7SENG011W Object Oriented Programming

Class Inheritance Drawbacks: Polymorphism and Design Contracts with Interfaces; the Java Object Class, Custom Exceptions

Dr Francesco Tusa

Readings

Books

- Head First Java
  - Chapter 8: Serious Polymorphism: Interfaces and Abstract Classes
- Object-Oriented Thought Process
  - Chapter 7: Mastering Inheritance and Composition
  - Chapter 8: Frameworks and Reuse: Designing with Interfaces and Abstract Classes
  - Chapter 9: Building Objects and Object-Oriented Design
  - Chapter 11: Avoiding Dependencies and Highly Coupled Classes

Online

- The Java Tutorials: Lesson on Interfaces and Inheritance
- The Java Tutorials: Lesson on Exceptions

Outline

- Recap: Design Contracts and Polymorphism with Abstract Classes
- Class Inheritance: Drawbacks
  - Weakened Encapsulation
  - Complexity of Multiple Inheritance
- Interface Inheritance
  - Abstract classes versus Interfaces
  - Comparison with Class Inheritance
  - Conclusions
- The Java Object Class
  - Object superclass methods
  - Constructors Invocation
  - Custom Exceptions with Inheritance

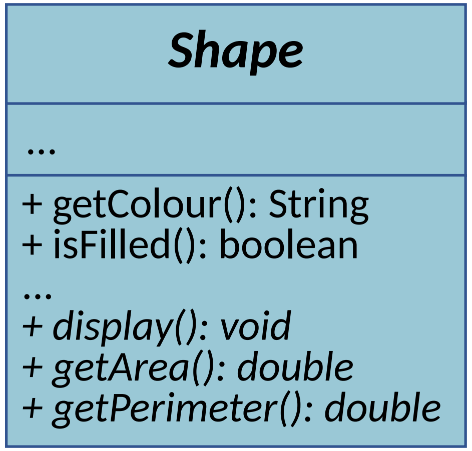
Object-Oriented Programming (OOP) Principles

- **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 (many forms)

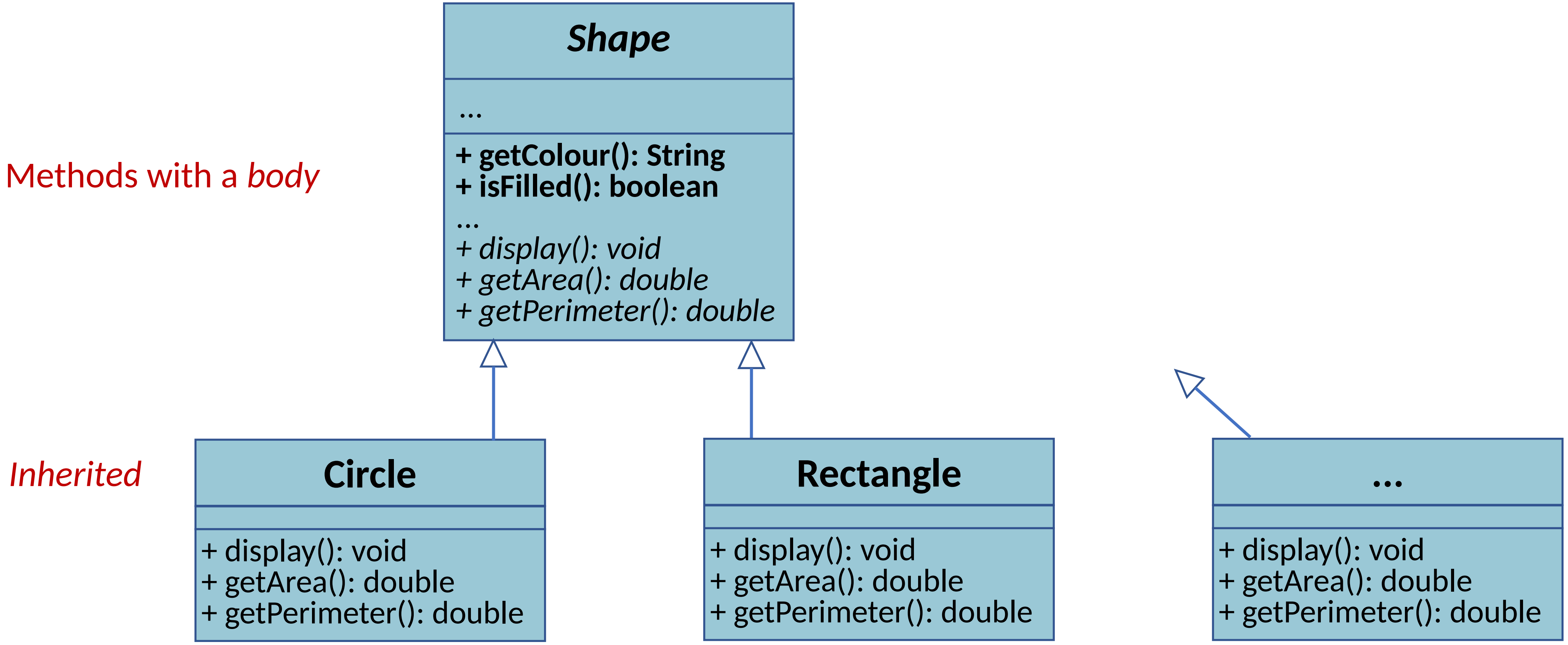
from the Greek words *"poly"* (many) and *"morph"* (form)

Defining Design Contracts: Abstract Classes



Shape abstract class we presented and used last week
(not all the concrete methods are shown)

Defining Design Contracts: Abstract Classes



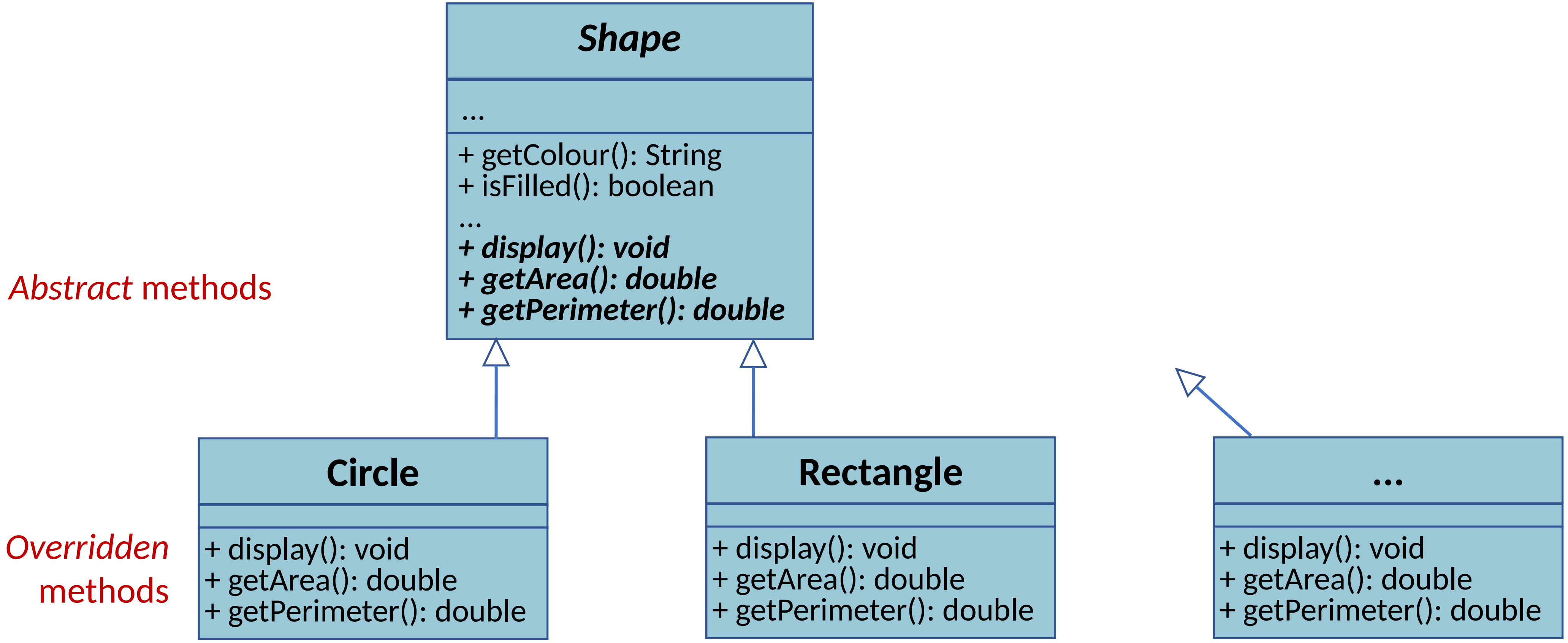
Classes attributes not represented in the diagram

Defining Design Contracts: Abstract Classes

```
public abstract class Shape
{
    private String name;
    private boolean filled;
    private String colour;

    public Shape(String c, boolean 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 defined methods that all the subclasses will **inherit**: code **reuse**

Defining Design Contracts: Abstract Classes



Classes attributes not represented in the diagram

Defining Design Contracts: Abstract Classes

```java
public abstract class Shape
{
    private String name;
    private boolean filled;
    private String colour;

    public Shape(String c, boolean 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 have **no body**: each subclass **must** implement them in a specific way

They are used to define a **design contract** that other **(sub)classes** must **fulfil**

Defining Design Contracts: Abstract Classes

```java
public abstract class Shape
{
    private String name;
    private boolean filled;
    private String colour;

    public Shape(String c, boolean 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 have **no body**: each subclass **must** implement them in a specific way

The *Shape* superclass becomes a **blueprint** for creating subclasses adhering to a **common** and **consistent structure** (interface)

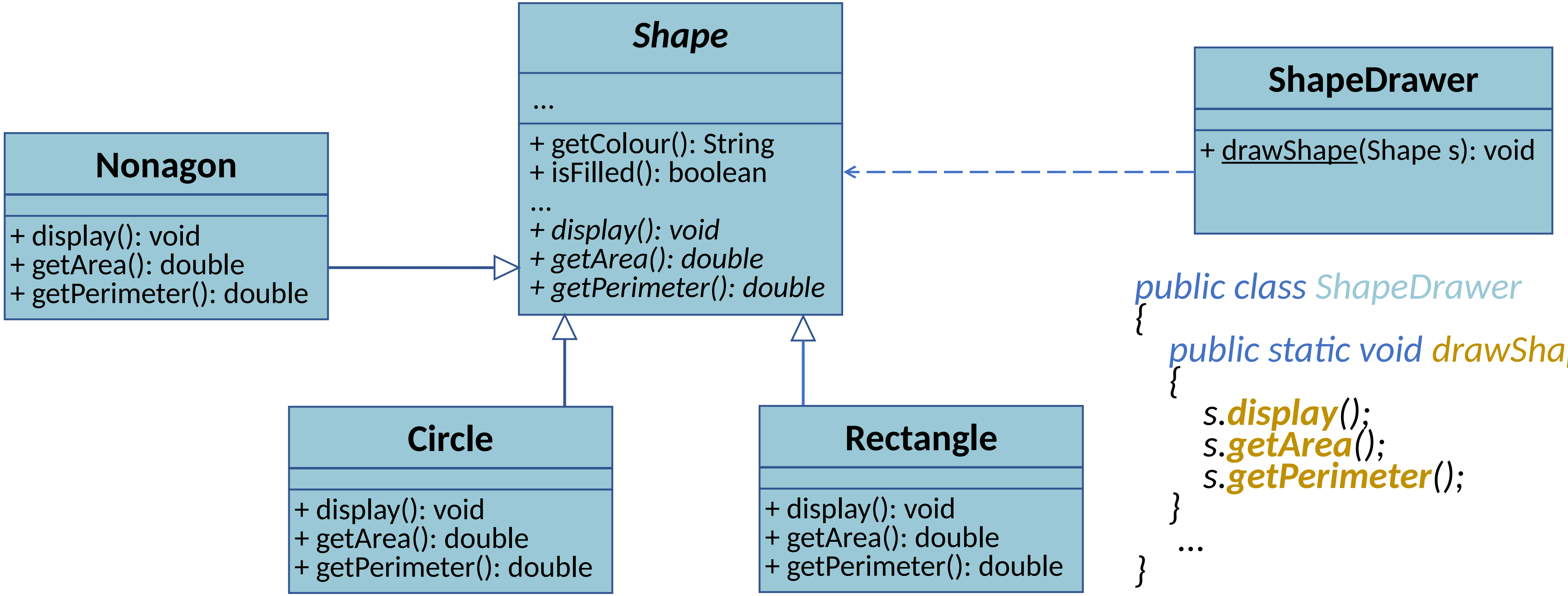
Defining Design Contracts: Abstract Classes

```
public class Rectangle extends Shape
{
    // attributes
    ...

    public Rectangle( ... ) { ... }

    public void display() {
     // specific rectangle implementation
    }

    public double getArea() {
     // specific rectangle implementation
    }

    public double getPerimeter() {
    // specific rectangle implementation
    }

}
```

Circle, *Rectangle* (and others) must provide an implementation of those methods to **fulfil the contract**.

After fulfilling the contract, instances of the *Rectangle* class can be created.

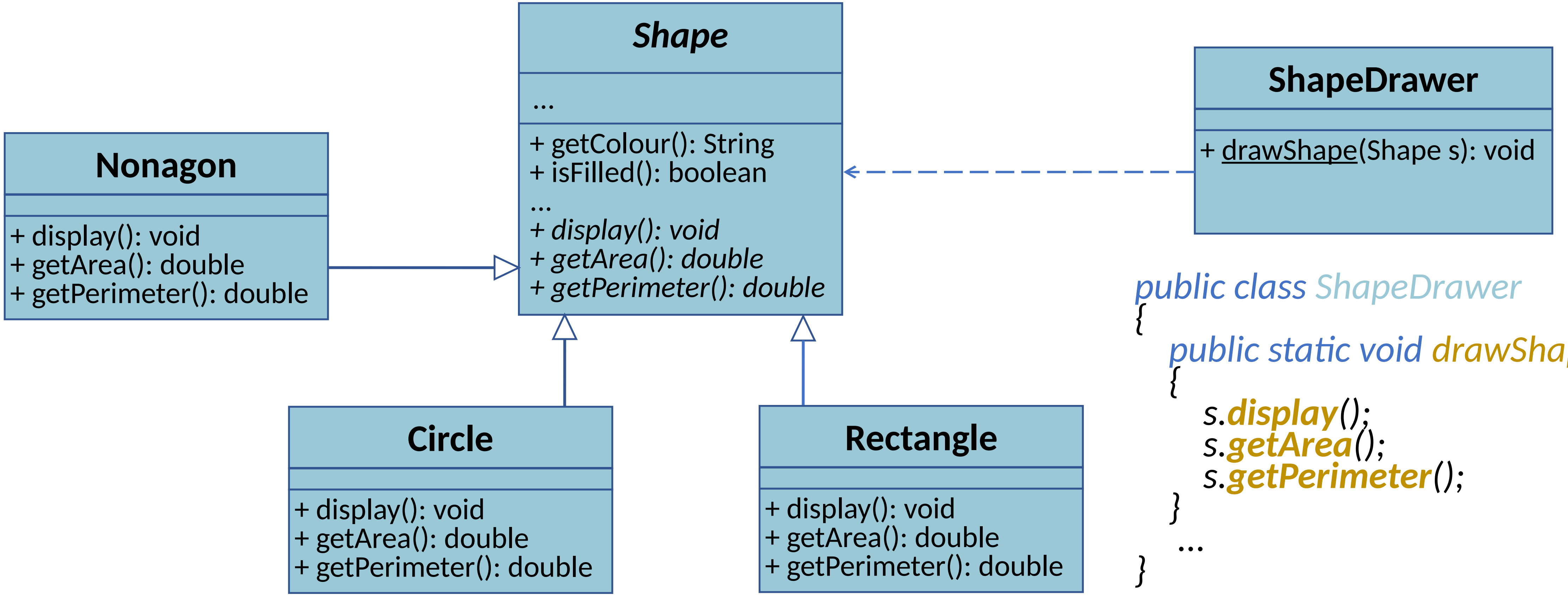
Defining Design Contracts: Abstract Classes




```
public class ShapeDrawer
{
   public static void drawShape(Shape s)
   {
      s.display();
      s.getArea();
      s.getPerimeter();
   }
    ...
}
```

The code of *ShapeDrawer* works with **any geometric shape** that fulfils the *Shape* **contract without needing changes**

Defining Design Contracts: Abstract Classes




```
public class ShapeDrawer
{
    public static void drawShape(Shape s)
    {
        s.display();
        s.getArea();
        s.getPerimeter();
    }
    ...
}
```

The actual version of the methods *display*, *getArea* and *getPerimeter* that will be executed is only **known at runtime** (late binding)—**not at compile time**

Defining Design Contracts: Summary

- **Contracts**: abstract classes define **consistent contracts** or **blueprints** for *subclasses*.
- **Code Reuse**: Polymorphism **enforces** contracts, **reducing code duplication**.
- **Flexibility**: Contracts allow **easy extension** with **new subclasses without altering** existing code.
- **Maintainability**: Contracts ensure **organised**, **readable**, and **flexible** code.

Outline

- Recap: Design Contracts and Polymorphism with Abstract Classes
- Class Inheritance: Drawbacks
  - **Weakened Encapsulation**
  - Complexity of Multiple Inheritance
- Interface Inheritance
  - Abstract classes versus Interfaces
  - Comparison with Class Inheritance
  - Conclusions
- The Java Object Class
  - Object superclass methods
  - Constructors Invocation
  - Custom Exceptions with Inheritance

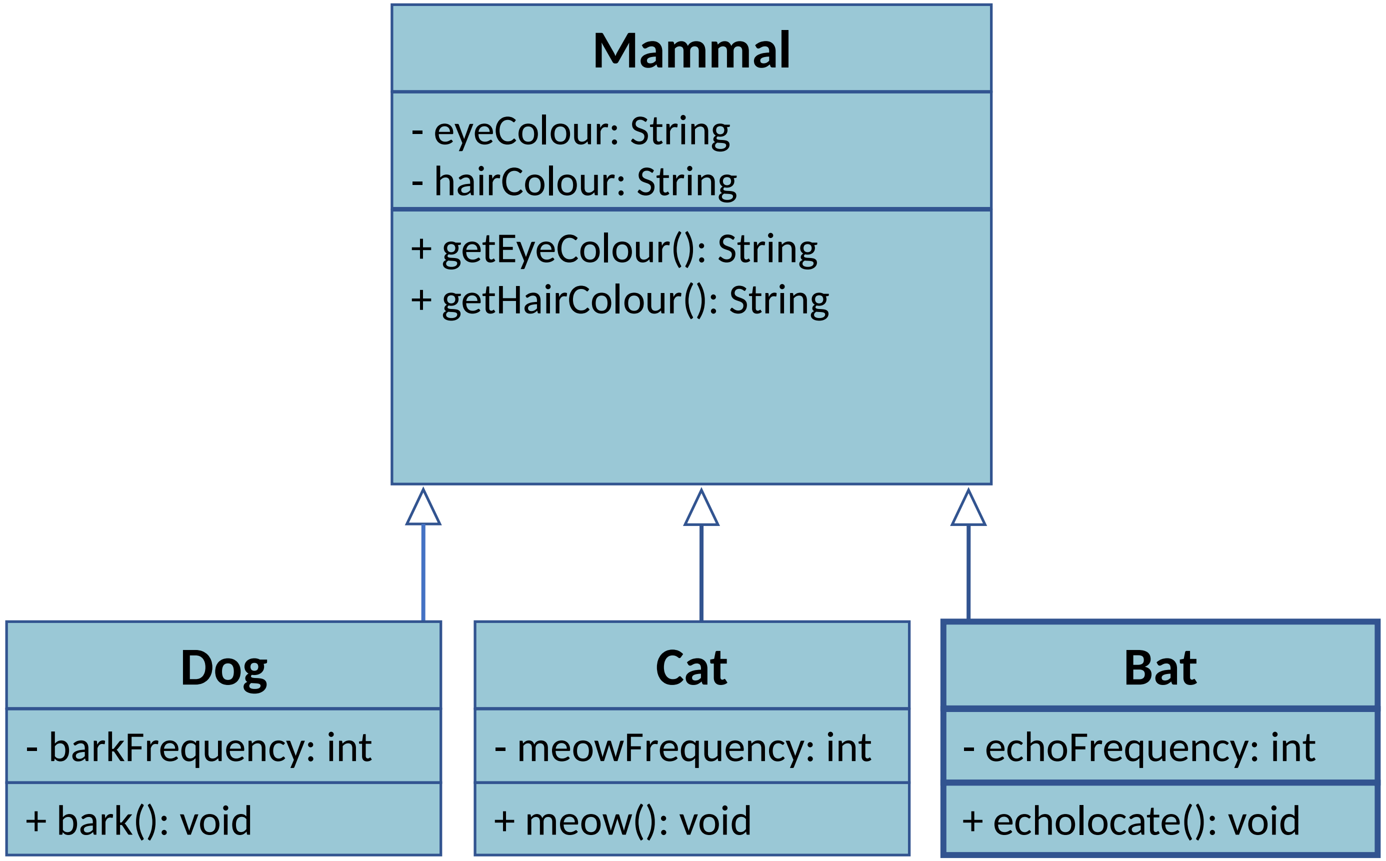
Class Inheritance: Drawbacks

- 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

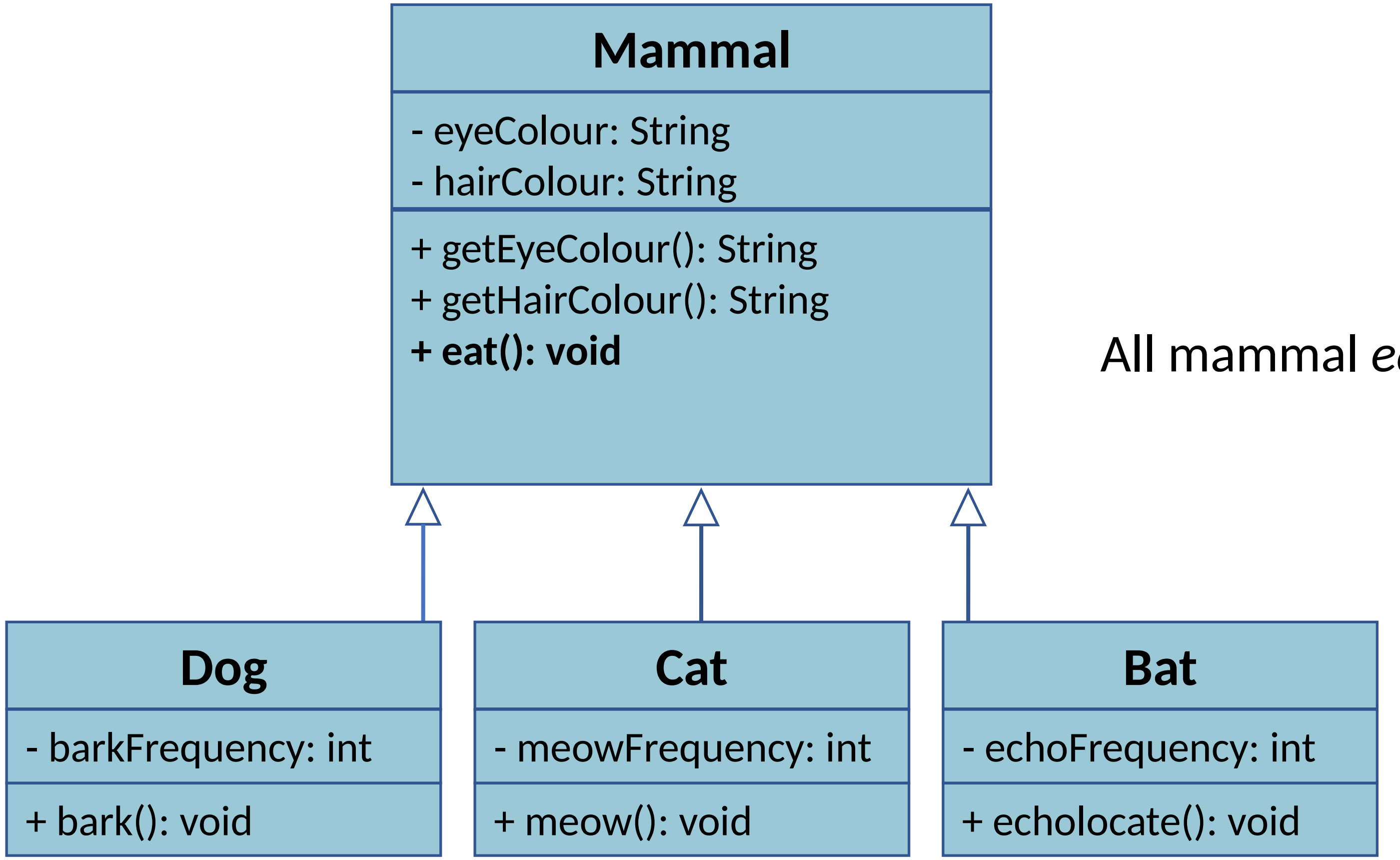
Class Inheritance: Drawbacks

- **Weakened** encapsulation: **changes** to a *superclass* can **ripple** on the *subclasses*
- **Complexity** in case of **multiple inheritance**
- Based on a **rigid** and not **flexible** inheritance hierarchy

Class Inheritance: Weakened Encapsulation

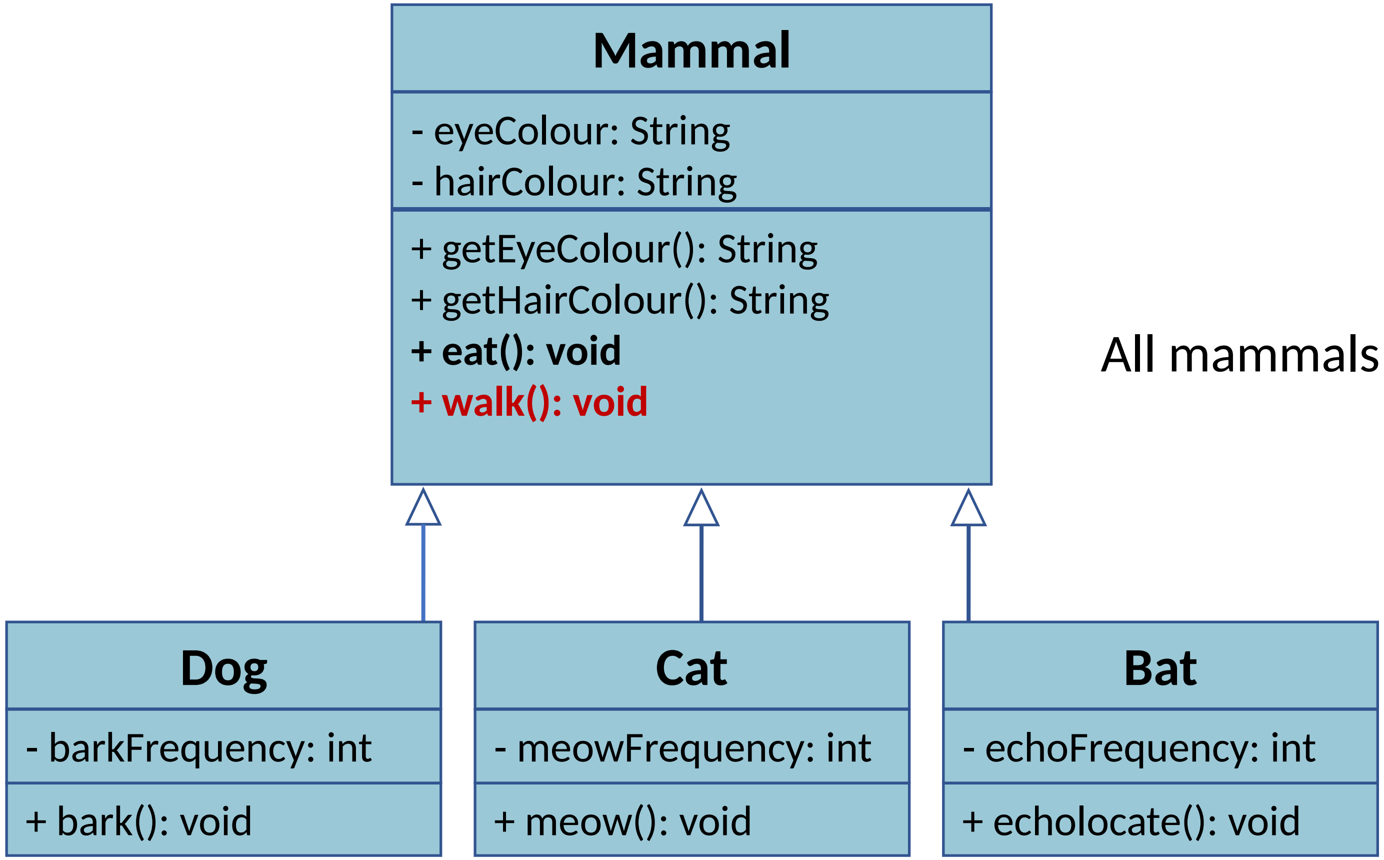


Class Inheritance: Weakened Encapsulation



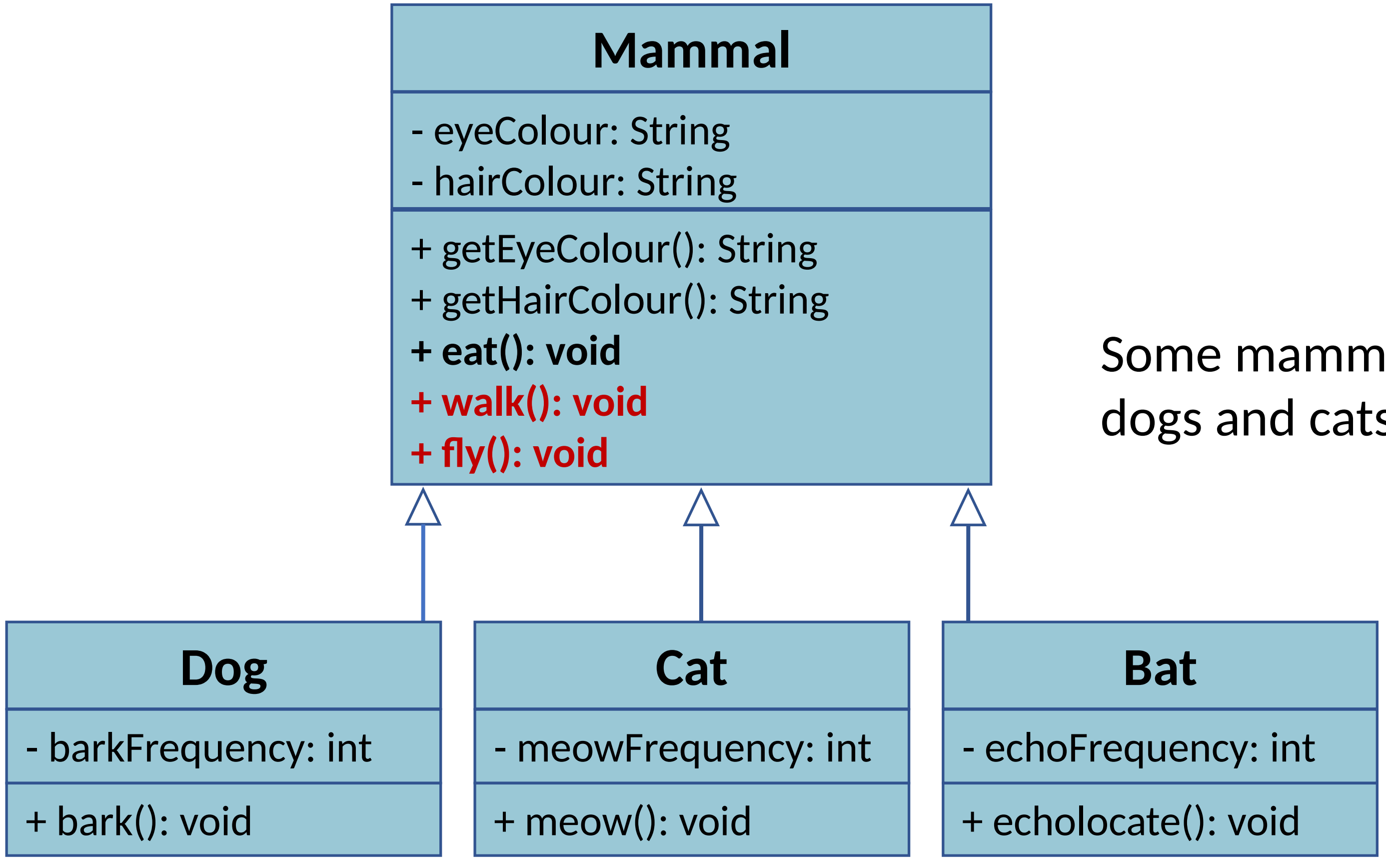
All mammal *eat*: this works fine

Class Inheritance: Weakened Encapsulation



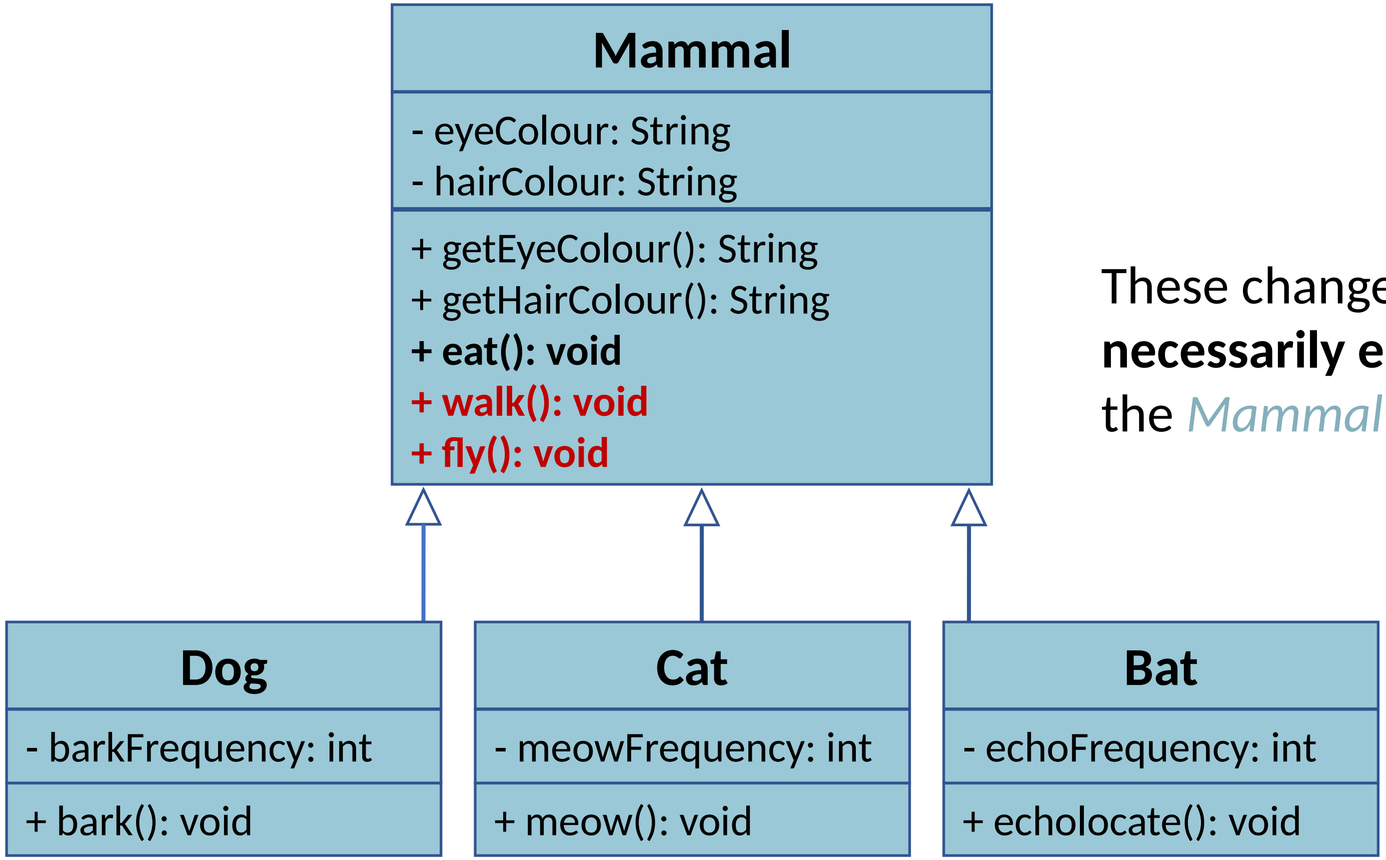
All mammals *walk*, not the bat!

Class Inheritance: Weakened Encapsulation



Some mammals *fly*, but not dogs and cats!

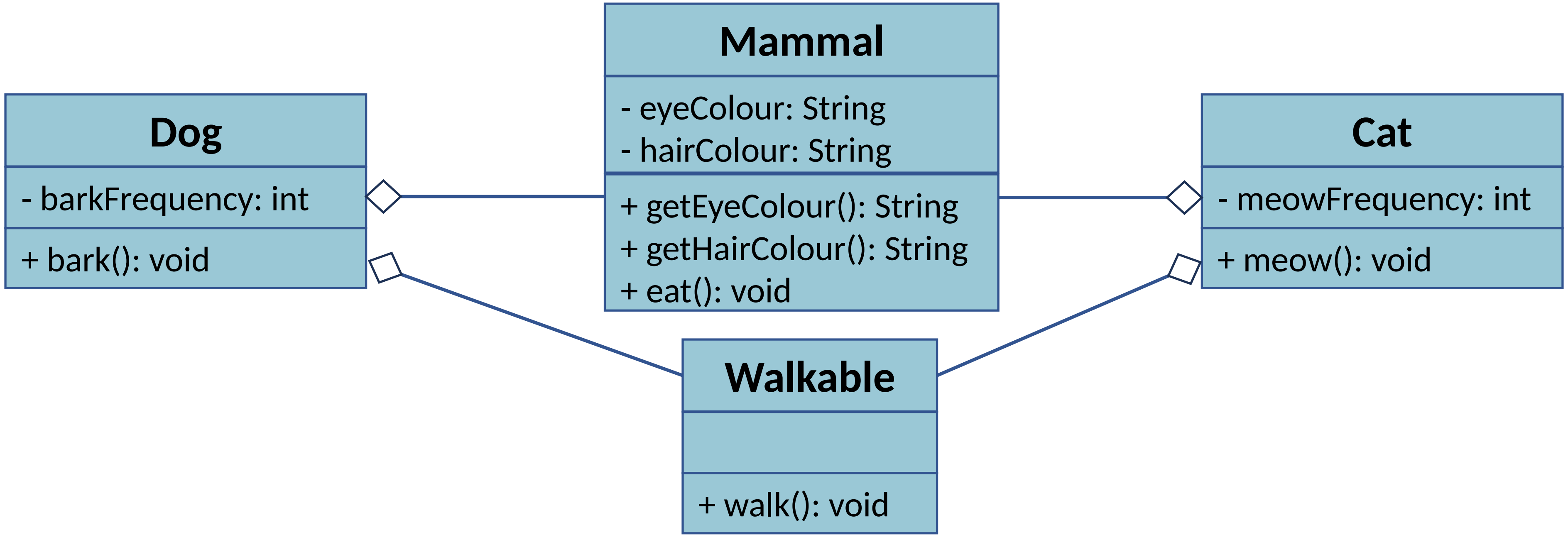
Class Inheritance: Weakened Encapsulation



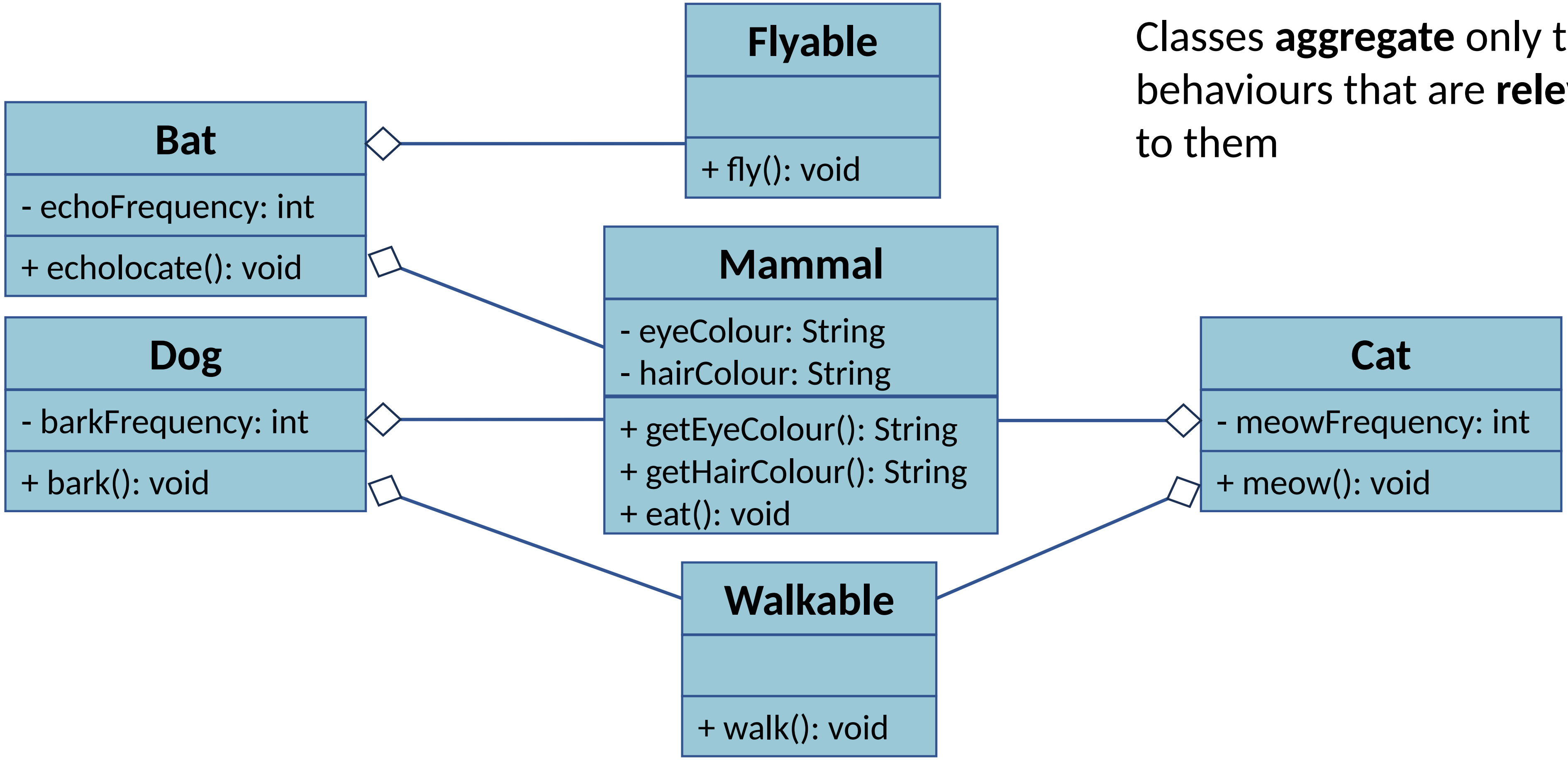
These changes **are not necessarily encapsulated** in the *Mammal* class

Rippling effect: the subclasses’ code should be **retested** and likely **updated**

Inheritance versus Aggregation



Inheritance versus Aggregation



Classes **aggregate** only the behaviours that are **relevant** to them

Class Inheritance: final classes

- 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

```
public final class Dog {
        …
}
```

A *final* class **cannot be extended** by other classes

Outline

- Recap: Design Contracts and Polymorphism with Abstract Classes
- Class Inheritance: Drawbacks
  - Weakened Encapsulation
  - **Complexity of Multiple Inheritance**
- Interface Inheritance
  - Abstract classes versus Interfaces
  - Comparison with Class Inheritance
  - Conclusions
- The Java Object Class
  - Object superclass methods
  - Constructors Invocation
  - Custom Exceptions with Inheritance

Class Inheritance: Drawbacks

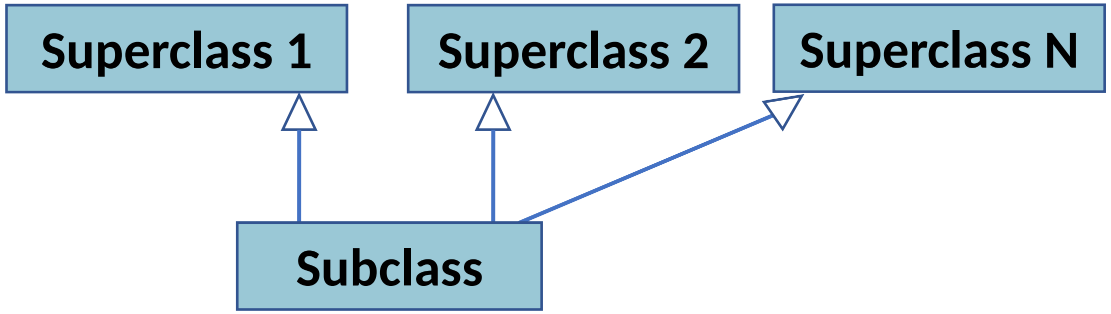
- **Weakened** encapsulation: **changes** to a *superclass* can **ripple** on the *subclasses*
- **Complexity** in case of **multiple inheritance**
- Based on a **rigid** and not **flexible** inheritance hierarchy

Multiple Inheritance

- An abstract class allows for defining a **design contract**

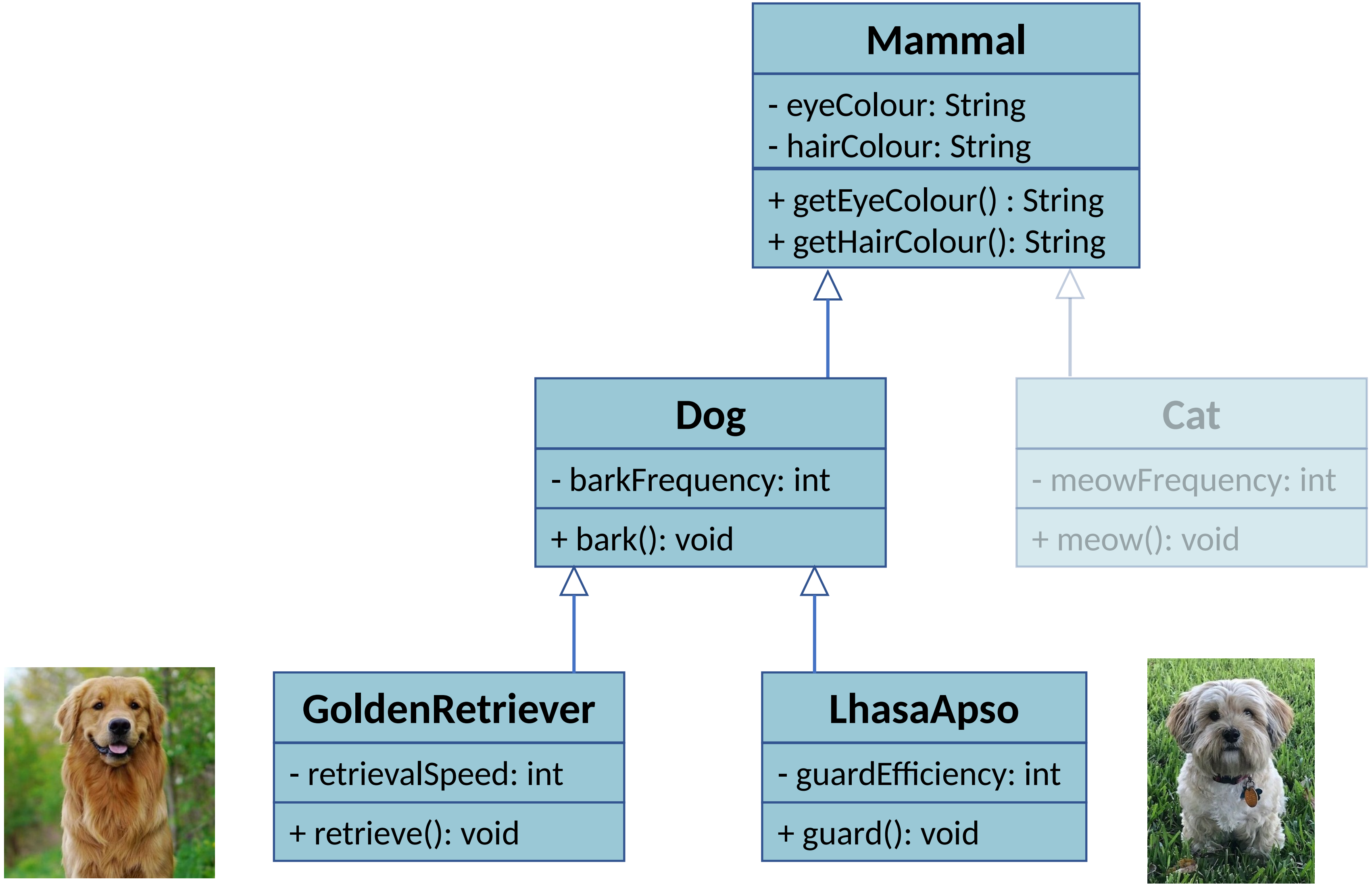
In theory:

- A subclass could inherit from **multiple** abstract superclasses, **each** describing a **design contract**

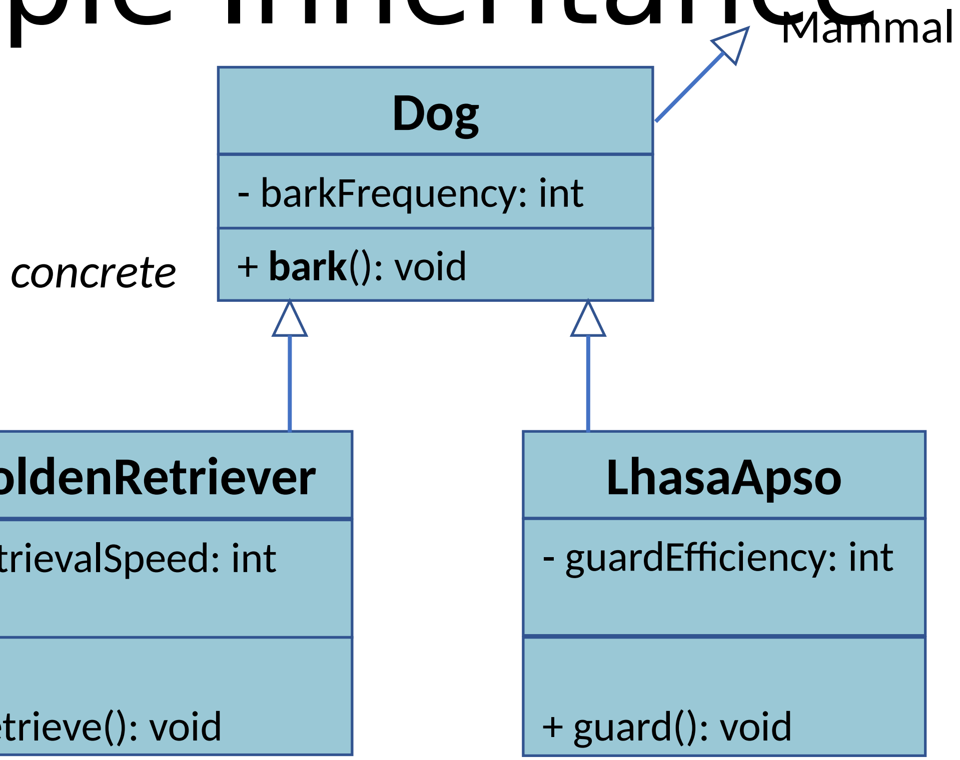


Let's see with an example how multiple inheritance would work in general

Multiple Inheritance

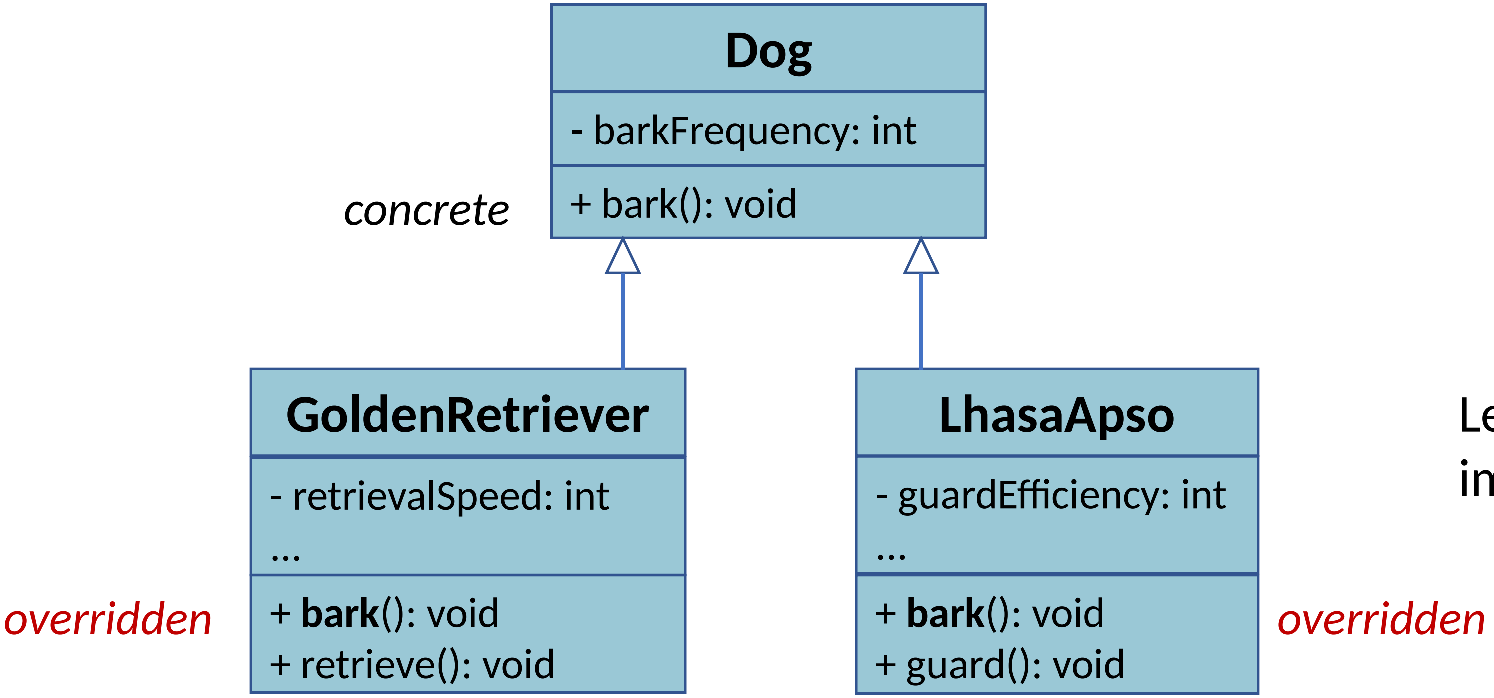


Multiple Inheritance



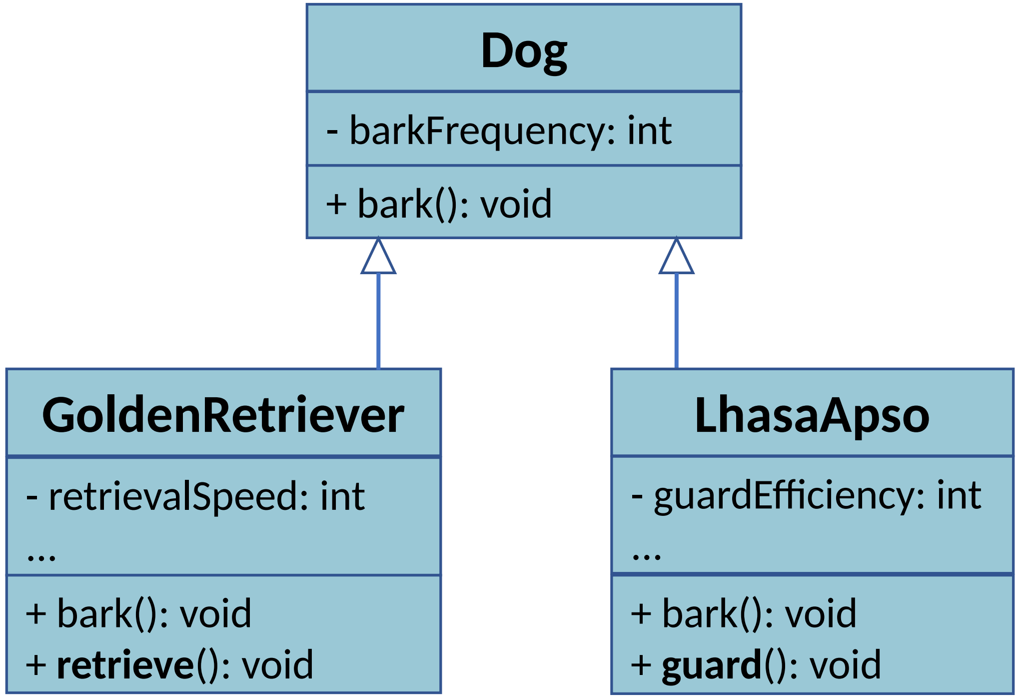
After testing, we realised *bark* should **not** be the **same** in the subclasses

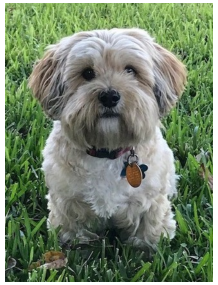
Multiple Inheritance



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

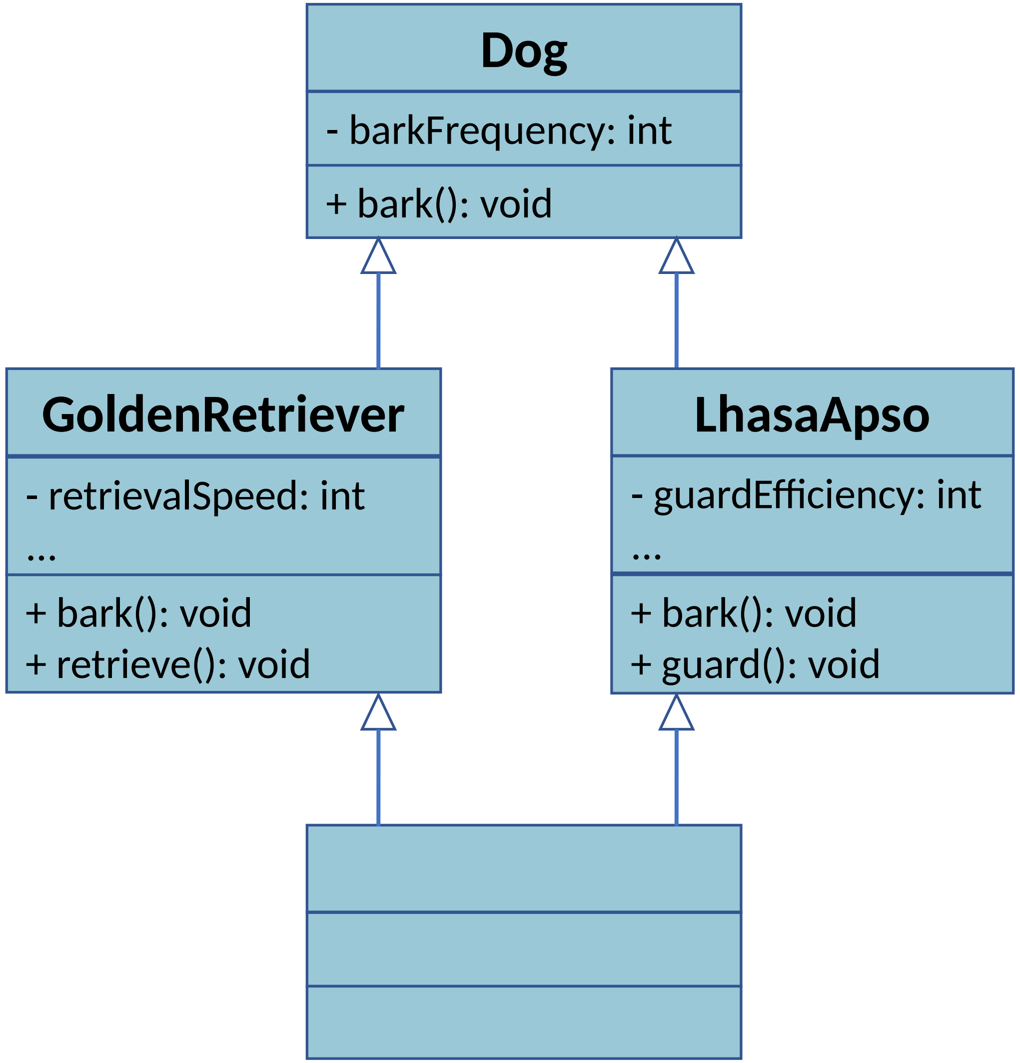
Multiple Inheritance

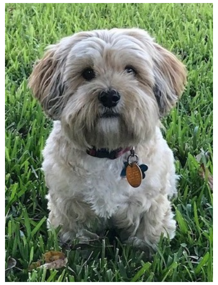




We use this *hierarchy* to create a dog that has both *retrieve* and *guard*

Multiple Inheritance





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

Multiple Inheritance

Dog

- barkFrequency: int

+ bark(): void

GoldenRetriever

- retrievalSpeed: int

…

+ bark(): void
+ **retrieve**(): void

LhasaApso

- guardEfficiency: int

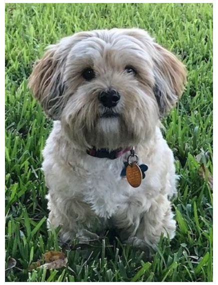
…

+ bark(): void
+ **guard**(): void

GoldenApso

- friendliness: int

+ **play**(): void



GoldenApso **inherits** *retrieve* from *GoldenRetriever*, *guard* from *LhasaApso*, and **defines** *play*

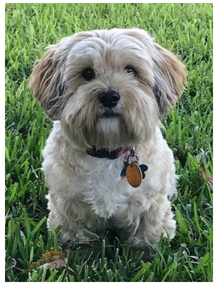
Multiple Inheritance

Dog
- barkFrequency: int
+ bark(): void

GoldenRetriever
- retrievalSpeed: int
...
+ **bark**(): void
+ retrieve(): void

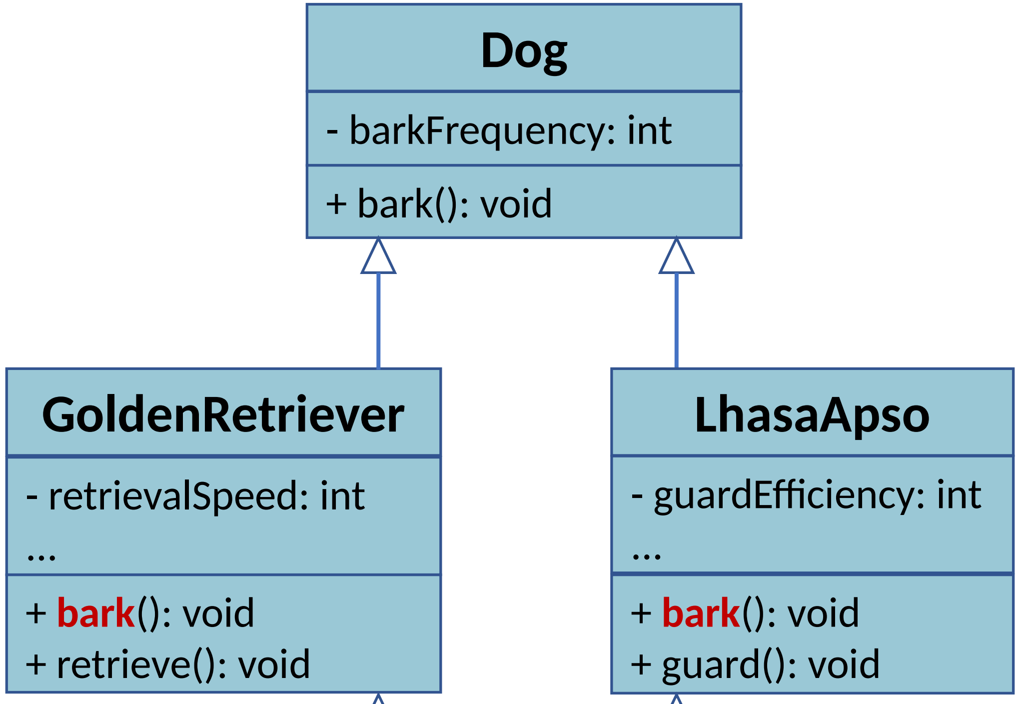
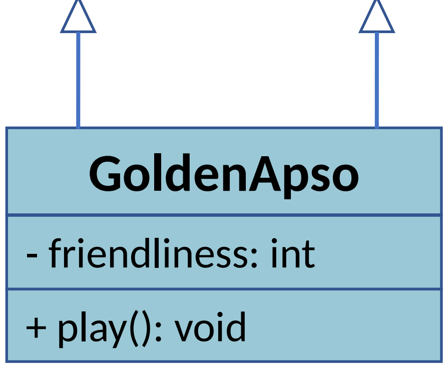
LhasaApso
- guardEfficiency: int
...
+ **bark**(): void
+ guard(): void

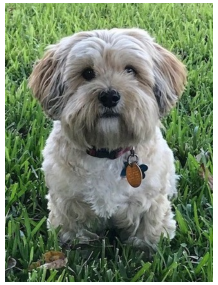
GoldenApso
- friendliness: int
+ play(): void



How does a *GoldenApso* *bark*?

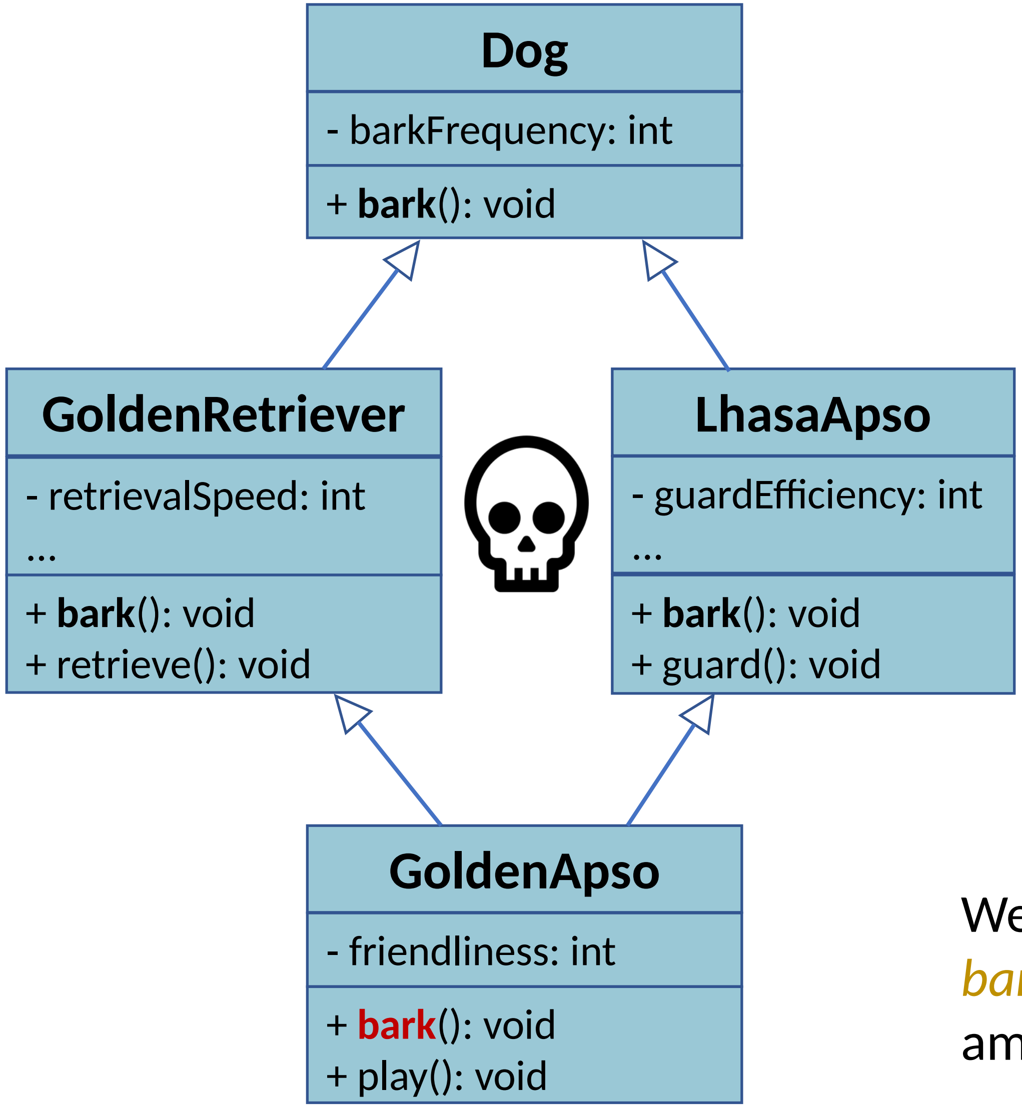
Multiple Inheritance



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

Multiple Inheritance



“Deadly Diamond of Death”

We do not know! *GoldenApso* must provide its *bark* overridden method to prevent errors and ambiguity: complexity

Multiple Inheritance

- An abstract class allows for defining a **design contract**

In theory:

- A subclass could inherit from **multiple** abstract superclasses, **each** describing a **design contract**

In practice:

- **Possible** in *C++* but **increases software complexity** and can lead to the “Deadly Diamond of Death”
- **Not possible** in *Java* or *C#*

How can we use multiple inheritance in Java?

Outline

- Recap: Design Contracts and Polymorphism with Abstract Classes
- Class Inheritance: Drawbacks
  - Weakened Encapsulation
  - Complexity of Multiple Inheritance
- Interface Inheritance
  - **Abstract classes versus Interfaces**
  - Comparison with Class Inheritance
  - Conclusions
- The Java Object Class
  - Object superclass methods
  - Constructors Invocation
  - Custom Exceptions with Inheritance

Question on interfaces

Which one of the answers about **interfaces** is **wrong**?

Answer on PollEveryWhere

https://pollev.com/francescotusa

Abstract Classes versus Interfaces

- We used the term "**interface**" to describe the set of public methods that an object exposes
- In many OOP languages, such as *Java* and *C#*, an interface is an alternative to abstract classes for defining **design contracts** and achieving **polymorphism**

Abstract Classes versus Interfaces

- An abstract class can have *attributes*, *methods* and abstract *methods* (like the *Shape* class in last week's tutorial)

- An interface **only has methods:** all these **methods** are *implicitly* abstract and public (< Java 8)
- This prevents the **ambiguity** of the **diamond problem**
- A class can fulfil **multiple design contracts** by “inheriting” from **multiple interfaces**

Defining an Interface

-able: interface as **capability**

interface **Nameable**
+ *getName(): String* + *setName(String n): void*

```
public interface Nameable
{
    // no attributes (only constants)

    // no constructors

    public void setName(String name);
    public String getName();
}
```

Nameable defines a behaviour associated with *nameable* entities

Defining an Interface

interface
Nameable

+ getName(): String
+ setName(String n): void

```
public interface Nameable
{
   // no attributes (only constants)

  // no constructors

   public void setName(String name);
   public String getName();
}
```

No state / attributes cannot be instantiated

Nameable defines a behaviour associated with *nameable* entities

Defining an Interface

interface **Nameable**
+ *getName(): String* + *setName(String n): void*

```
public interface Nameable
{
    // no attributes (only constants)

   // no constructors

    public void setName(String name);
    public String getName();
}
```

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

Nameable defines a behaviour associated with *nameable* entities

Outline

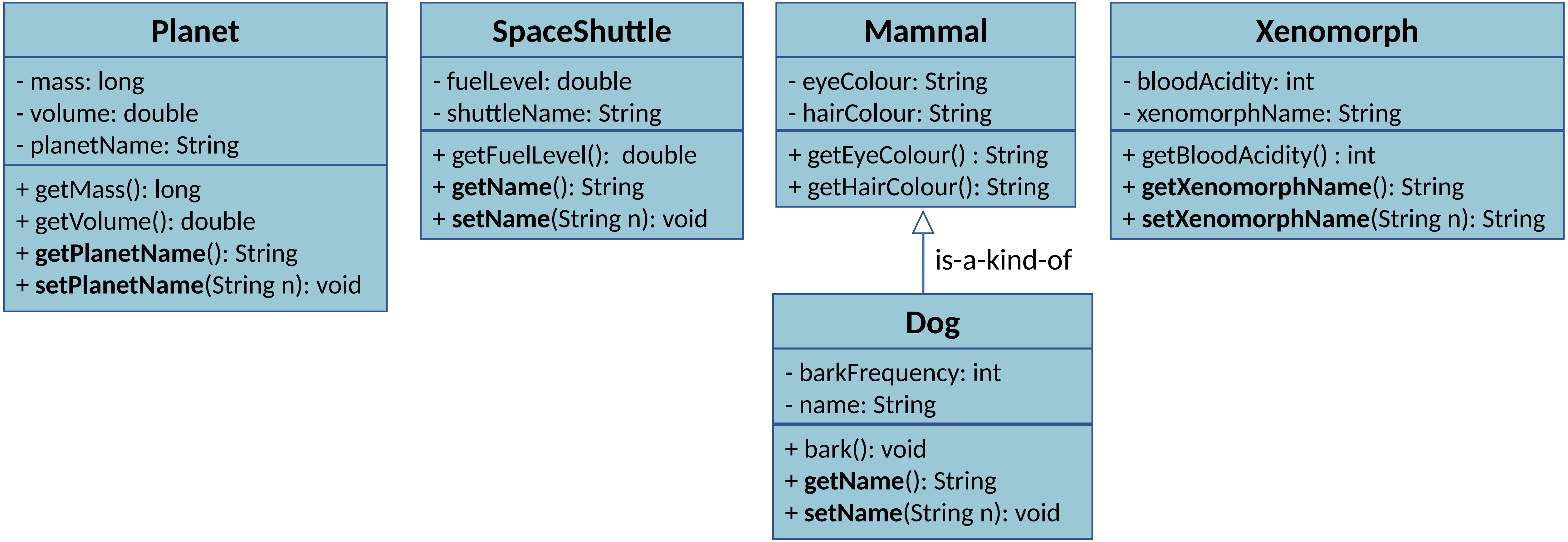
- Recap: Design Contracts and Polymorphism with Abstract Classes
- Class Inheritance: Drawbacks
  - Weakened Encapsulation
  - Complexity of Multiple Inheritance
- Interface Inheritance
  - Abstract classes versus Interfaces
  - **Comparison with Class Inheritance**
  - Conclusions
- The Java Object Class
  - Object superclass methods
  - Constructors Invocation
  - Custom Exceptions with Inheritance

Example: *a Space Exploration Game*

Embark on an interstellar adventure, journeying through diverse worlds where extraordinary creatures come to life!

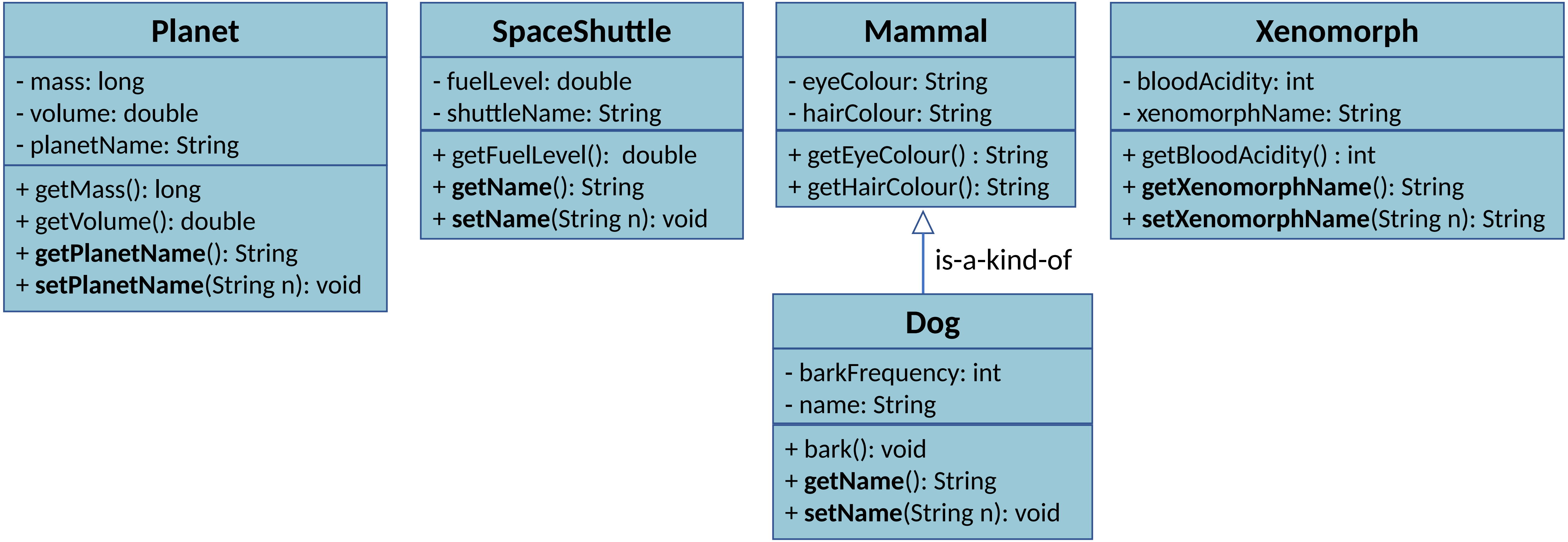
Class Inheritance versus *Interface* Inheritance

Most classes have behaviours associated with the **name**



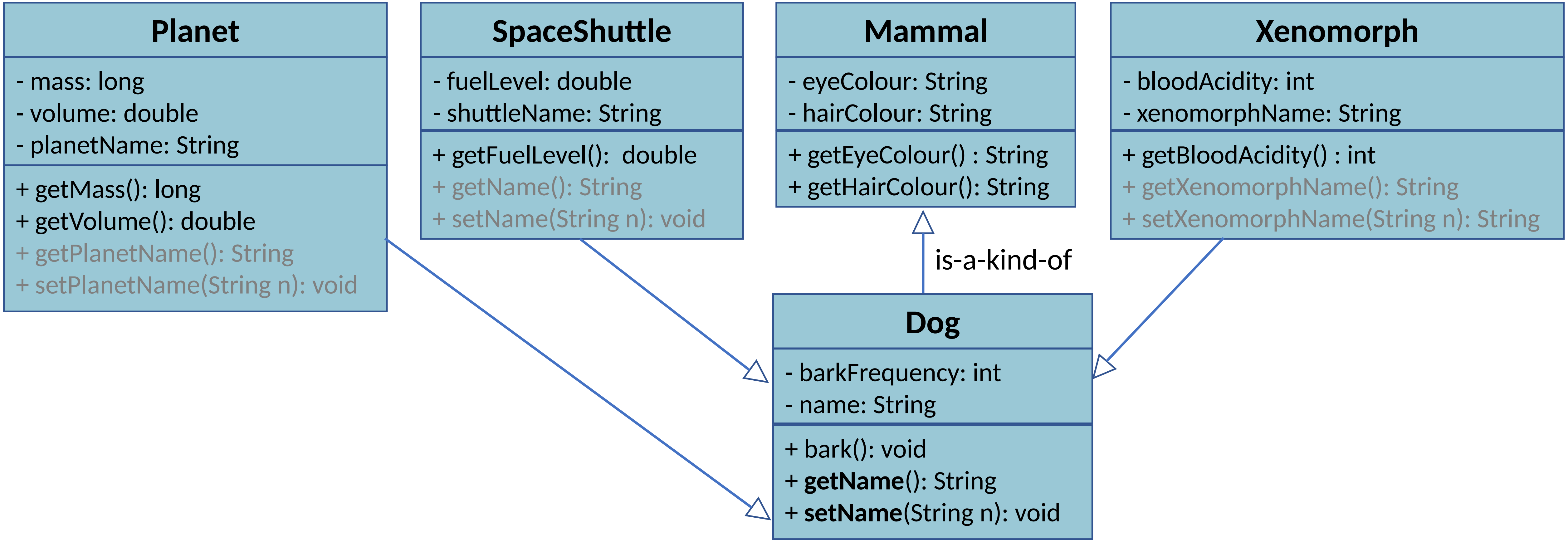
Class Inheritance versus *Interface* Inheritance

A well-thought-out design should **standardise** those name-related behaviours



Class Inheritance versus *Interface* Inheritance

These methods could be **inherited** from *Dog*—would this make sense?

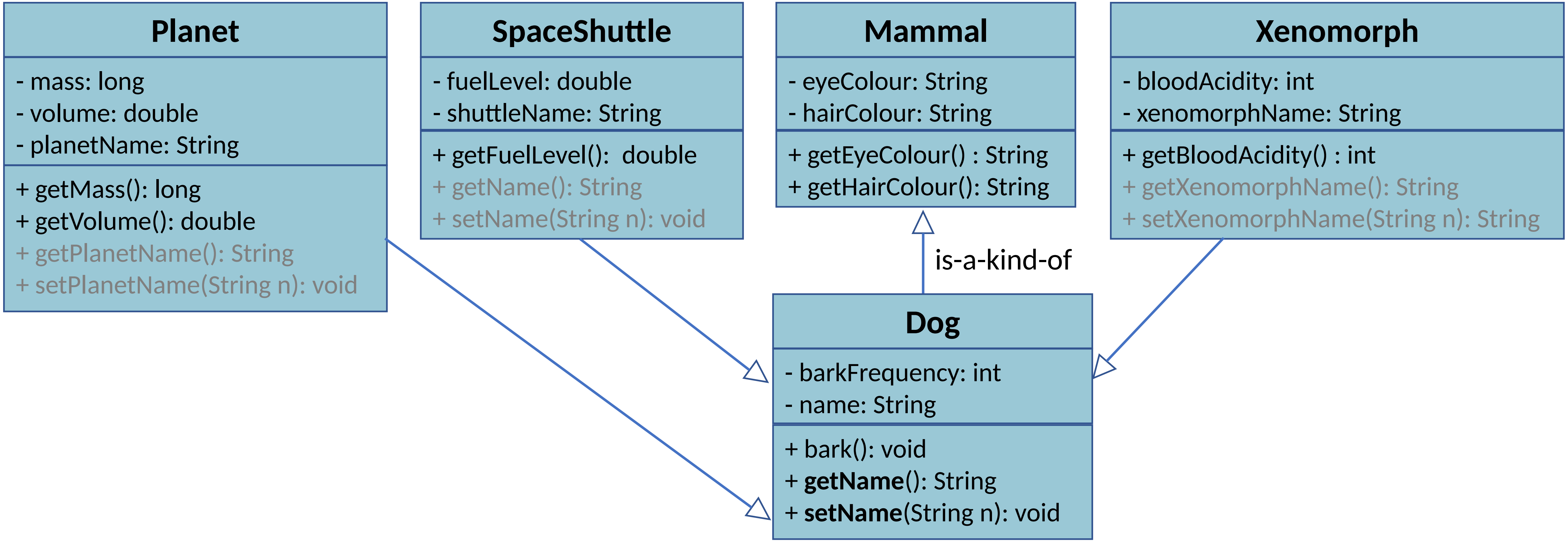


Class Inheritance: Drawbacks

- **Weakened** encapsulation: **changes** to a *superclass* can **ripple** on the *subclasses*
- **Complexity** in case of **multiple inheritance**
- Based on a **rigid** and not **flexible** inheritance hierarchy

Class Inheritance versus *Interface* Inheritance

No! *Planet*, *SpaceShuttle* and *Xenomorph* **are not a kind of** *Dog*

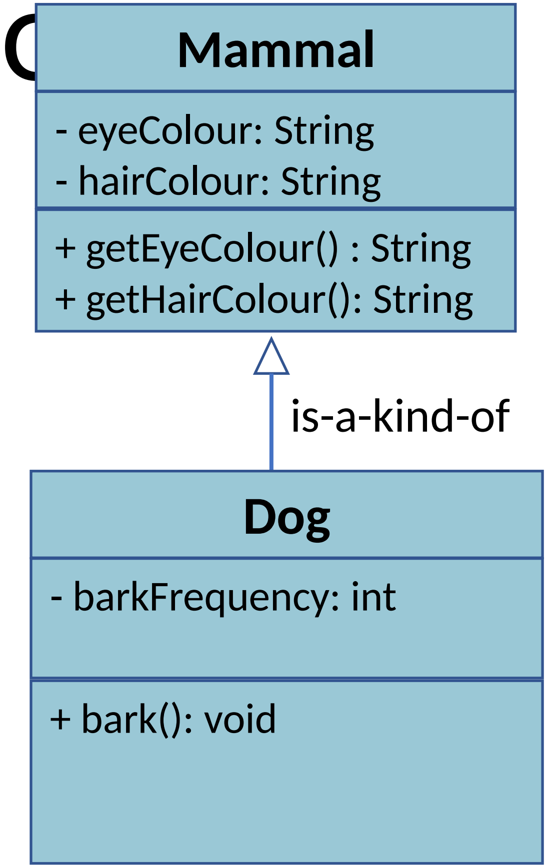


Example: a *Space Exploration Game*

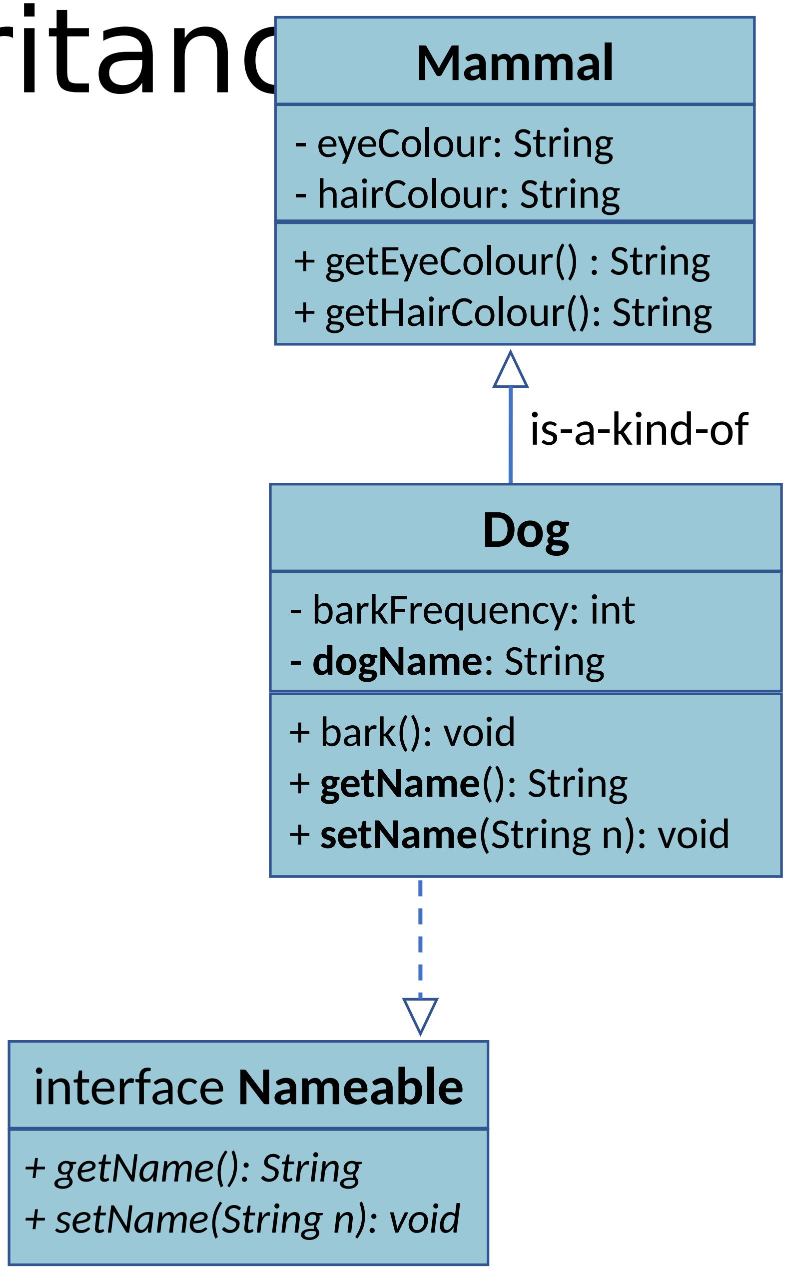
Embark on an interstellar adventure, journeying through diverse worlds where extraordinary creatures come to life!

Can **interfaces** be used in our game’s design?

Interface Inheritanc

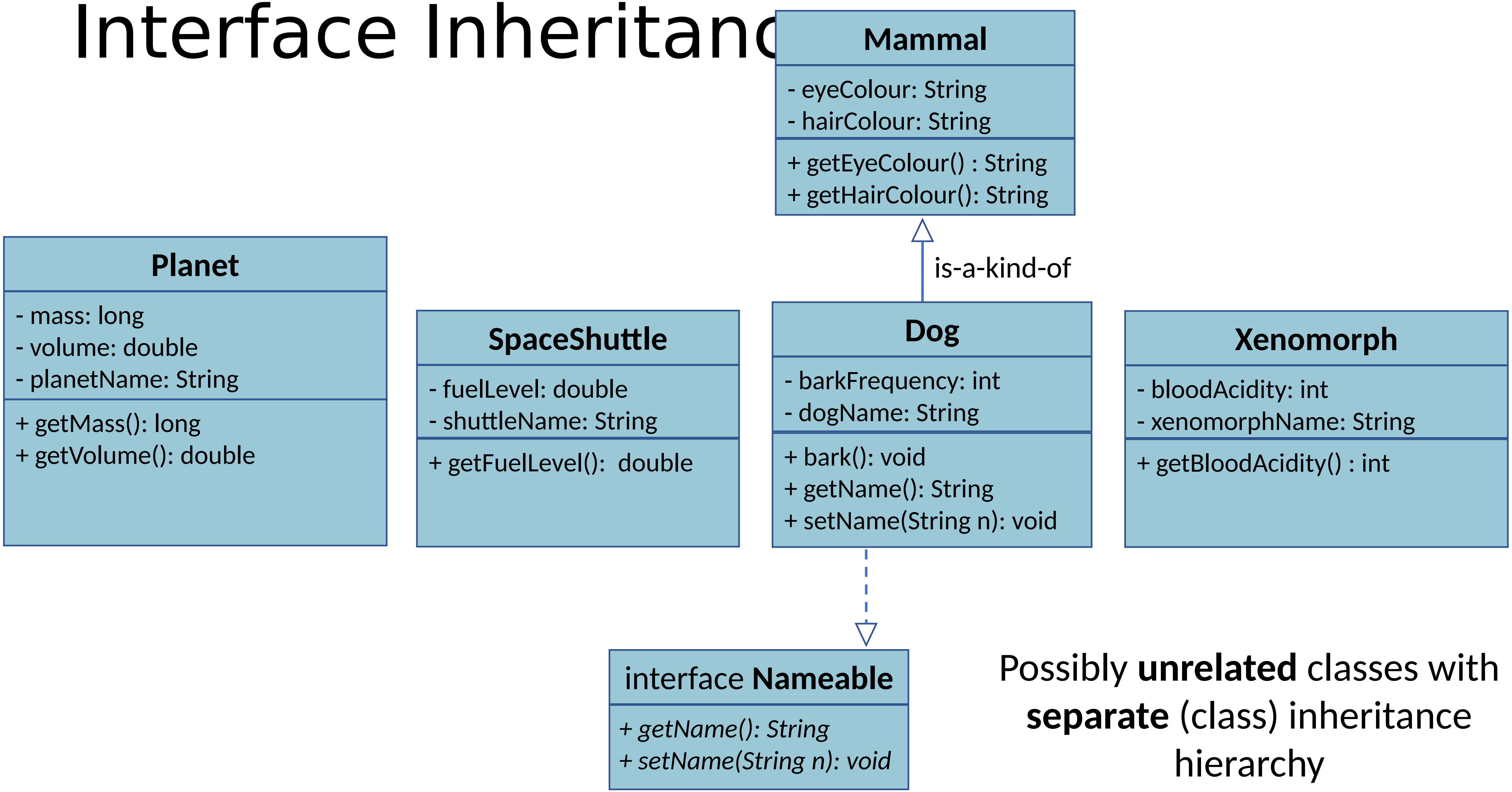


Interface Inheritance



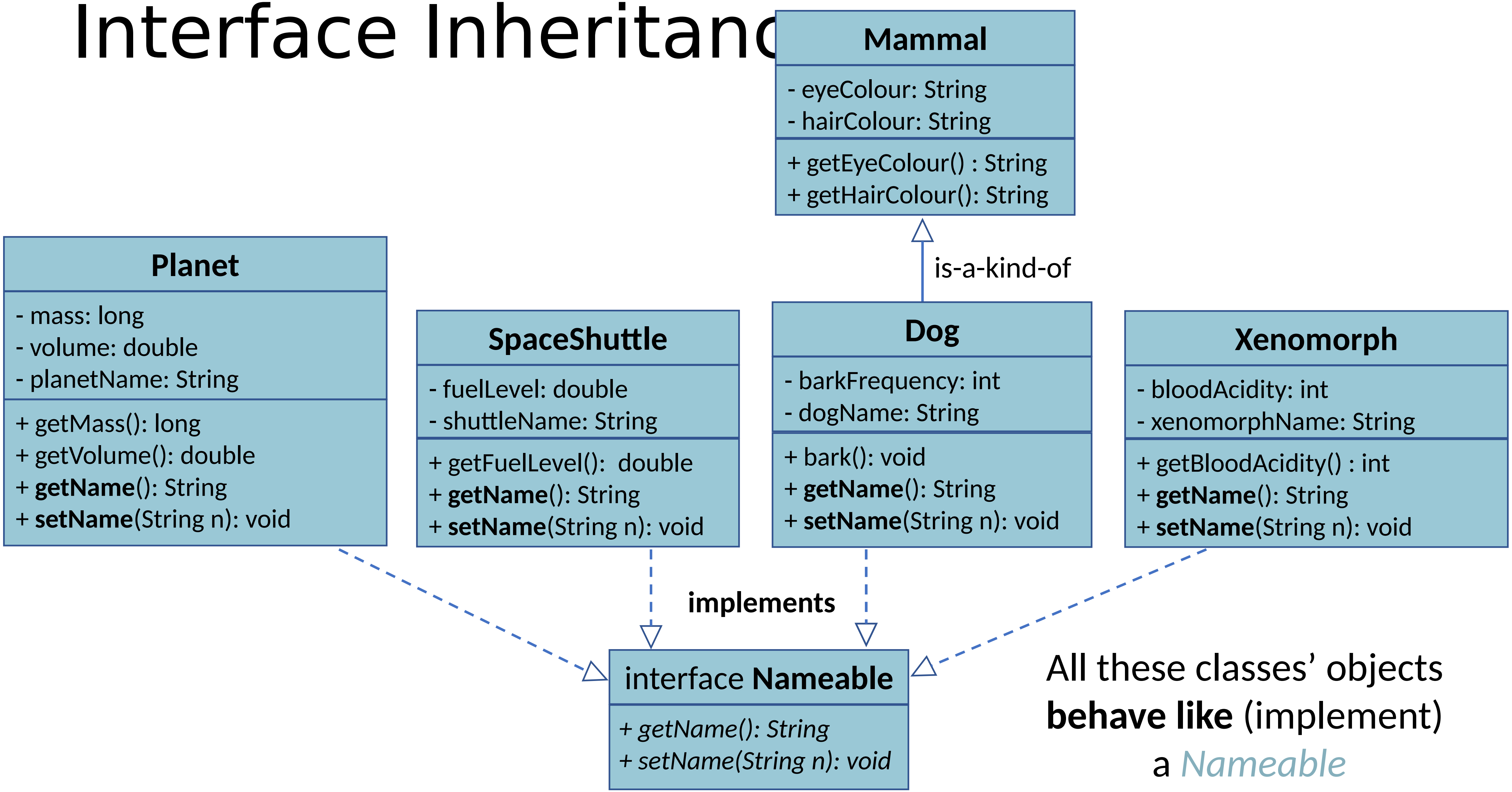
The *Nameable* **contract** requires the implementor to define the abstract methods *getName* and *setName* and (implicitly) an attribute for that behaviour

Interface Inheritanc



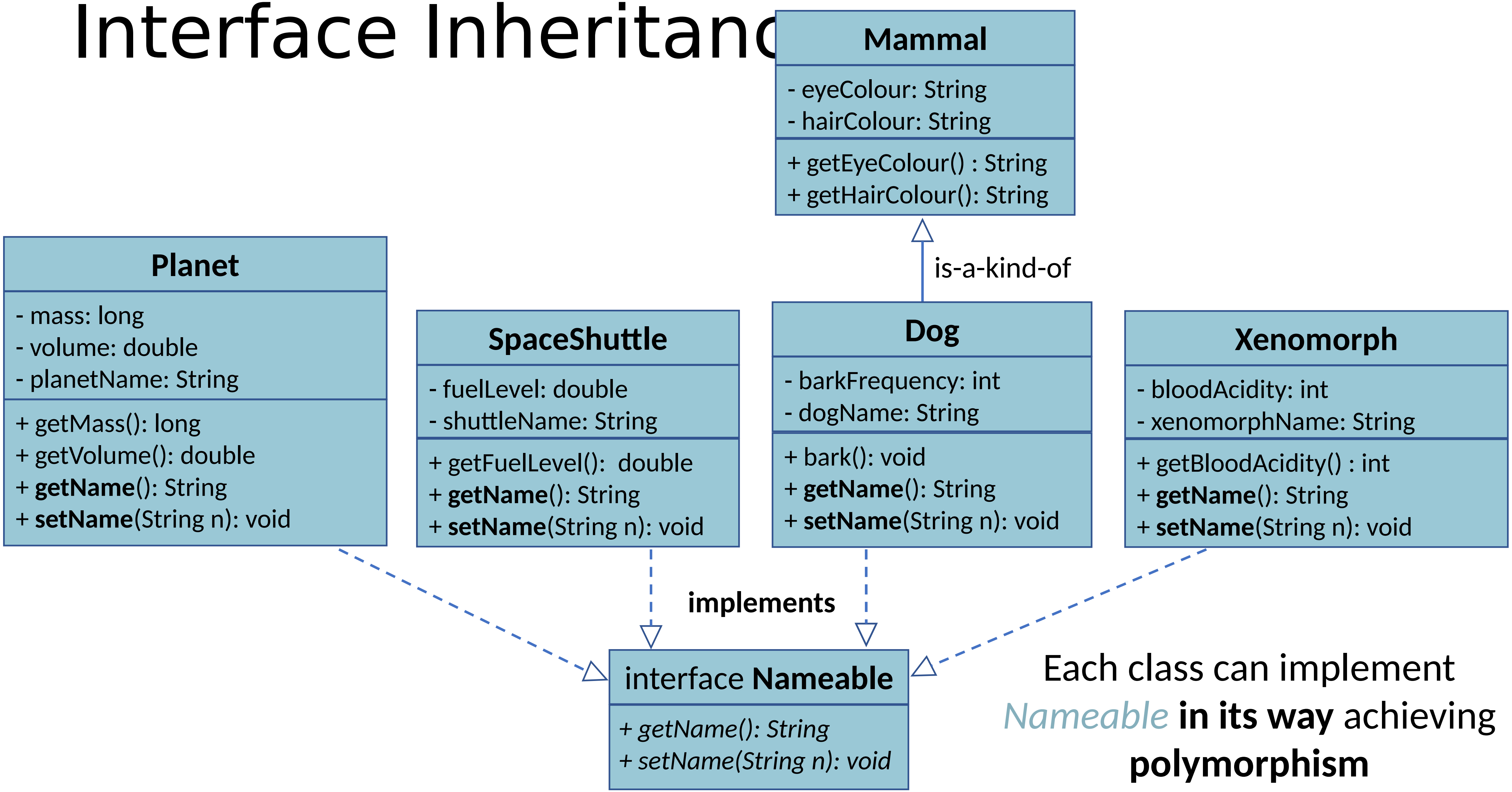
Possibly **unrelated** classes with **separate** (class) inheritance hierarchy

Interface Inheritanc



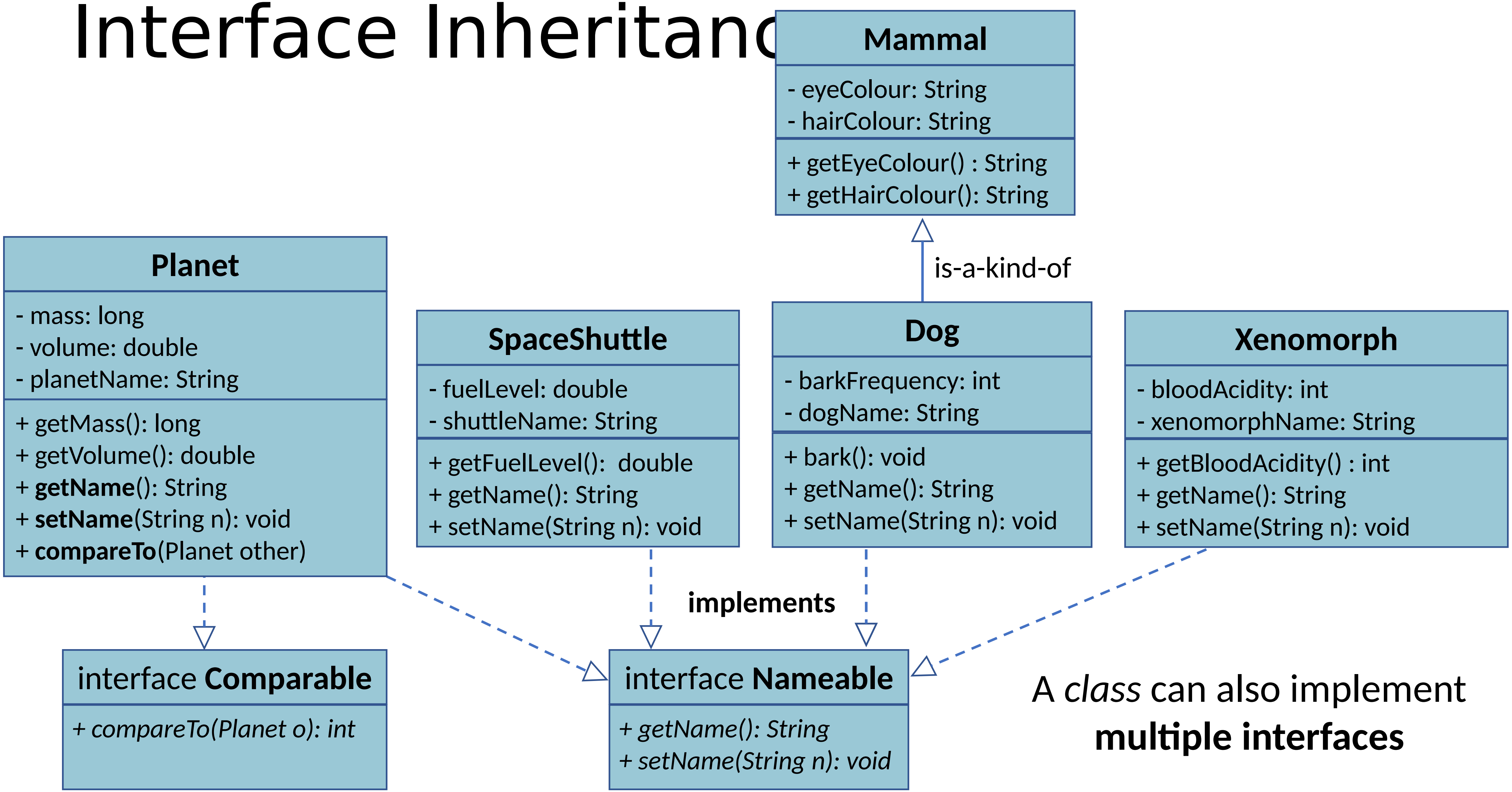
All these classes' objects
behave like (implement)
a *Nameable*

Interface Inheritanc



Each class can implement *Nameable* **in its way** achieving **polymorphism**

Interface Inheritance



A *class* can also implement **multiple interfaces**

Class Inheritance versus *Interface* Inheritance

Class inheritance:

- A **class** *inherits* from a (abstract) class and **all** its **parents**

Interface inheritance:

- No rigid and **formal inheritance structure**
- One or more interfaces could be added to **any class** where the design makes sense

Interface Inheritance

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

Polymorphism: code based on an interface's contract can **seamlessly interact** with **any object** that implements that interface, ensuring **flexibility** and **interoperability**.

Example: a *Space Exploration Game*

Embark on an interstellar adventure, journeying through diverse worlds where extraordinary creatures come to life!

Let’s write some **code**!

Interfaces: Code Example

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

    public Dog(String ec, String hc, int bf) { super(ec, hc); … }
    public void bark() { … }
    // methods inherited from Mammal


}
```

Interfaces: Code Example

```
public class Dog extends Mammal implements Nameable
{
        private int barkFrequency;
        private String dogName;

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

        public void bark() { … }

        // methods inherited from Mammal

        public String getName() { return dogName; }

        public void setName(String n) { dogName = n; }
}
```

Nameable contract implementation

A class can implement the interface in its **way**

Interfaces: Code Example

```
public class Planet
{
	private long mass;
	private double volume;

	public Planet(long m, double v) { … }

	public long getMass() { return mass }

	public double getVolume() { return volume }




}
```

Interfaces: Code Example

```
public class Planet implements Nameable, Comparable
{
	private long mass;
	private double volume;
	private String planetName;

	public Planet(long m, double v) { … }

	public long getMass() { return mass }

	public double getVolume() { return volume }

	public String getName() { return planetName; }

	public void setName(String n) { planetName = n; }

	public int compareTo(Planet other) {
		return other.mass - this.mass;
	}

}
```

Nameable contract implementation

Comparable contract implementation

Each class can implement the **same** interface in a **different way**

Interfaces: Code Example

```
public static class NameLogger
{
        public static void Log(Nameable nameable)
        {
            System.out.println(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();

        Planet planet1 = new Planet(1000000, 200000);

        planet1.setName("Earth");
        System.out.println(planet1.getMass());

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

Interfaces: Code Example

```
public static class NameLogger
{
    public static void Log(Nameable nameable)
    {
        System.out.println(nameable.getName());
        // now let's test the bark method
        nameable.bark();
        // also log to a file...
    }
}
```

```
public class Program
{
    public static main(String[] args)
    {
        Nameable dog1 = new Dog("brown", "white", 10);
        dog1.setName("Alan");
        dog1.bark();

        Planet planet1 = new Planet(1000000, 200000);

        planet1.setName("Earth");
        System.out.println(planet1.getMass());

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

Will this program version **work**?

Question

- What instruction(s) will generate a compiler error in the program?

Answer on PollEveryWhere

https://pollev.com/francescotusa

Answer

- 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*

Why is this important?

Outline

- Recap: Design Contracts and Polymorphism with Abstract Classes
- Class Inheritance: Drawbacks
  - Weakened Encapsulation
  - Complexity of Multiple Inheritance
- Interface Inheritance
  - Abstract classes versus Interfaces
  - Comparison with Class Inheritance
  - **Conclusions**
- The Java Object Class
  - Object superclass methods
  - Constructors Invocation
  - Custom Exceptions with Inheritance

Interfaces: Single Responsibility

```
public static class NameLogger
{
    public static void Log(Nameable nameable)
    {
        System.out.println(nameable.getName());
        // now let's test the bark method
        nameable.bark();
        // also log to a file...
    }
}
```

Fewer class dependencies and improved modularity and maintainability:

The code should have a **single responsibility** and not handle object functionalities that are not part of *Nameable*.

Interfaces: Decoupling

```
public static class NameLogger
{
    public static void Log(Nameable nameable)
    {
        System.out.println(nameable.getName());
        // now let's test the bark method
        nameable.bark();
        // also log to a file...
    }
}
```

Fewer class dependencies and improved modularity and maintainability:

The code works with **any object** that **implements** the *Nameable* interface and is **decoupled** from the details of an object's concrete **class**.

Interfaces: Flexibility and Maintainability

```
public static class NameLogger
{
    public static void Log(Nameable nameable)
    {
        System.out.println(nameable.getName());
        // now let's test the bark method
        nameable.bark();
        // also log to a file...
    }
}
```

Fewer class dependencies and improved modularity and maintainability:

The code is **polymorphic** and can **easily accommodate** new classes that implement *Nameable* **without** any **changes**:

Robot implements *Nameable*

Interfaces: Summary and Conclusions

- A class can **inherit** from **only one** *superclass*, e.g., *Mammal*
- It can **implement many** *interfaces*, e.g., *Nameable*, *Comparable*, etc. (no "*Diamon Problem*")
- Unrelated classes, e.g., *Dog*, *Planet*, etc., can implement the **same** interface (**design contract**) in **different polymorphic ways**

- Interfaces allow for **loose coupling** and **separation of concerns** via adherence to the **Single Responsibility Principle**
- More **modular**, **maintainable**, **flexible** and **extensible** codebase

Outline

- Recap: Design Contracts and Polymorphism with Abstract Classes
- Class Inheritance: Drawbacks
  - Weakened Encapsulation
  - Complexity of Multiple Inheritance
- Interface Inheritance
  - Abstract classes versus Interfaces
  - Comparison with Class Inheritance
  - Conclusions
- The Java Object Class
  - **Object superclass methods**
  - Constructors Invocation
  - Custom Exceptions with Inheritance

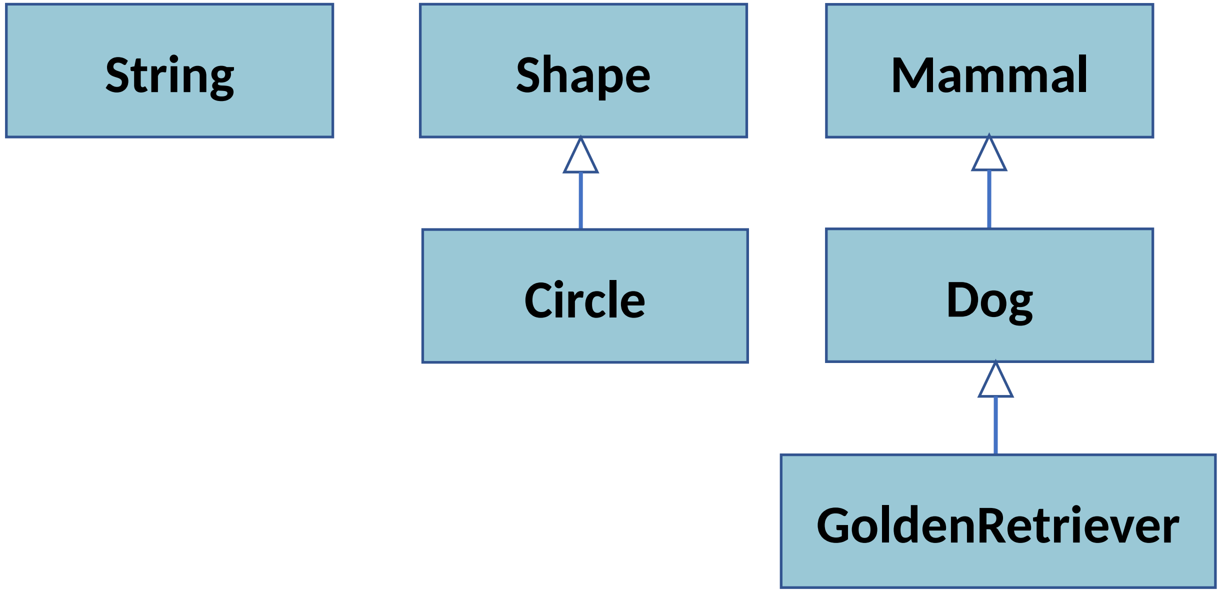
Reflection

- **Composition** and **interface inheritance** should be **used instead** of **class inheritance**
- *Why did we study class inheritance?*

Hand-on task

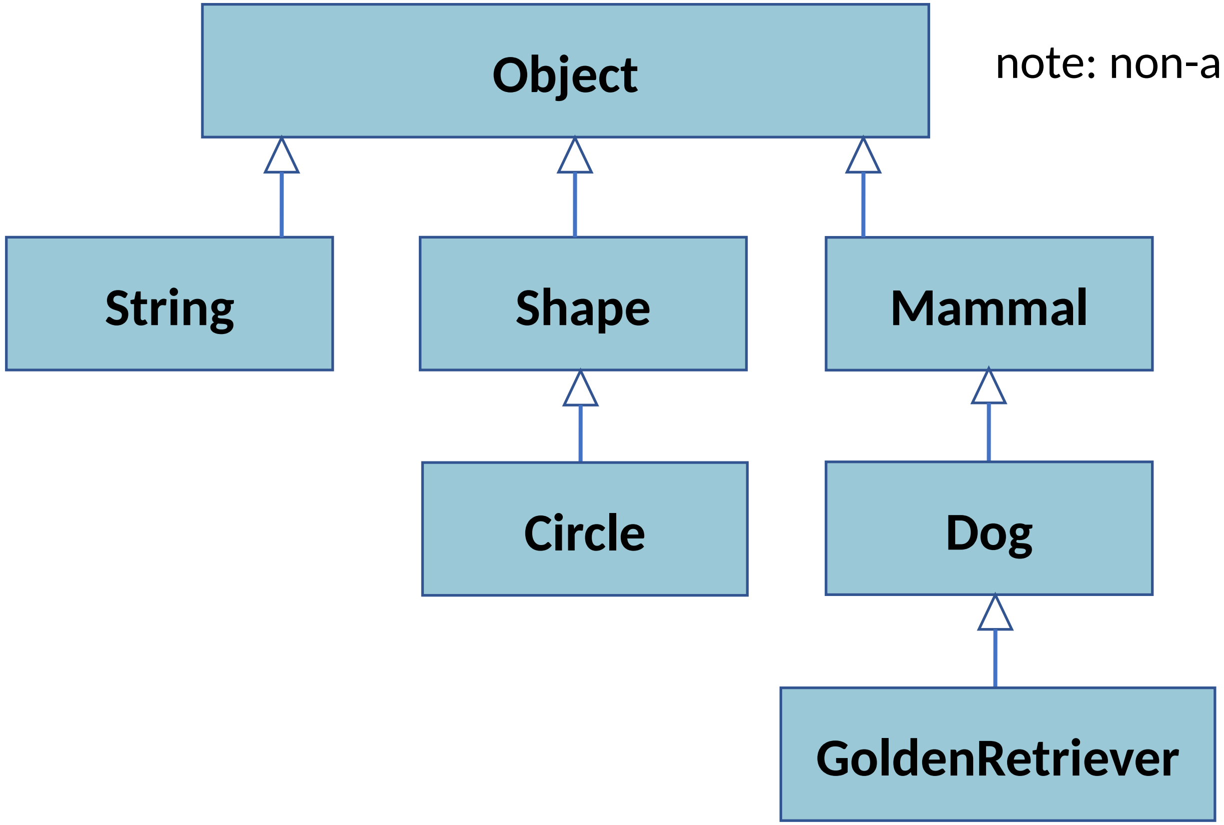
- Learning through the code available on Blackboard (Week11)

Java Inheritance Tree: *Object* class



Java Inheritance Tree: *Object* class

All Java classes you will use or write extend the *java.lang*.*Object* class



note: non-abstract class

Java Inheritance Tree: *Object* class

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

```
Shape circle = new Circle(p1, 4.78);
```

- *Liskov Substitution Principle* (LSP): a circle *is a* shape.

Java Inheritance Tree: *Object* class

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

Shape circle = new Circle(p1, 4.78);

- *Liskov Substitution Principle* (LSP): a circle *is a* shape.

- Object can act as a sort of *universal container*: a variable of this type can **hold a reference to almost anything**:

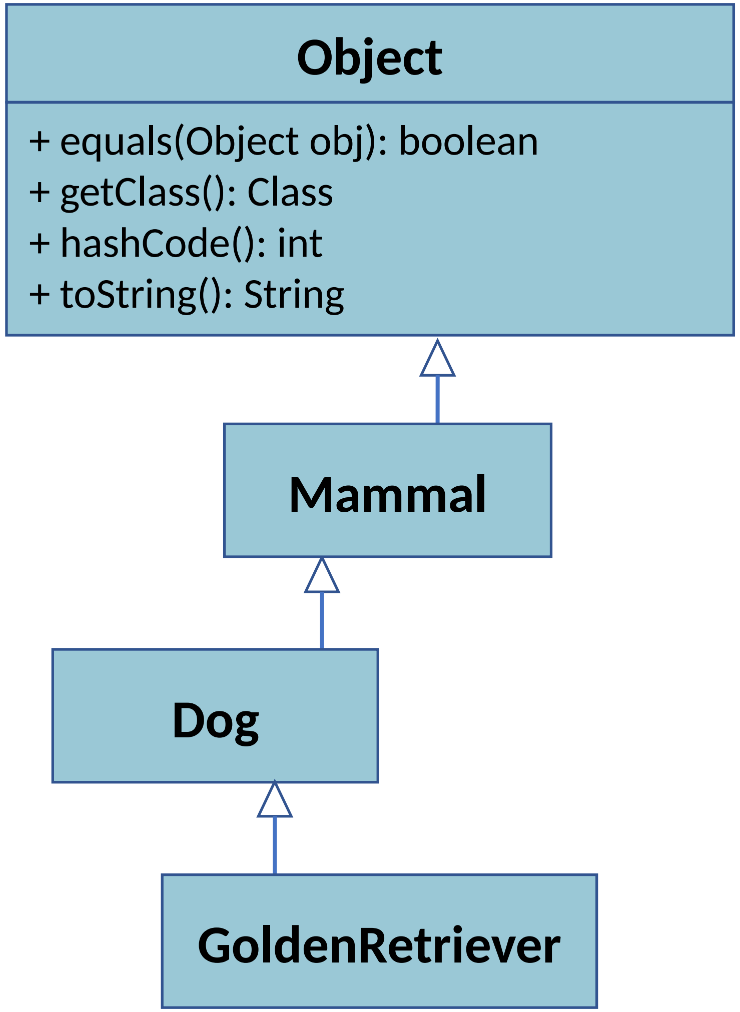
Object obj = new Circle(p1, 4.78)

- *Liskov Substitution Principle* (LSP): a circle *is an* object.

Java Inheritance Tree: *Object* class

All the classes we write will **inherit** the public (and protected) methods defined in the Object class

Subclasses **inherit** the *default implementations* of these methods



some of the *Object* class methods

Java Inheritance Tree: *Object* class

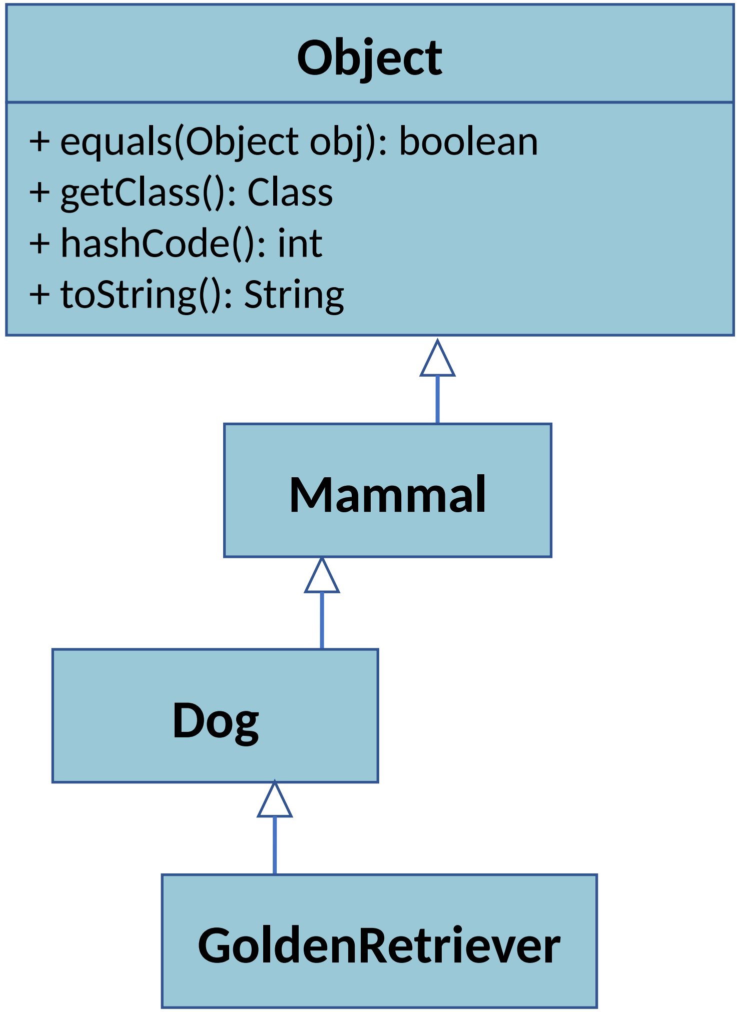
- Some of the methods of the *Object* class
  - *equals(Object obj)*: returns true if this object is **equal** to *obj*
  - *getClass()*: returns the *Class* (type) of the object
  - *hashCode()*: returns an int (hash) of the object
  - *toString()*: returns a String representing the object

Object class: toString()

- returns a readable textual representation of the object
- It is called automatically when an object is provided as the argument of *System.out.println(…)*

```
Circle circle1 = new Circle( ... );
System.out.println(circle1); // automatically calls circle1.toString()
```

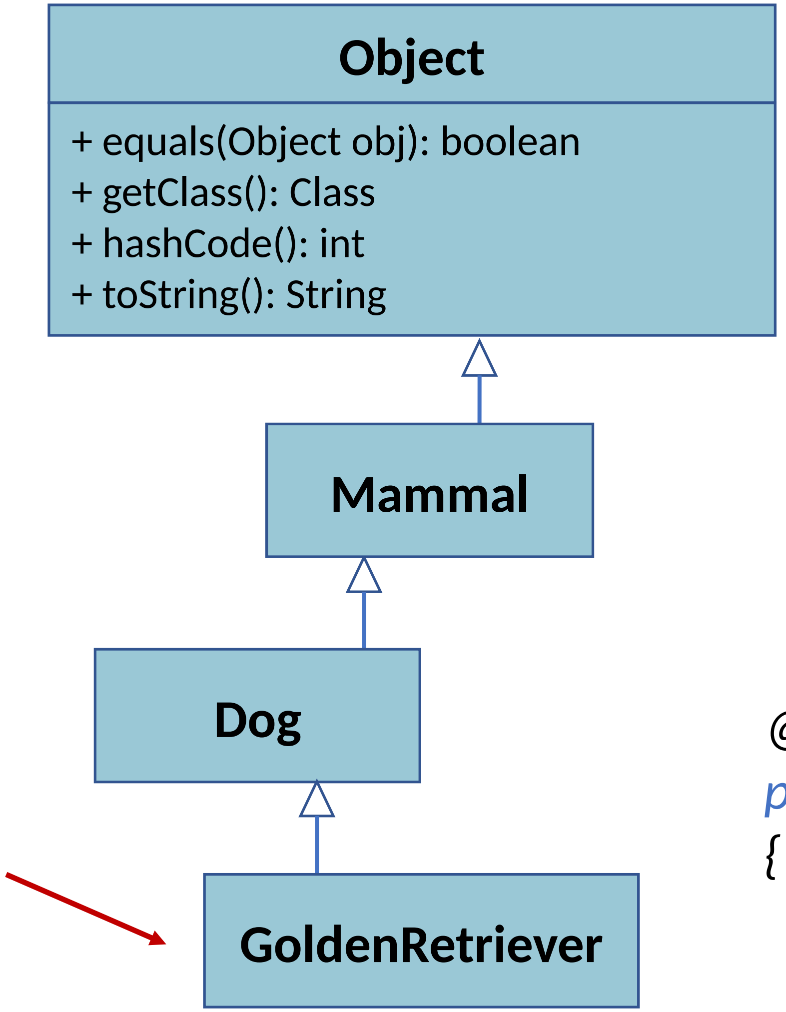
Object class: toString()




```
public String toString()
{
   return getClass().getName() + "@" +

         Integer.toHexString(hashCode());
}
```

The default implementation of toString returns the fully qualified name of the object's type: *packageName.className* and the associated *hashcode*

Object class: toString()



toString is usually **overridden** by subclasses to represent the object status as a *String*

```
@Override
public String toString()
{
  // return a custom String that
  // represents a GoldenRetriever object
}
```

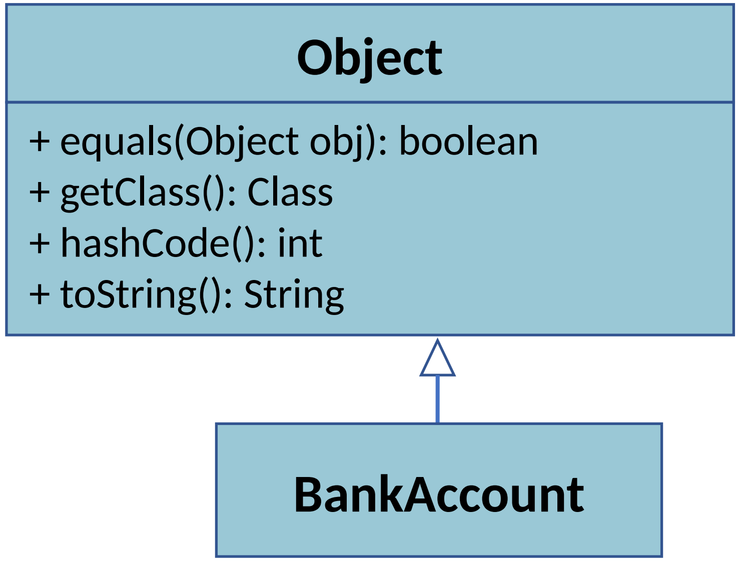
Object class: equals() versus == operator

```
class Program
{
    public static void main(String[] args)
    {
        int a = 10;
        int b = 10;
        if (a == b)
            System.out.println("a is equal to b");

        BankAccount account1 = new BankAccount("AB456", 200.0);
        BankAccount account2 = new BankAccount("AB456", 200.0);
        if (account1 == account2)
            System.out.println("account1 is equal to account2");
    }
}
```

Object class: equals() versus == operator

```
class Program
{
    public static void main(String[] args)
    {
        int a = 10;
        int b = 10;
        if (a == b)
            System.out.println("a is equal to b");

        BankAccount account1 = new BankAccount("AB456", 200.0);
        BankAccount account2 = new BankAccount("AB456", 200.0);
        if (account1.equals(account2))
            System.out.println("account1 is equal to account2");
    }
}
```

Object class: equals() versus == operator

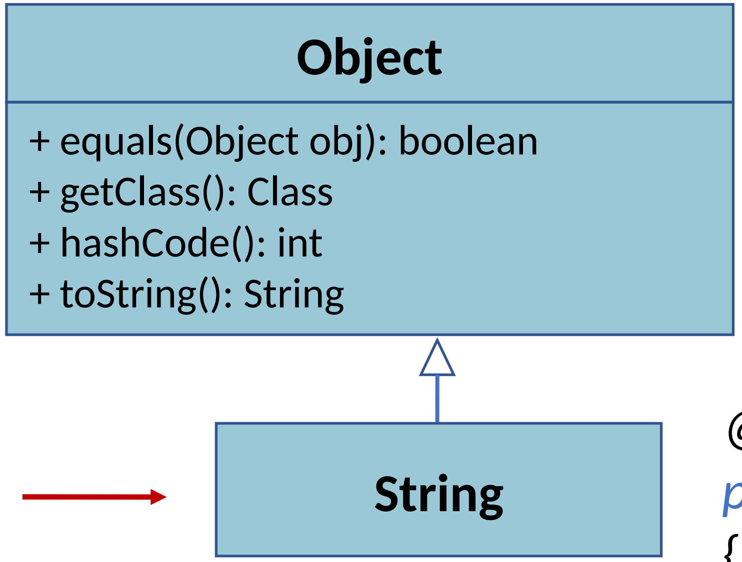



```
public boolean equals(Object obj)
{
   return this==o;
}
```

The default implementation of equals simply checks the object references (as the *if* in the previous code)

Object class: equals() versus == operator

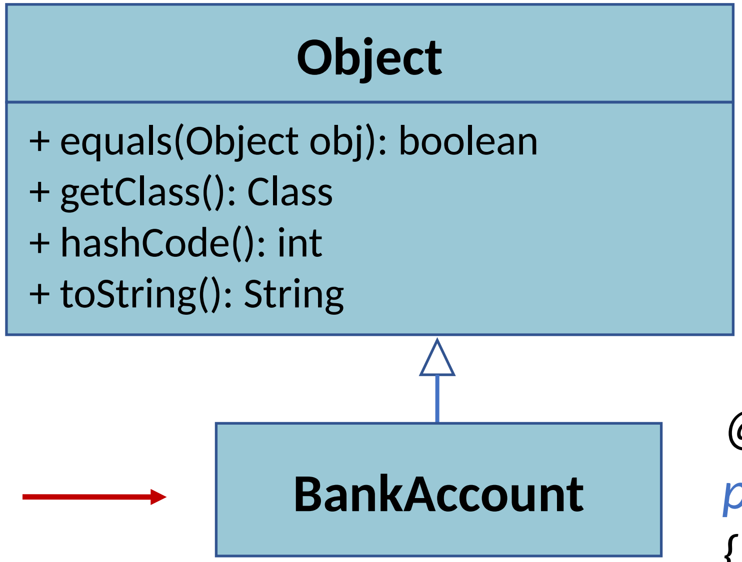
Built-in object types such as *String override* equals according to their equality criteria




```
@Override
public boolean equals(Object obj)
{
  // implements the criteria to check
  // string equality
}
```

Object class: equals() versus == operator

How can we check equality for a **type of object we define**?




```
@Override
public boolean equals(Object obj)
{
  // We need to define our criteria
  // i.e., when two BankAccount objects
  // are the same
}
```

Object class: equals() versus == operator

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

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

    public void deposit(double amount) { … }
    public double getBalance() { … }
    …

    @Override
    public boolean equals(Object obj) {


        // we can return true if this.number is equal to other.number
        // and this.balance is equal to other.balance



    }
}
```

Object class: equals() versus == operator

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

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

    public void deposit(double amount) { … }
    public double getBalance() { … }
    …

    @Override
    public boolean equals(Object obj) {
        if (! (obj instanceof BankAccount)) // check we received a BankAccount
            return false;
        BankAccount other = (BankAccount) obj; // convert the object back to a BankAccount
        return this.number.equals(other.number) && // we use equals already available from the String class
                  Double.compare(this.balance, other.balance)==0;
        // cannot use == with double due to approximation errors;
        // Double.compare returns 0 if the two double values are equals
    }
}
```

Outline

- Recap: Design Contracts and Polymorphism with Abstract Classes
- Class Inheritance: Drawbacks
  - Weakened Encapsulation
  - Complexity of Multiple Inheritance
- Interface Inheritance
  - Abstract classes versus Interfaces
  - Comparison with Class Inheritance
  - Conclusions
- The Java Object Class
  - Object superclass methods
  - **Constructors Invocation**
  - Custom Exceptions with Inheritance

Object class: Default Constructors

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

    public Mammal() {
        eyeColour = "green";
        hairColour = "white";
    }

    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 { // simplified: no Nameable
    private 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
}
```

We can invoke a superclass constructor using super(…) with a specific number of arguments.

Object class: Default Constructors

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

    public Mammal() {
        eyeColour = "green";
        hairColour = "white";
    }

    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 {
    private 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 superclass constructor using super(...) with a specific number of arguments.
What happens if we do not do it explicitly?

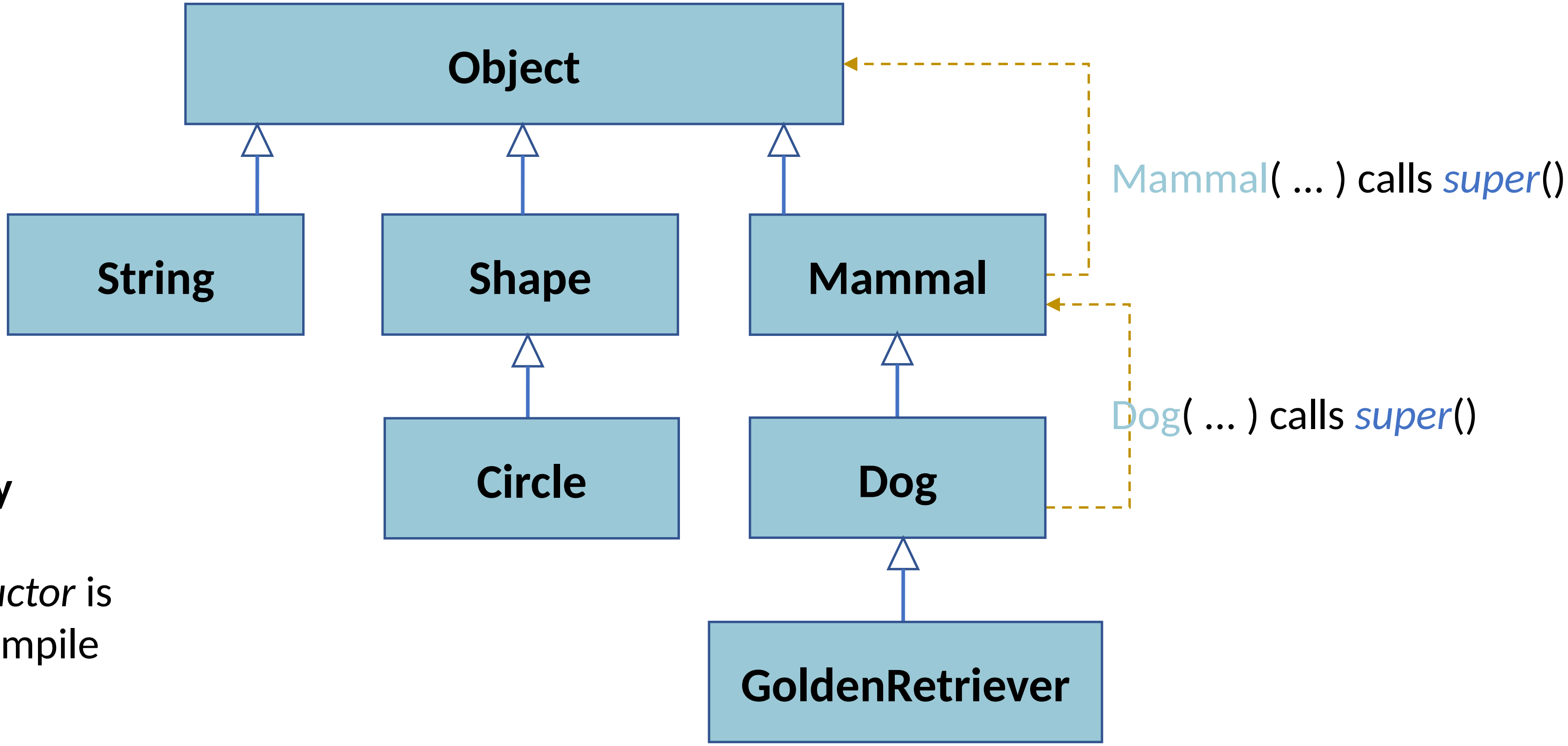
Object class: Default Constructors

```java
public class Mammal extends Object {
    private String eyeColour;
    private String hairColour;

    public Mammal() {
        super(); // from Object
        eyeColour = "green";
        hairColour = "white";
    }

    public Mammal(String ec, String hc) {
        super(); // from Object
        eyeColour = ec;
        hairColour = hc;
    }

    public String getEyeColour() {
        return eyeColour;
    }

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

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

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

    public void bark() {
        // uses barkFrequency
    }

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

The compiler adds an implicit call to super()—the *zero-args* superclass constructor

Object class: Default Constructors



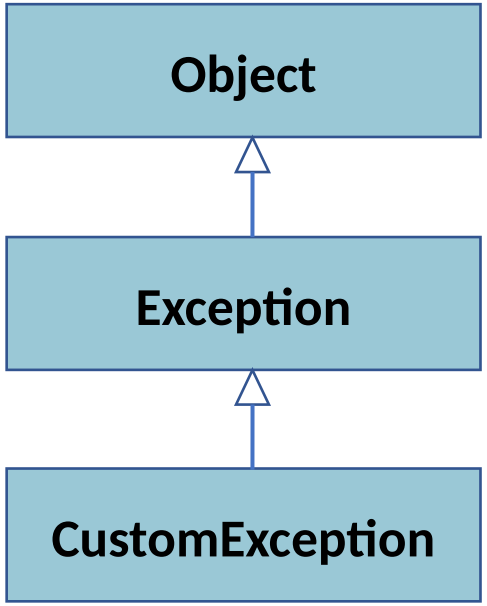
If a class does not explicitly define <u>any</u> constructors:
An *empty zero-args constructor* is automatically created at compile time

Outline

- Recap: Design Contracts and Polymorphism with Abstract Classes
- Class Inheritance: Drawbacks
  - Weakened Encapsulation
  - Complexity of Multiple Inheritance
- Interface Inheritance
  - Abstract classes versus Interfaces
  - Comparison with Class Inheritance
  - Conclusions
- The Java Object Class
  - Object superclass methods
  - Constructors Invocation
  - **Custom Exceptions with Inheritance**

Defining Custom Exceptions with Inheritance

- The *Exception* class also extends the *Object* class
- *Generalisation* and *Inheritance* can be applied to define **specialised kinds of** *Exception* objects
- Let's define a custom exception for our *previous space exploration game!*



Defining Custom Exceptions with Inheritance

```
public class SpaceShuttle implements Nameable {
    private String name;
    private double fuelLevel;

    public SpaceShuttle(double fuel) {
        fuelLevel = fuel;
    }

    // methods from Nameable ...

    // class specific methods
    public double getFuelLevel() { ... }
    public void setFuelLevel(double fuel) { ... }

    public void launch() throws InsufficientFuelException {
        double minFuelLevel = 50;
        if (fuelLevel < minFuelLevel) {
            throw new InsufficientFuelException("Fuel level too low for launch!", minFuelLevel);
        }

       System.out.println("Launching the space shuttle!");
    }
}
```

Defining Custom Exceptions with Inheritance

```
public class InsufficientFuelException extends Exception {
    private double requiredFuelLevel;

    public InsufficientFuelException(String message, double requiredFuelLevel) {
        super(message); // initialises the message attribute defined in the superclass
        this.requiredFuelLevel = requiredFuelLevel;
    }

    public double getRequiredFuelLevel() {
        return requiredFuelLevel;
    }
}
```

The **constructor** calls the *Exception* **superclass constructor** to initialise the *message* attribute and then initialises the class *requiredFuelLevel* attribute, which will be accessible via the corresponding *setter*.

Defining Custom Exceptions with Inheritance

```
public class ExplorationGame {
    public static void main(String[] args) {
        double fuel = 20.0;
        SpaceShuttle shuttle = new SpaceShuttle(fuel);

        try {
            shuttle.launch();
        } catch (InsufficientFuelException e) {
            System.out.println("Launch failed: " + e.getMessage());
            System.out.println("Current fuel level: " + fuel);
            System.out.println("Required fuel level: " + e.getRequiredFuelLevel());
        }
    }
}
```

The compiler “forces” us to catch *InsufficientFuelException* because it is a subclass of *Exception*—*a* ***checked exception***.

Catching exception objects that inherit from *RuntimeException* is [not enforced] by the compiler.

Questions