass: long
+ GetMass(): long
interface
INameable
+ GetName(): string
+ SetName(string n): void
Mammal
- eyeColour: string
- hairColour: string
+ GetEyeColour() : string
+ GetHairColour(): string
is-a-kind-of
is-a-kind-of
RockyPlanet
- surface: long
+ GetSurface(): long
Dog
- barkFrequency: int
+ Bark(): void

Interfaces



Planet

- mass: long

+ GetMass(): long

interface

INameable

+ GetName(): string

+ SetName(string n): void

Mammal

- eyeColour: string
- hairColour: string

+ GetEyeColour() : string

+ GetHairColour(): string

RockyPlanet

- surface: long

+ GetSurface(): long

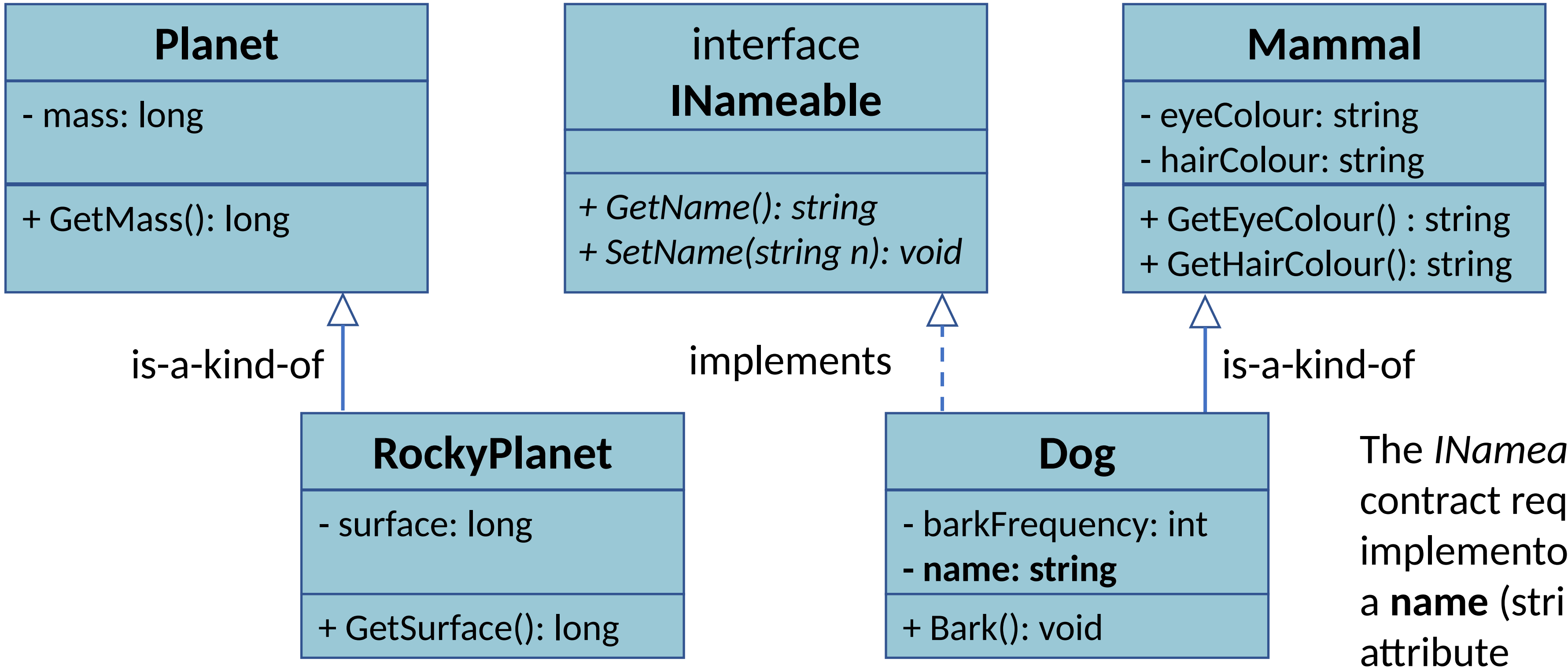
Dog

- barkFrequency: int

+ Bark(): void

implements: behaves like (dashed line)

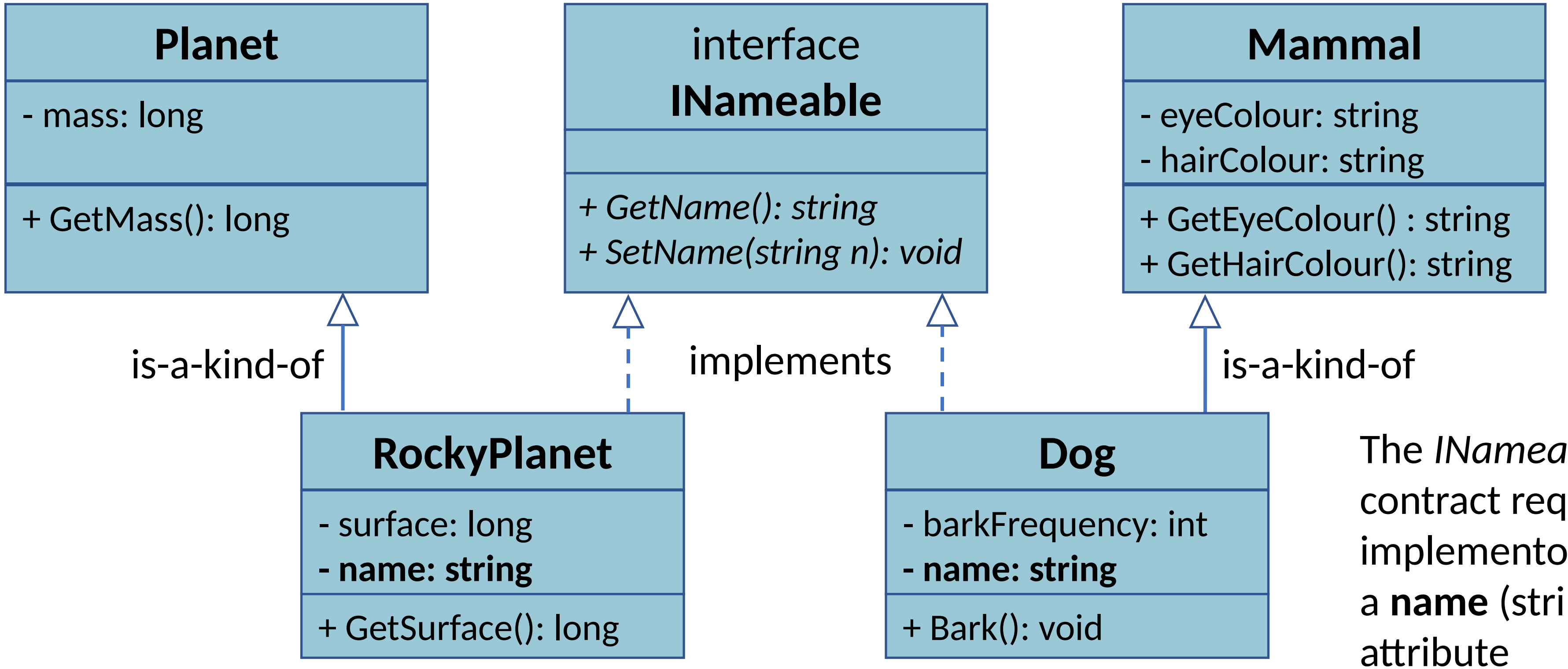
Interfaces



The *INameable* contract requires the implementor to have a **name** (string) attribute

implements: behaves like (dashed line)

Interfaces



The *INameable* contract requires the implementor to have a **name** (string) attribute

implements: behaves like (dashed line)

Interfaces: concepts

- **Class inheritance**: inheriting from a (abstract) class and **all** its **parents**

- **Interface inheritance:** no rigid and **formal inheritance structure**
- An interface could be added to **any class** where the design makes sense

Interfaces: concepts

- An interface defines methods **without implementation**
- These methods are a **design contract** to be **fulfilled**
- **Classes** that **have no connection** can fulfil the **same contract**
- Not only are *dogs* nameable, but so are *cars*, *planets*, and so on

Interfaces: example

- A class can **inherit** from **only one** *superclass*
- It can **implement one** or **many** *interfaces*

- *We will see a code example now*

Interfaces: example

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

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

    public void Bark() { … }

    // methods inherited from Mammal


}
```

Interfaces: example

```
public class Dog : Mammal, INameable // use comma
{
    int barkFrequency;
    string name;

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

    public void Bark() { … }

    // methods inherited from Mammal

    public string GetName() { return name; }

    public void SetName(string aName) { name = aName; }
}
```

INameable contract implementation

When implementing an interface contract, the implementor class does not have to use the override keyword

Interfaces: example

```
public class RockyPlanet : Planet, INameable
{
        long surface;
        string name;

        public RockyPlanet(long m, long s) : base(m) { … }

        public long GetSurface() { return surface }

        public string GetName() { return name; }

        public void SetName(string aName) { name = aName; }
}
```

INameable contract implementation

Interfaces: program version *one*

```
public static class NameLogger
{
        public static void Log(INameable nameable)
        {
                Console.WriteLine(nameable.GetName());
                // also log to a file…
        }
}
```

```
public class Program
{
    public static Main(string[] args)
    {
        Dog dog1 = new Dog("brown", "white", 10);
        dog1.SetName("Alan");
        dog1.Bark();

        RockyPlanet planet1 = new
                        RockyPlanet(1000000, 200000);

         planet1.SetName("Earth");
         Console.WriteLine(planet1.GetMass());

         NameLogger.Log(dog1);
         NameLogger.Log(planet1);
    }
}
```

From Replit

Interfaces: program version *two*

```
public static class NameLogger
{
    public static void Log(INameable nameable)
    {
        Console.WriteLine(nameable.GetName());
        // also log to a file...
    }
}
```

```
public class Program
{
    public static Main(string[] args)
    {
        INameable dog1 = new Dog("brown", "white", 10);
        dog1.SetName("Alan");
        dog1.Bark();

        RockyPlanet planet1 = new
                    RockyPlanet(1000000, 200000);

         planet1.SetName("Earth");
         Console.WriteLine(planet1.GetMass());

         NameLogger.Log(dog1);
         NameLogger.Log(planet1);
    }
}
```

Will this program version **work**?

Interfaces: summary

- A class can **inherit** from **only one** *superclass*, e.g., *Mammal*
- It can **implement many** *interfaces*, e.g., *INameable*, *IComparable*, etc.
- Completely unrelated classes, e.g., *Dog*, *Planet*, etc. can implement the same interface (**design contract**), e.g., *INameable*

Interfaces: summary

- A class can **inherit** from **only one** *superclass*, e.g., *Mammal*
- It can **implement many** *interfaces*, e.g., *INameable*, *IComparable*, etc.
- Completely unrelated classes, e.g., *Dog*, *Planet*, etc. can implement the same interface (**design contract**), e.g., *INameable*

- A *reference variable* of an *interface type* can hold references to **different objects** that implement **that interface**
- Only the **interface methods can** be **invoked** via that *reference variable*

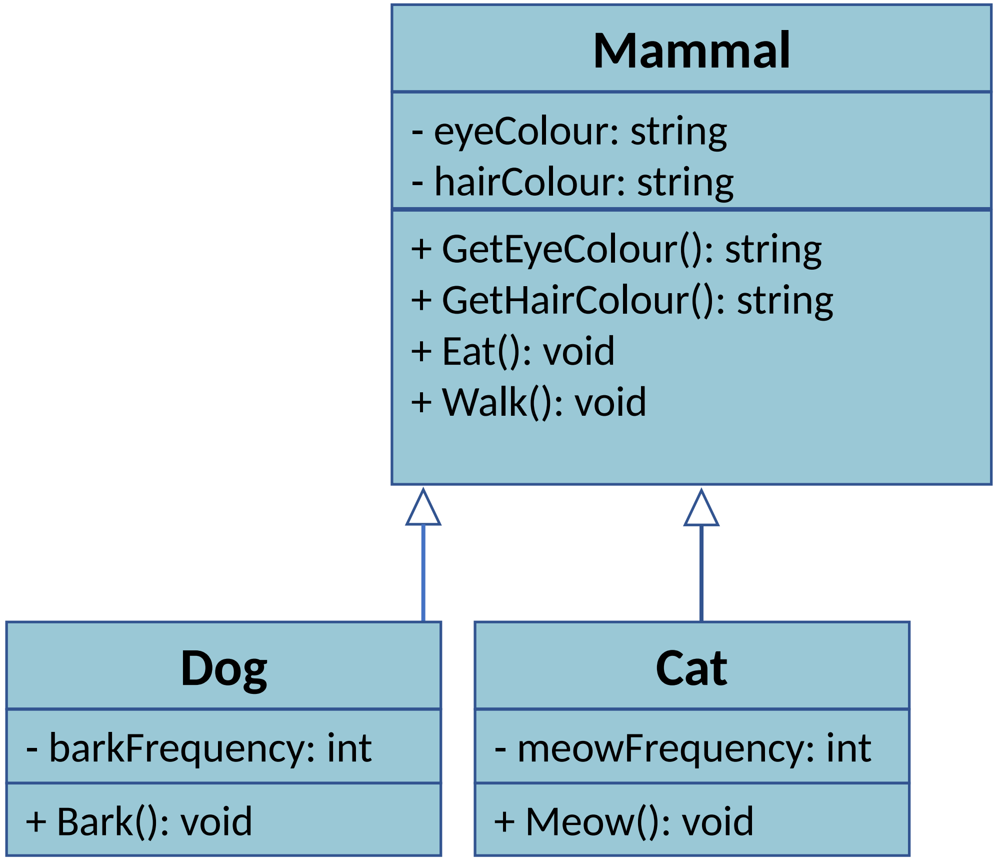
Outline

- Design Contracts
  - Summary of abstract classes
  - Interfaces
- C# inheritance tree: Object class

Inheritance: drawbacks

- **Rigid** and not **flexible** inheritance hierarchy: *RockyPlanet* is not a *Dog*
- Class inheritance **breaks** encapsulation: *issues* if *not a true is-a-kind-of* —**changes** to a superclass can have a **ripple effect** on subclasses

Inheritance: Mammal Hierarchy



Mammal

- eyeColour: string
- hairColour: string

+ GetEyeColour(): string
+ GetHairColour(): string
+ Eat(): void
+ Walk(): void

Dog

- barkFrequency: int

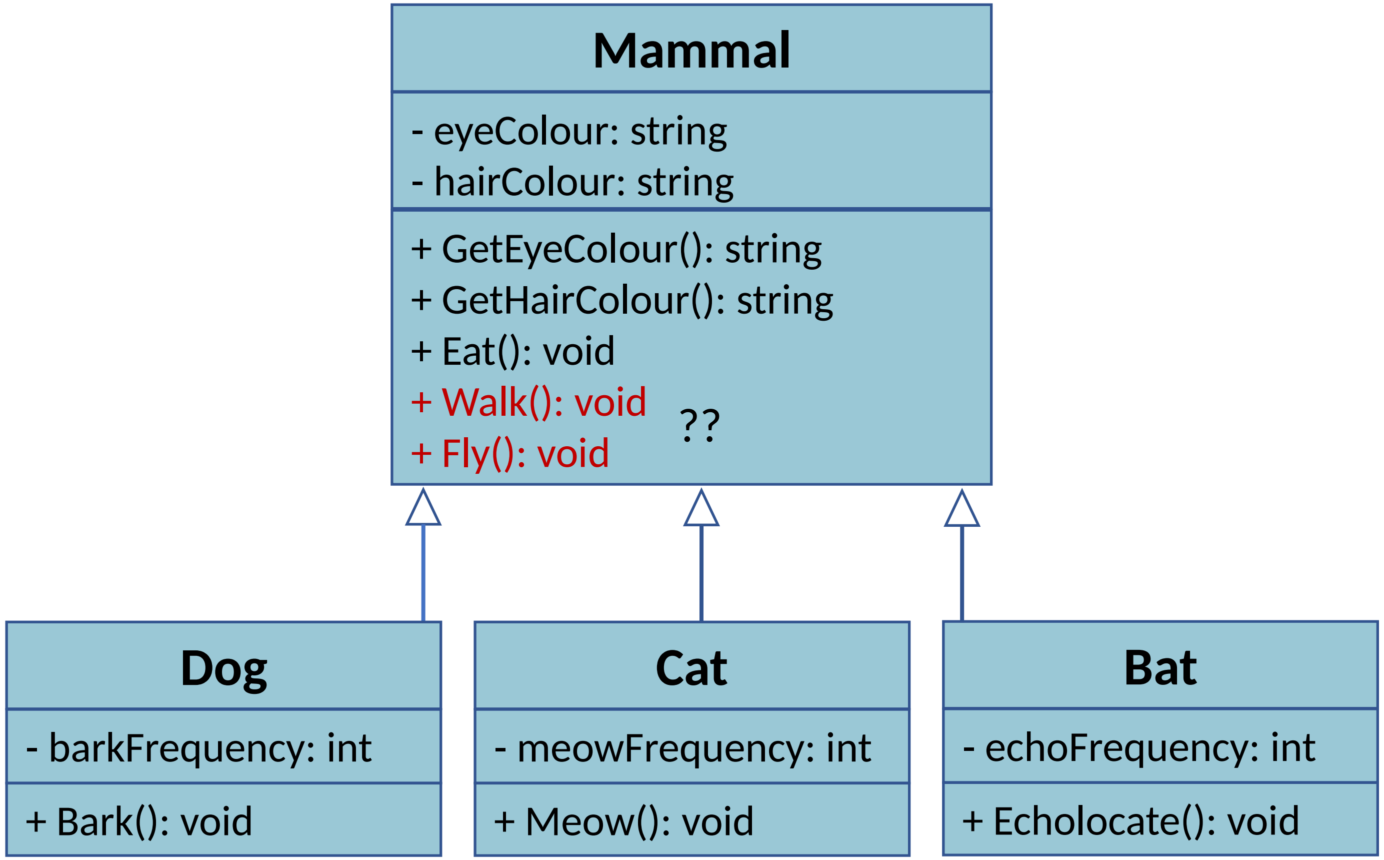
+ Bark(): void

Cat

- meowFrequency: int

+ Meow(): void

Inheritance: Mammal Hierarchy



Mammal

- eyeColour: string
- hairColour: string

+ GetEyeColour(): string
+ GetHairColour(): string
+ Eat(): void
+ Walk(): void
+ Fly(): void

??

Dog

- barkFrequency: int

+ Bark(): void

Cat

- meowFrequency: int

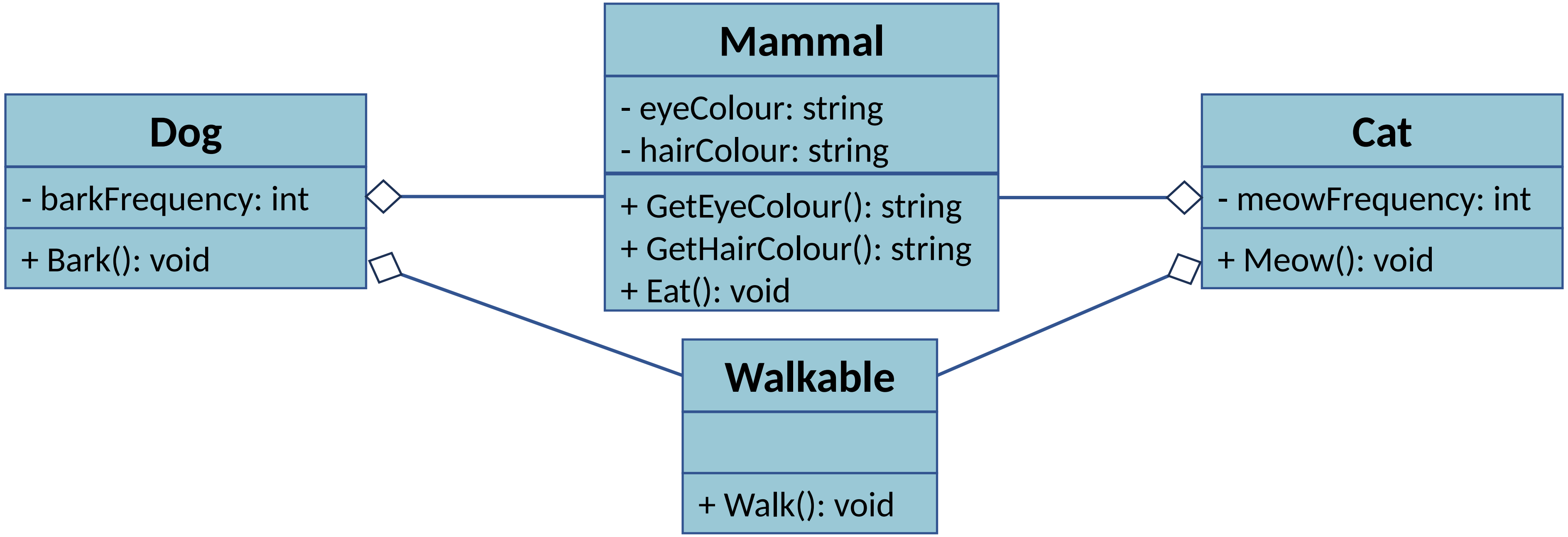
+ Meow(): void

Bat

- echoFrequency: int

+ Echolocate(): void

Inheritance versus aggregation



Dog

- barkFrequency: int

+ Bark(): void

Mammal

- eyeColour: string
- hairColour: string

+ GetEyeColour(): string
+ GetHairColour(): string
+ Eat(): void

Cat

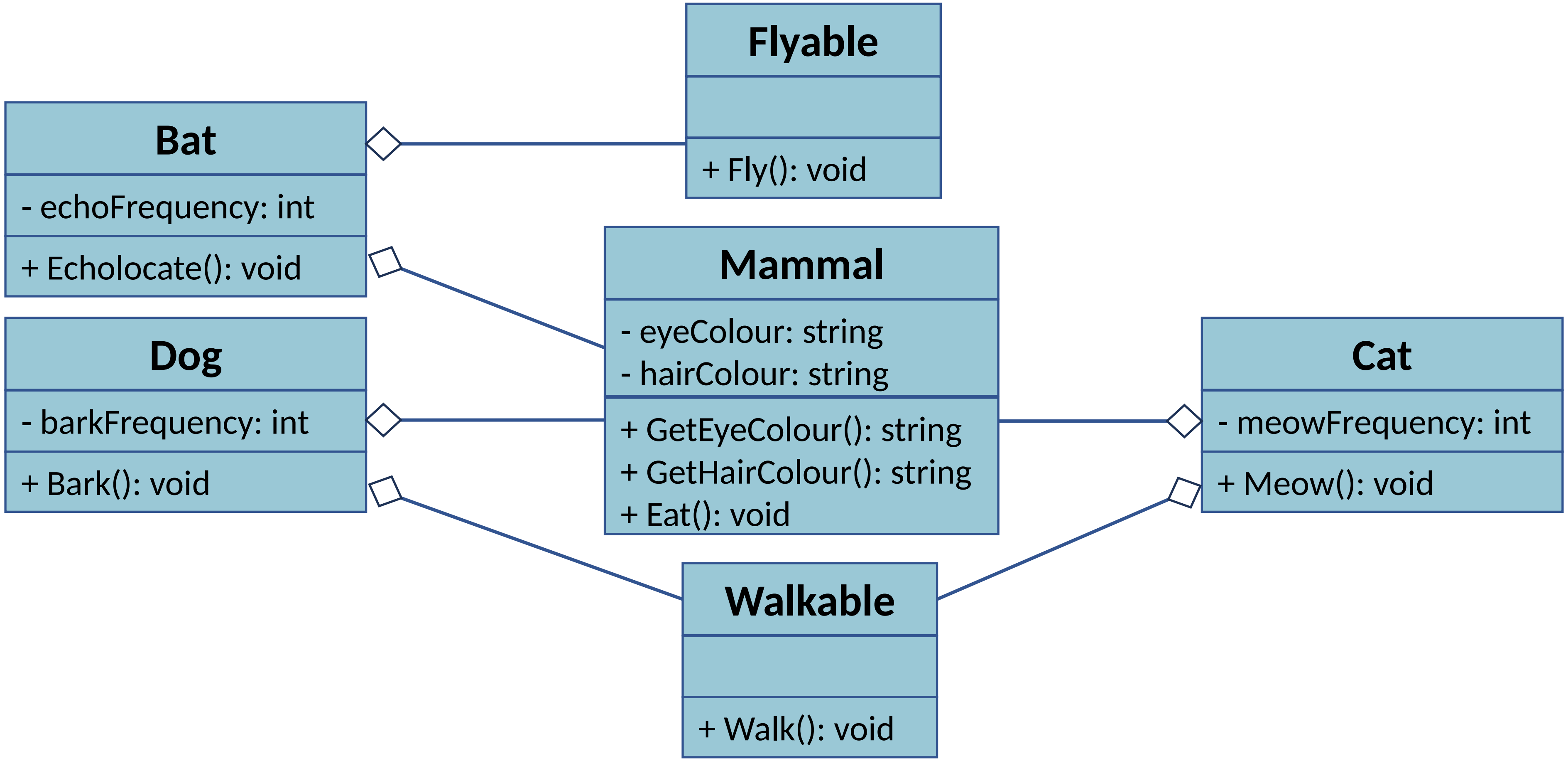
- meowFrequency: int

+ Meow(): void

Walkable

+ Walk(): void

Inheritance versus aggregation



Inheritance: drawbacks

- From the book Effective Java
  - Item 18: Favour aggregation and composition over inheritance
  - Item 19: Design and document for inheritance or else prohibit it
  - Item 20: Prefer interfaces to abstract classes
- From the book Object-Oriented Thought Process
  - 11 – Avoiding Dependencies and Highly Coupled Classes

sealed classes

sealed classes: **cannot** be **extended** (subclassed)

```
sealed class GoldenRetriever
{


}
```

```
class GoldenApso: GoldenRetriever
{


}
```

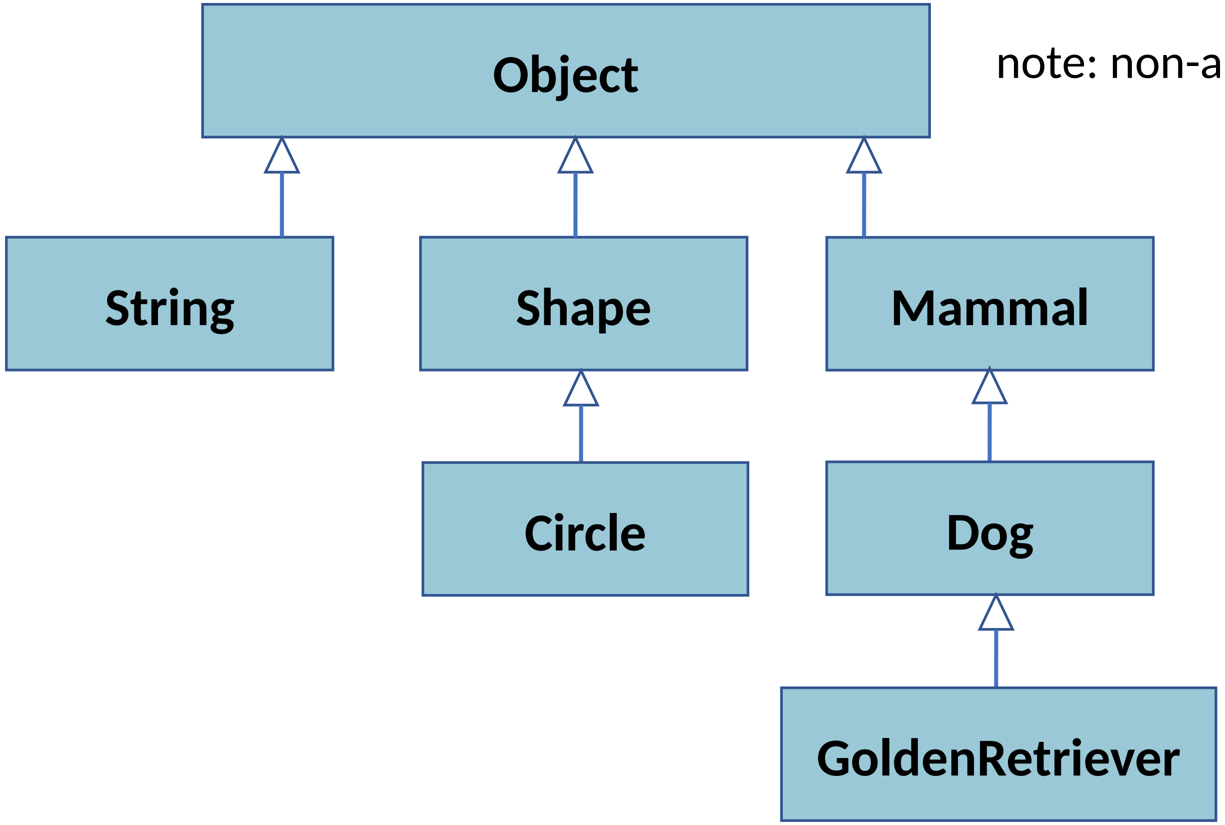
Compiler Error CS0509: *GoldenApso* cannot derive from sealed type *GoldenRetriever*

Reflection

- **Why** did we study *class inheritance*?

C# inheritance tree: *Object* class

- All C# classes you will ever use or write extend the *System*.*Object* class



note: non-abstract class

C# inheritance tree: *Object* class

```
public class Mammal : Object
{
    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
}
```

We can invoke a constructor of a superclass using base(…), with the specific number of parameters

C# inheritance tree: *Object* class

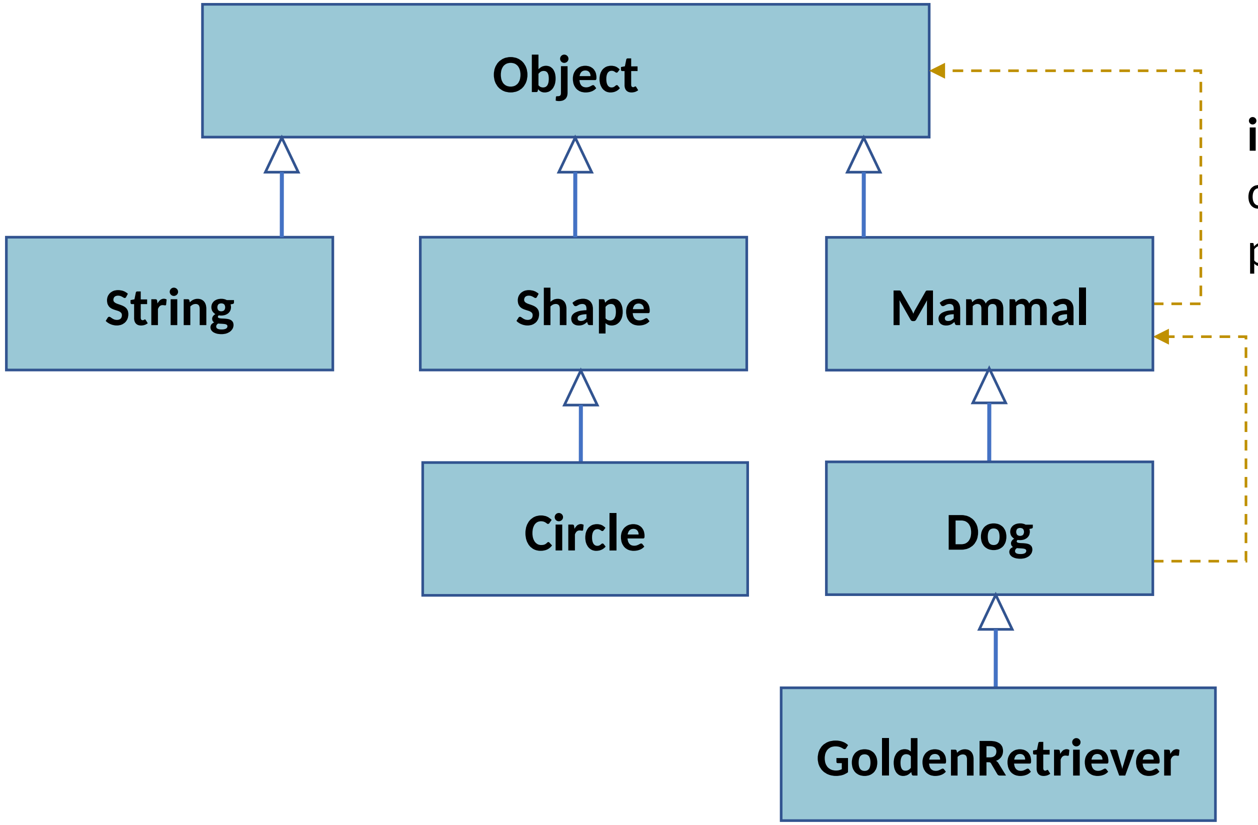
```
public class Mammal : Object
{
    private string eyeColour;
    private string hairColour;

    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) //  ???
    {
       barkFrequency = bf;
    }

    public void Bark()
    {
        // uses barkFrequency
    }

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

We can invoke a constructor of a superclass using base(…), with the specific number of parameters

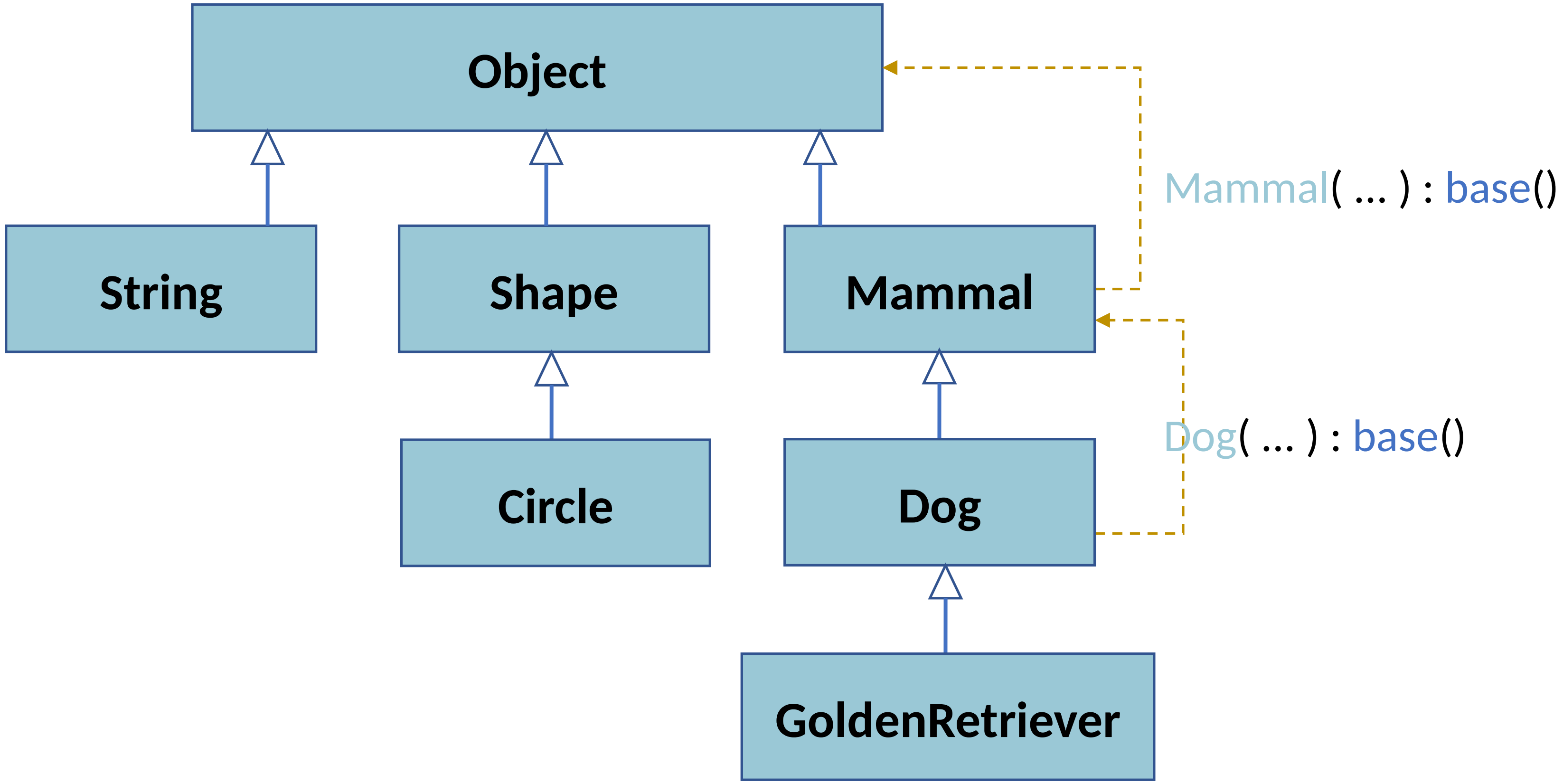
What happens if we do not do it explicitly?

C# inheritance tree: *Object* class



implicit invocation of constructors with zero parameters

C# inheritance tree: *Object* class



C# inheritance tree: *Object* class

```
public class Mammal : Object
{
    private string eyeColour;
    private string hairColour;

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

    public void Bark()
    {
        // uses barkFrequency
    }

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

Implicit invocation of *base*() (from the upper superclass)

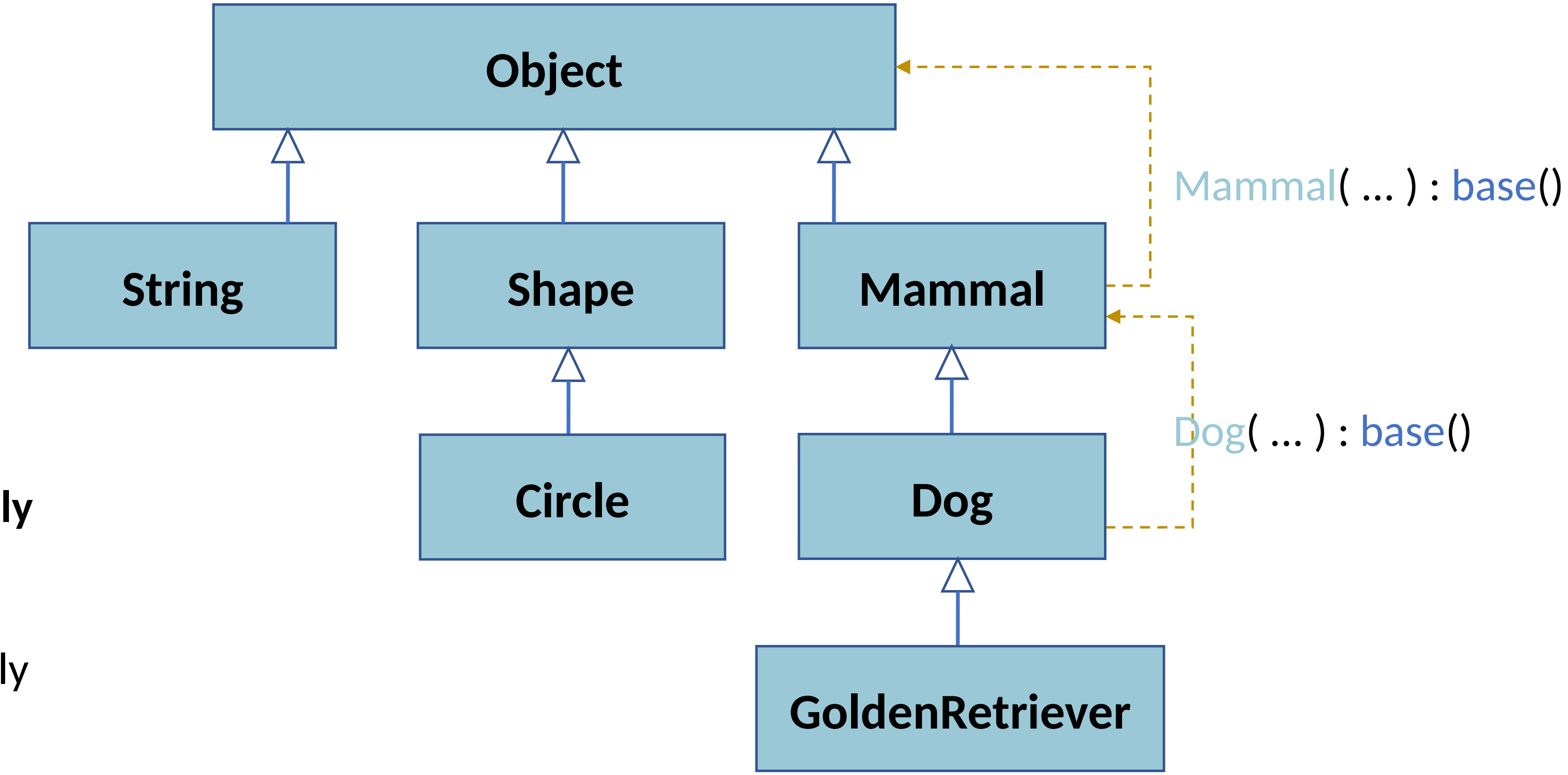
C# inheritance tree: *Object* class

```
public class Mammal : Object
{
    private string eyeColour;
    private string hairColour;

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

    public void Bark()
    {
        // uses barkFrequency
    }

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

Does *Mammal* have a constructor public *Mammal*()?

C# inheritance tree: *Object* class



if a class does not explicitly define <u>any</u> constructors:
An *empty zero-parameter constructor* is automatically created at compile time

C# inheritance tree: *Object* class

- **Remember**: an instance of a **subclass** can be **assigned** to a reference variable of its **superclass hierarchy**

C# inheritance tree: *Object* class

- **Remember**: an instance of a **subclass** can be **assigned** to a reference variable of its **superclass hierarchy**
- Object can act as a sort of *universal container*: a variable of this type can hold a reference to almost anything
- **All** the classes we write will **inherit** the public (and protected) methods defined in the Object class

C# inheritance tree: *Object* class

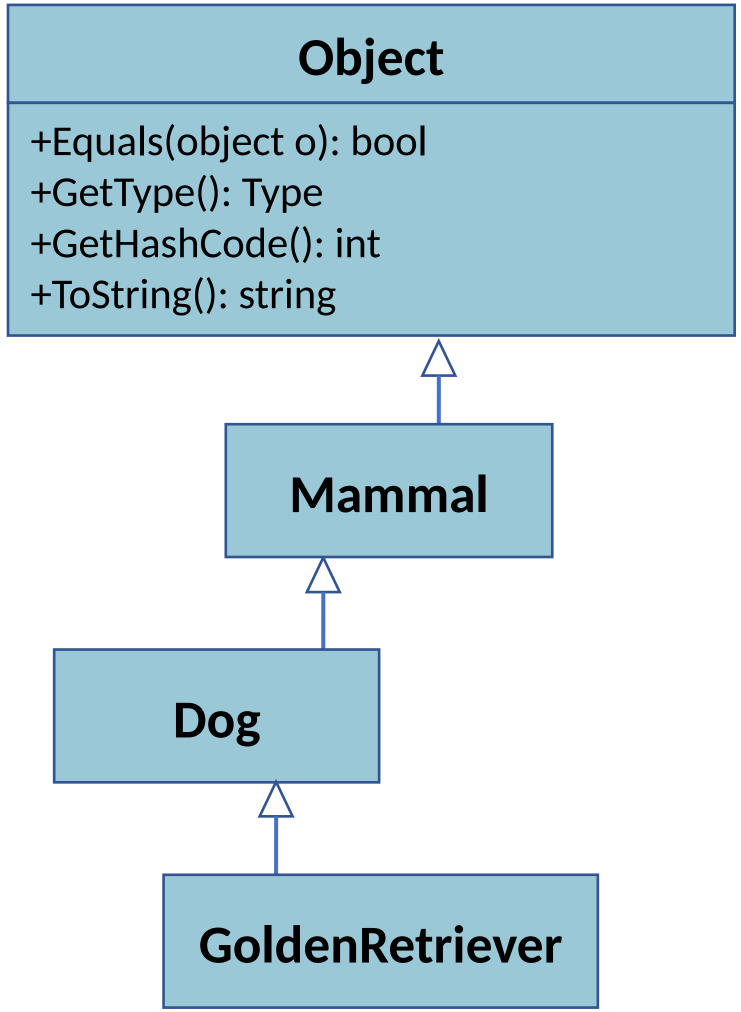
- They inherit all the methods of the *Object* class
  - *Equals(object obj)*: returns true if this object is equal to *obj*
  - *GetType()*: returns the type of the object
  - *GetHashCode()*: returns an int (hash) of the object
  - *ToString()*: returns a string representing the object

Object class: ToString()

- returns a readable textual representation of the object
- It is called automatically when an object is provided as the argument of Console.WriteLine(...)

```
Circle circle1 = new Circle( ... );
Console.WriteLine(circle1); // implicitly calls circle1.ToString()
```

Object class: ToString()



Object

+Equals(object o): bool
+GetType(): Type
+GetHashCode(): int
+ToString(): string

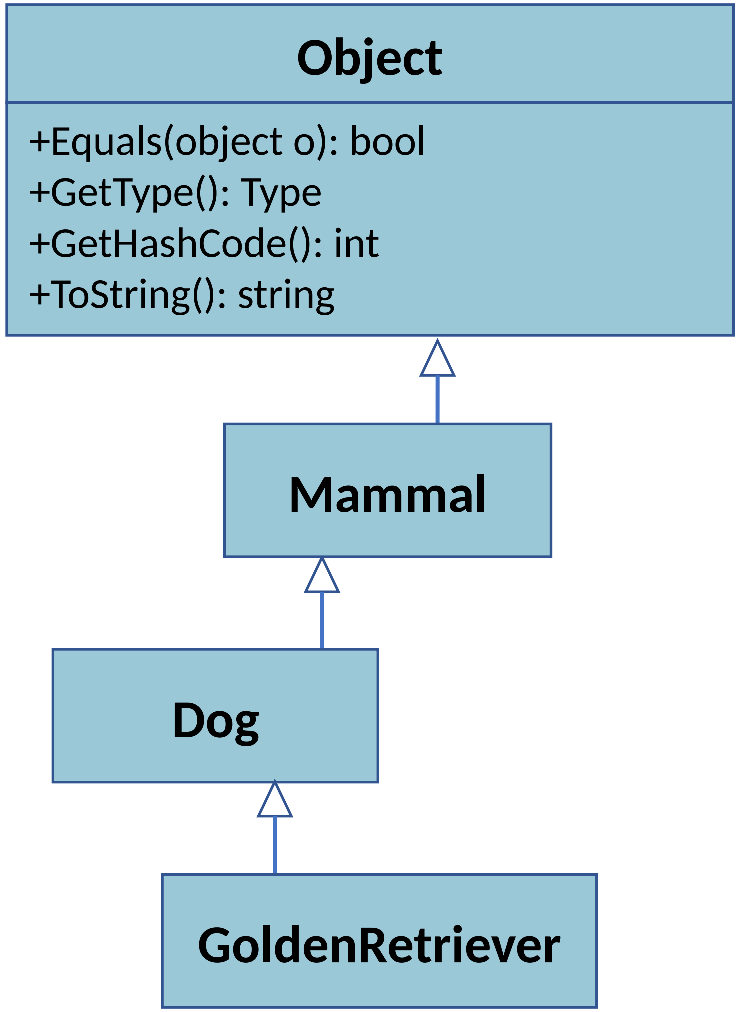
Mammal

Dog

GoldenRetriever

subclasses inherits the default implementations of these methods

Object class: ToString()



Object

+Equals(object o): bool
+GetType(): Type
+GetHashCode(): int
+ToString(): string

Mammal

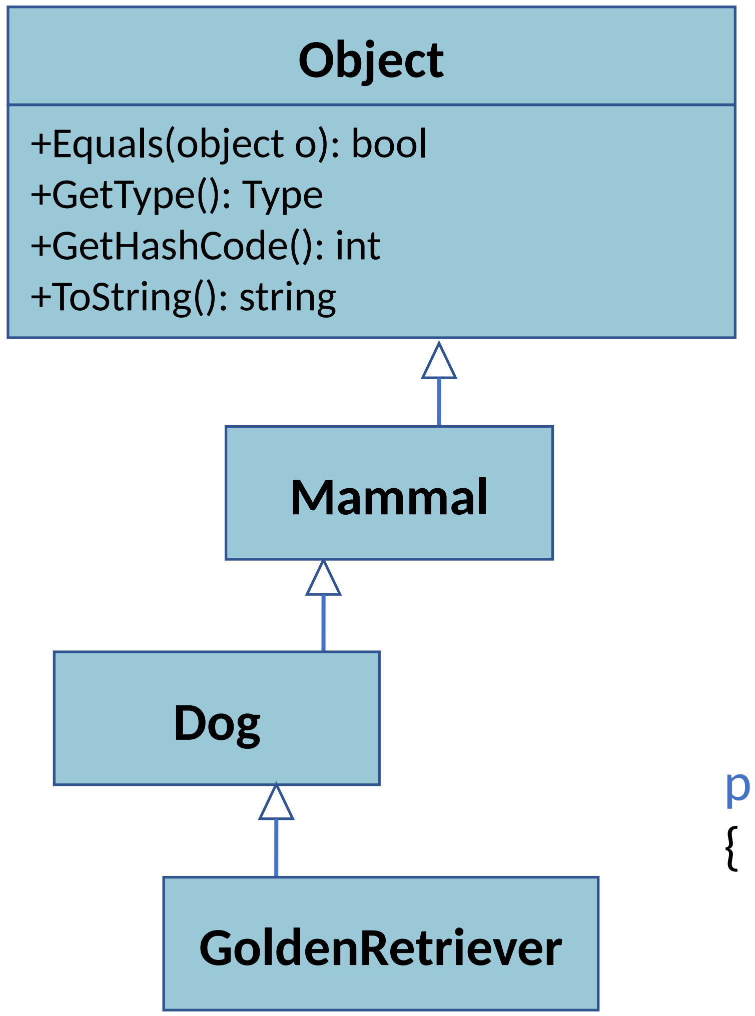
Dog

GoldenRetriever

Default implementation of the ToString method returns the fully qualified name of the object's type

(namespace.class)

Object class: ToString()



ToString method is declared as virtual so subclasses can override it

```
public override string ToString()
{
   // return a custom string that
   // represents a GoldenRetriever object
}
```

The == operator

```
class Program
{
    public static void Main(string[] args)
    {
        int a = 10;
        int b = 10;
        if (a == b)
            Console.WriteLine("a is equal to b");

        BankAccount account1 = new BankAccount("AB456", 200.0);
        BankAccount account2 = new BankAccount("AB456", 200.0);
        if (account1 == account2)
            Console.WriteLine("account1 is equal to account2");
    }
}
```

Object class: Equals()

```
class Program
{
    public static void Main(string[] args)
    {
        int a = 10;
        int b = 10;
        if (a == b)
            Console.WriteLine("a is equal to b");

        BankAccount account1 = new BankAccount("AB456", 200.0);
        BankAccount account2 = new BankAccount("AB456", 200.0);
        if (account1.Equals(account2))
            Console.WriteLine("account1 is equal to account2");
    }
}
```

Object class: Equals()

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

    public BankAccount(string num, double bal) { … }

    public void Deposit(double amount) { … }

    public double GetBalance() { … }

    …

    public override bool Equals(object obj)
    {
        // return true if this.number is equal to obj.number and this.balance is equal to obj.balance
    }
}
```

Exceptions and inheritance

- Exceptions are **objects** derived from the *Object* class
- *Inheritance* can be used to define **custom** exceptions

- Let's define a custom exception for our *previous space exploration game*

Exceptions and inheritance

```
public class SpacePlanetException : Exception
{
    public SpacePlanetException(string message) : base(message)
    {
    // initialise specific attributes
    }

   // other relevant constructors
}
```