Interface

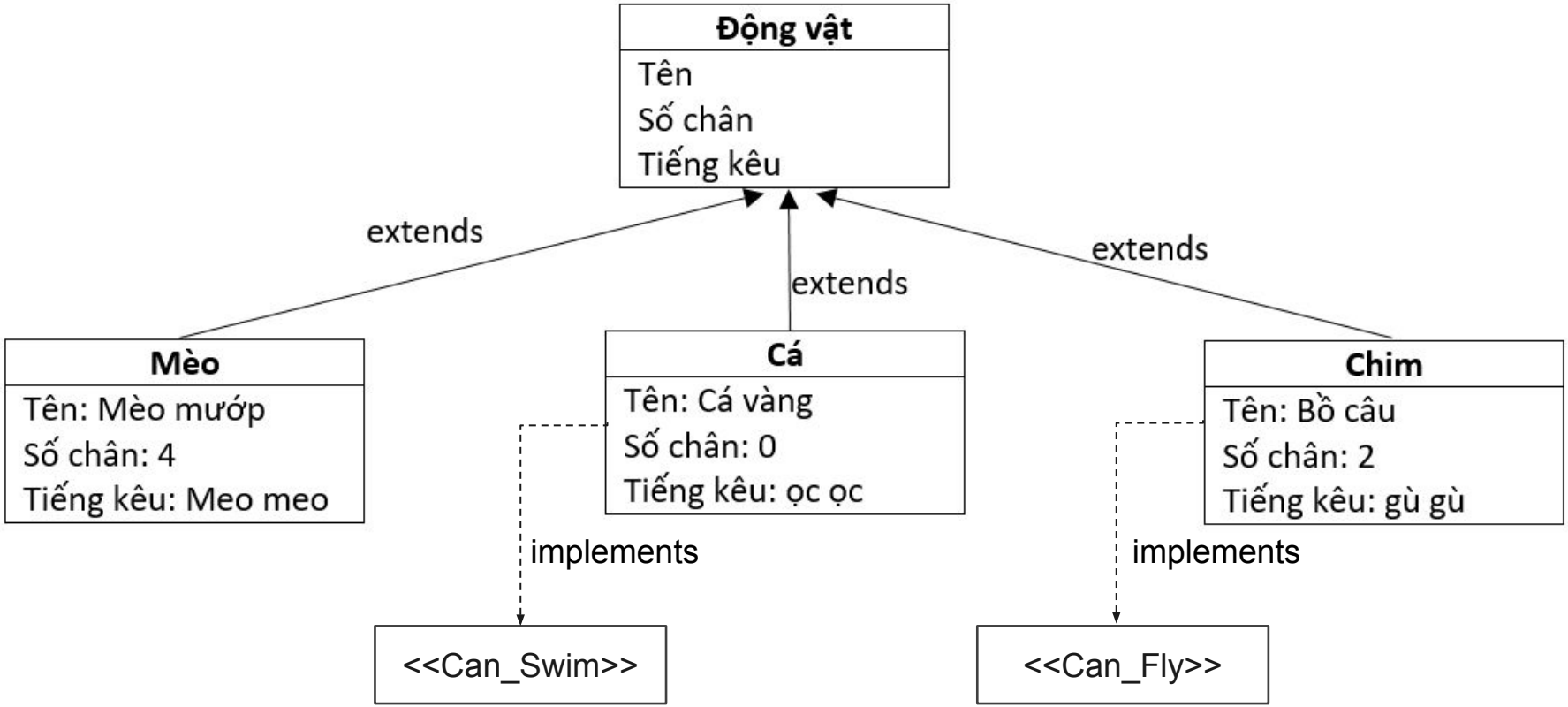
Interface

Interface là một kiểu dữ liệu tham chiếu trong Java. Nó là tập hợp các phương thức **abstract** (trừu tượng). Khi một lớp triển khai **interface**, thì nó sẽ hiện thực hóa những mẫu hàm của **interface** đó

Interface giống như một hợp đồng, nó không thể tự thực hiện các công việc được liệt kê trong hợp đồng, nhưng khi một class ký vào hợp đồng đó (**implements Interface**), thì class đó phải **tuân thủ** theo hợp đồng

Tại sao chúng ta lại sử dụng interface?

- Interface định nghĩa các tập hành vi mà class tuân thủ hơn là kế thừa
- Java không hỗ trợ đa kế thừa. Do đó, ta không thể kế thừa cùng một lúc nhiều class. Để giải quyết vấn đề này interface ra đời


Động vật
Tên
Số chân
Tiếng kêu
extends
extends
extends
Mèo
Tên: Mèo mướp
Số chân: 4
Tiếng kêu: Meo meo
Cá
Tên: Cá vàng
Số chân: 0
Tiếng kêu: ọc ọc
Chim
Tên: Bồ câu
Số chân: 2
Tiếng kêu: gù gù
implements
implements
<<Can_Swim>>
<<Can_Fly>>

Interface

01

Không thể khởi tạo, nên không có phương thức khởi tạo.

02

Tất cả các phương thức trong interface luôn ở dạng public abstract mà không cần khai báo.

03

Các thuộc tính trong interface luôn ở dạng public static final mà không cần khai báo, yêu cầu phải có giá trị

Tạo Interface

Cú pháp:

```
interface <Tên Interface>{
    //Các thành phần bên trong interface
}
```

```
public interface Flying {
    void fly();

    void flapWings();
}
```

Để truy cập các phương thức interface, interface phải được thực hiện bởi một lớp khác bằng từ khóa “implements”

```
public class Bird implements Flying{
    private String name;

    @Override
    public void fly() {
        System.out.println(name + " is flying!");
    }

    @Override
    public void flapWings() {
        System.out.println(name + " is flapping its wings!");
    }
}
```

Đa kế thừa

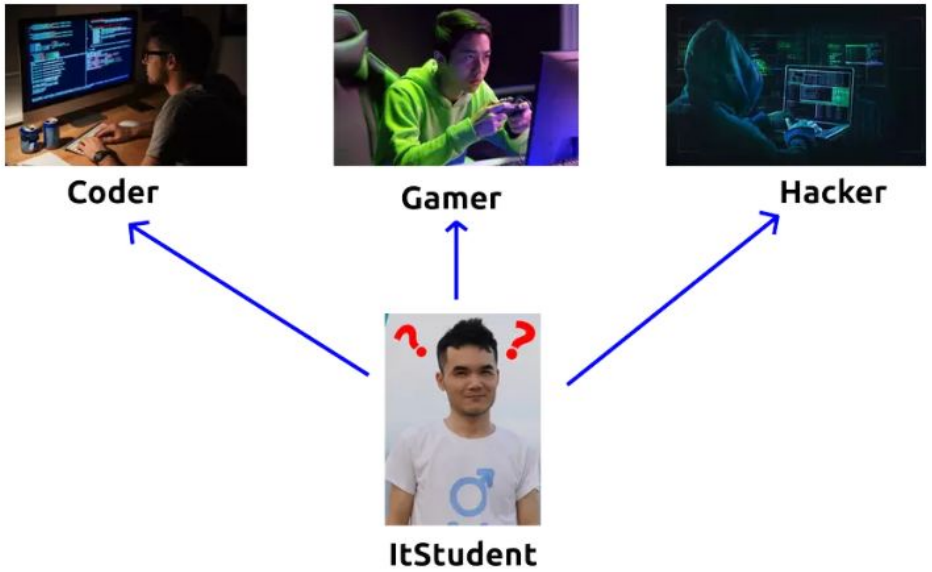
Trong trường hợp một con cá vừa bơi dưới nước nhưng cũng có thể bay (Cá chuồn bay)

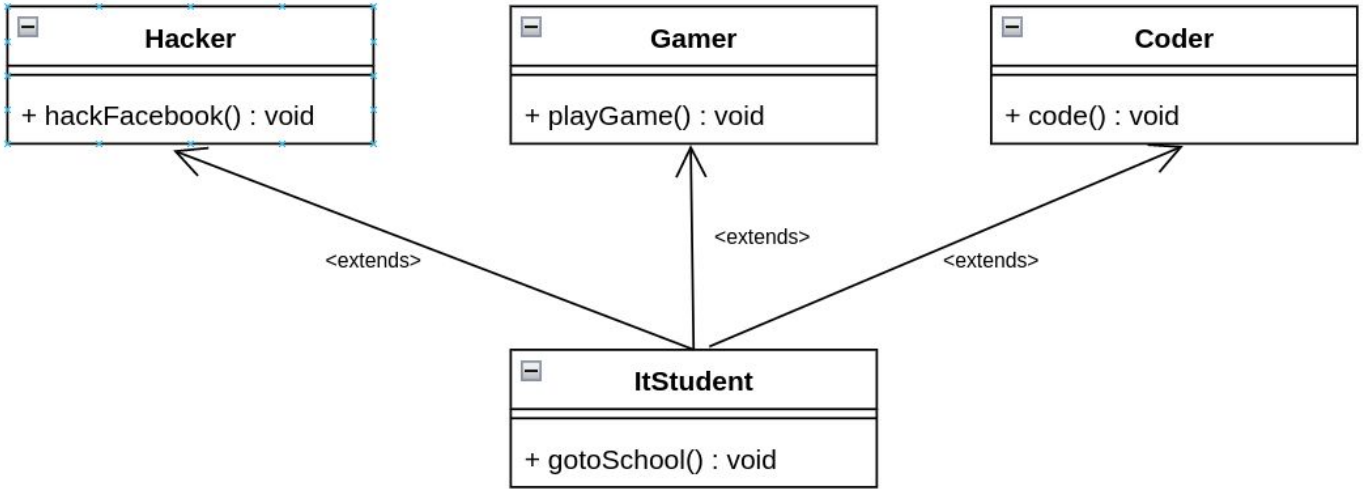
Cùng một lúc class FlyingFish triển khai Swimming và Flying, vậy là con cá vừa có thể bơi cũng có thể bay

```
public class FlyingFish implements Swimming, Flying{
    private String name;
    @Override
    public void fly() {
        System.out.println(name + " is flying!");
    }

    @Override
    public void flapWings() {
        System.out.println(name + " is flapping its wings!");
    }

    @Override
    public void swim() {
        System.out.println(name + " is swimming!");
    }
}
```

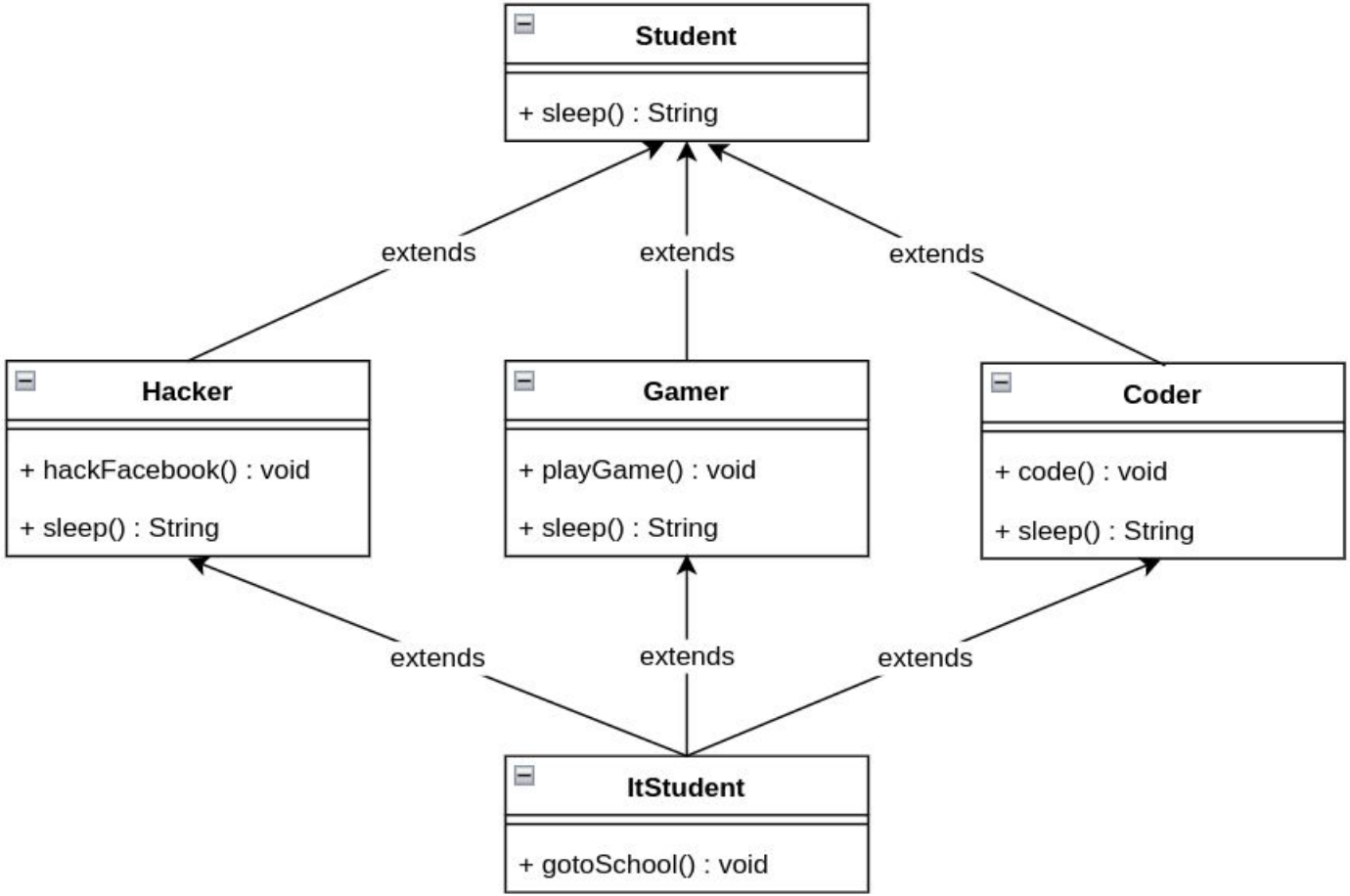
Diamond Problem

Đa kế thừa là trường hợp một class kế thừa **nhiều hơn** một class khác. Tuy nhiên trong Java, đa kế thừa có thể dẫn đến một vấn đề là **Diamond Problem**

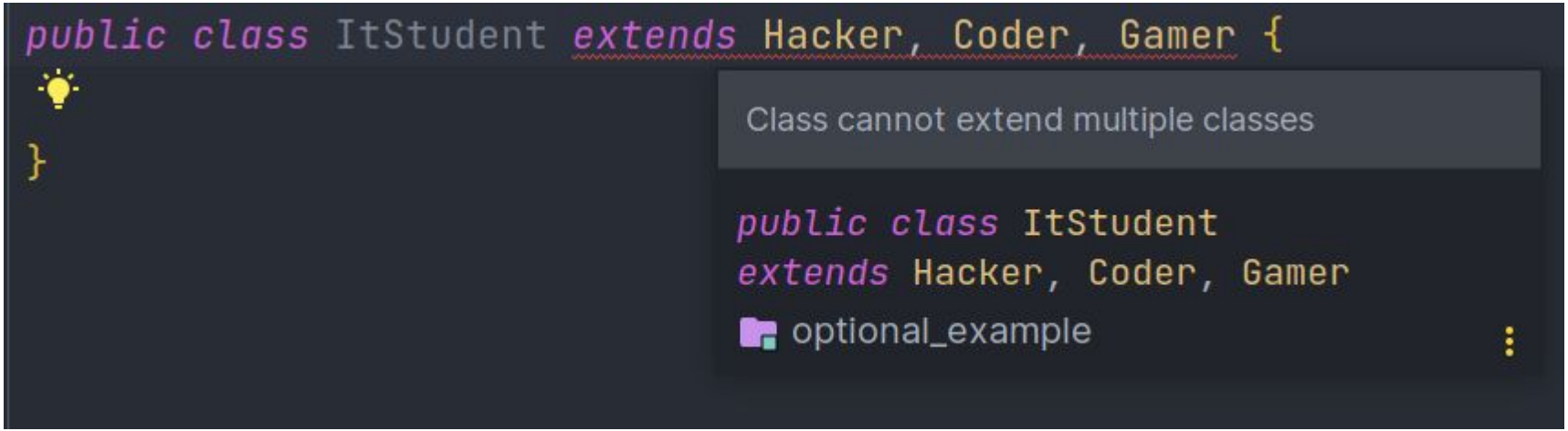




Mọi thứ cho đến lúc này vẫn ổn đúng không nào, nhưng hãy tưởng tượng nếu mình cho ba lớp Hacker, Gamer, Coder cùng kế thừa từ một lớp là Person


Student
+ sleep() : String
extends
extends
extends
Hacker
+ hackFacebook() : void
+ sleep() : String
Gamer
+ playGame() : void
+ sleep() : String
Coder
+ code() : void
+ sleep() : String
extends
extends
extends
ItStudent
+ gotoSchool() : void

Giả sử nếu là Hacker sẽ ngủ ngày thức đêm, Gamer sẽ thức cả ngày, còn Coder thức ngày ngủ đêm. Vậy câu hỏi trong một ngày thằng sinh viên It chỉ có thể hoặc là làm Hacker hoặc là Gamer hoặc là Coder (vì nó còn phải đi học mà) thì nó sẽ ngủ như thế nào? Câu này sẽ khó nếu không biết hôm đó được nghỉ sáng, nghỉ tối hay nghỉ cả ngày!



Một class không thể kế thừa từ nhiều class khác nhưng có thể implements nhiều interface khác nhau một lúc
Vậy để giải quyết vấn đề này, thì vì tạo 2 class Hacker, Coder và Gamer thì ta sẽ tạo 3 interface

```
public interface ICoder {
    String sleep(String time);
    void code();
}
```

```
public interface IHacker {
    String sleep(String time);
    void hackFacebook();
}
```

```
public interface IGamer {
    String sleep(String time);
    void playGame();
}
```

```java
public class ItStudent implements IHacker, IGamer, ICoder {
    @Override
    public void code() {
    }

    @Override
    public void playGame() {
    }

    @Override
    public void hackFacebook() {
    }

    @Override
    public String sleep(String time) {
        return time;
    }
}
```

Và khi muốn ItStudent là hacker, coder hay gamer các bạn chỉ cần làm như sau:

```
public class Main {
    public static void main(String[] args) {
        IGamer gamer = new ItStudent();
        gamer.playGame();
        gamer.sleep("no sleep :)");

        ICoder coder = new ItStudent();
        coder.code();
        coder.sleep("I work on morning and sleep in the evening" );

        IHacker hacker = new ItStudent();
        hacker.hackFacebook();
        hacker.sleep("I work on evening and sleep in the morning" );
    }
}
```

Ở đây ta thấy các interface ICoder, IHacker và IGamer đề có chung hàm **String sleep(String time);** nhưng khi ItStudent implements chúng thì chỉ có một hàm được triển khai. Đây chính là đặc điểm của interface, nó có khả năng thay đổi thay đổi hành vi ở runtime. Và cũng là lý do tại sao chúng ta dùng interface để đạt được mục đích "*đa kế thừa*"

So sánh Abstract class với interface

Interface	Abstract class
Interface chỉ chứa phương thức abstract. Tuy nhiên, từ phiên bản Java 8, Interface có thể chứa các phương thức non-abstract bằng cách sử dụng từ khóa **default** hoặc **static**	Một abstract class có thể chứa các phương thức abstract hoặc non-abstract
Interface chỉ chứa các biến **static** và **final**	Abstract class có thể chứa các biến **final**, **non-final**, **static** và **non-static**
Một abstract class có thể implements một interface	Interface lại không thể implements một abstract class
Interface được triển khai bằng cách sử dụng từ khóa **implements**	Abstract class được kế thừa bằng cách sử dụng **extends**