

---

# 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 các hàm của **interface** đó





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

# Đặc điểm của interface

01

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

02

Tất cả các phương thức trong interface không có thân hàm, 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();  
}
```



## Khai báo biến trong interface

- Tất cả các biến trong interface đều được ngầm hiểu là public static final
- Cấu trúc: <Type> name\_var

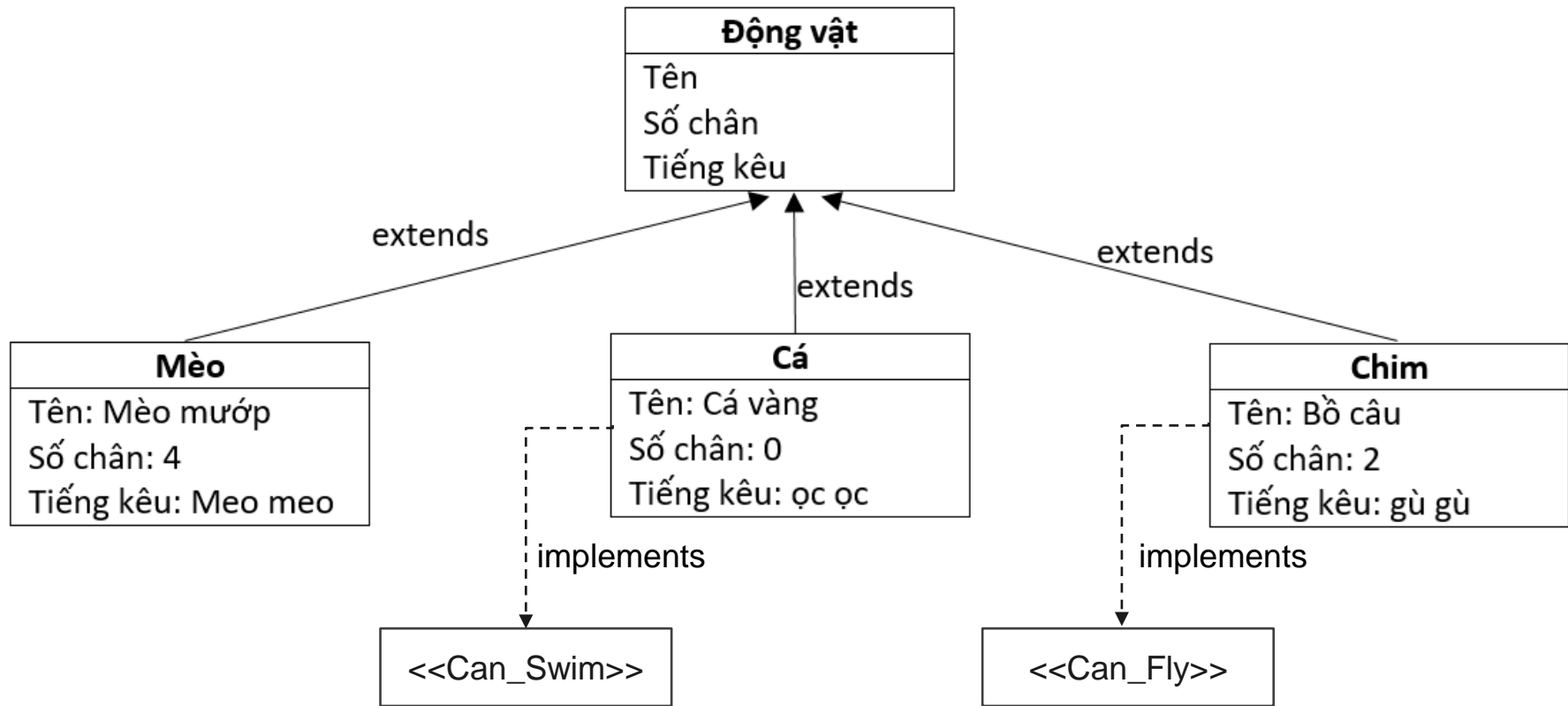
```
public interface Car {  
    String name= "Car"; // Tương đương public static final String name= "Car"  
    public static final String name2 = "Car";  
  
}
```

# Tại sao chúng ta lại sử dụng interface?

---

- 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







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

```
public class Bird implements CanFly {  
    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 CanSwim và CanFly, vậy là con cá vừa có thể bơi cũng có thể bay

```
public class FlyingFish implements CanSwim, CanFly {  
    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!");  
    }  
}
```



## Bài tập

- Tạo một interface INews chứa hàm void display()
- Tạo class News: (String) title, (String) author, (String) publicDate, (double) rate
- Set(gán) giá trị cho các thuộc tính của News
- Thực hiện implements INews và hiển thị các thông tin của class News



## Bài tập: Liên minh huyền thoại

- Tạo class Yasuo: (String) hair, (String) weapon: sword
- Tạo class Lucian: (String) hair, (String) weapon: pistols
- Tạo interface CanShoot: void shoot()
- Tạo interface CanSurf: void surf()
- NOTE: Yasuo thì có thể lướt, Lucian thì vừa có thể lướt, vừa có thể bắn.  
-> Hãy implements sao cho thật hợp lý



## Bài tập: Tính chu vi và diện tích các đa giác

- Tạo interface đa giác Polygon: void display(), void calArea()
- Tạo lớp Rectangle: (double) length, (double) width
- Tạo lớp Square: (double) side
- Cho 2 lớp trên implements interface Polygon để thực hiện hiển thị thông tin và tính diện tích



## Bài tập: Xây dựng chương trình quản lý học sinh

- Tạo lớp Student: name, age, marks, classification
- Thực hiện set giá trị cho: name, age, marks
- Xây dựng dựng interface Iclassification: classify(), display()
- Cho Student implements Iclassification để thực hiện các hàm trên
- Đối với việc xếp loại (thực hiện hàm classify() ) :
  - + Nếu điểm  $\geq 8$  loại A
  - +  $< 8$  và  $> 6.5$  loại B
  - + Còn lại là loại C

Hiển thị thông tin chi tiết của học sinh

# So sánh Interface với Abstract Class

Interface	Abstract Class
Chỉ chứa abstract methods. Tuy nhiên, từ Java 8, có thể chứa các phương thức non-abstract bằng cách sử dụng từ khóa <b>default</b> hoặc <b>static</b>	Một abstract class có thể chứa các phương thức abstract hoặc non-abstract
Interface chỉ chứa các biến <b>static final</b>	Abstract class có thể chứa các biến <b>final</b> , <b>non-final</b> , <b>static</b> và <b>non-static</b>
Abstract class có thể implements interface	Interface không thể implements abstract class
Interface được triển khai bằng cách sử dụng từ khóa <b>implements</b>	Abstract class được kế thừa bằng cách sử dụng từ khóa <b>extends</b>





# Phần mở rộng

## Một Interface cũng có thể extends một hoặc nhiều interface khác

```
public interface Car {  
    String name= "Car"; // Tương đương public static final name= "Car"  
}
```

```
public interface Service {  
    void fix();  
}
```

```
public interface CarService extends Service, Car{  
    void cleanCar();  
}
```



## Interface phục vụ tính chất nào trong 4 tính chất của OOP?

- Đầu tiên, tất nhiên là **tính trừu tượng**: interface là một tập hợp các phương thức để các class tuân theo. Có nghĩa là là trừu tượng hoá, đại diện cho các class tuân thủ (implement)
- Interface có thể kế thừa interface -> **Tính kế thừa**
- Khi một Class implements Interface, nó sẽ phải ghi đè (override) phương thức bên trong interface -> **Tính đóng gói**. VD: class Dog và class Cat implements I\_Animal { void speech() } Trong Interface có static và (từ java 9) private method, vậy nó cũng hỗ trợ tính đóng gói.
- Nhiều class có thể tuân thủ một interface. Mỗi một mẫu hàm lại có nhiều cách hiện thực hoá khác nhau đa dạng.  
Chúng ta có thể khởi tạo IAnimal animal = new Dog() -> **Tính đa hình**



## Static method trong interface

- Cấu trúc khai báo: `static <type> nameMethod(){ }`
- Đặc điểm:
  - + Có thân hàm
  - + **Không thể** override khi có Class implements

```
public interface Car {"  
    static void checkStatic(){  
        System.out.println("Đây là static method");  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Car.checkStatic();  
    }  
}
```


Result: “Đây là static method”



## Default method trong Interface

- Cấu trúc khai báo: `default <type> nameMethod(){ }`
- Đặc điểm:
  - + Có thân hàm
  - + **Có thể** override method khi có Class implements

```
public interface Car {  
  
    default void checkDefault(){  
        System.out.println("Đây là default method");  
    }  
}
```



```
public class CarServiceImpl implements CarService, Car{  
    @Override  
    public void cleanCar() {  
  
    }  
  
    @Override  
    public void fix() {  
  
    }  
  
    @Override  
    public void checkDefault() {  
  
        System.out.println("Đây là CarServiceImpl khi được override");  
    }  
}
```

## Quiz

<https://www.cs.umd.edu/class/summer2018/cmsc132/javatest/interfaces/interfaces.html>



## Kết hợp Interface với Generic

```
interface writer<T> {  
    void update(T obj);  
    void delete(T obj);  
    void write(T obj);  
}
```

```
class Book<T> implements writer<T> {  
    @Override    public void update(T obj) { // do something    }  
    @Override    public void delete(T obj) { // do something    }  
    @Override    public void write(T obj)  { // do something    }  
}
```

# Một class có thể implements nhiều interface cùng lúc. Vậy điều gì sẽ xảy ra khi các interface này có các default methods giống nhau?


Khi một class thực hiện nhiều interface chứa default method cùng tên sẽ xảy ra xung đột (conflict). Bởi vì lúc này Java sẽ không biết phải sử dụng phương thức mặc định nào. Khi đó, trình biên dịch của Java sẽ thông báo lỗi tương tự như sau:



```
interface Interface1 {  
    default void doSomething() {  
    }  
}
```

```
interface Interface2 {  
    default void doSomething() {  
    }  
}
```

```
public class MultiInheritance implements Interface1, Interface2 {  
}
```

 Duplicate default methods named doSomething with the parameters () and () are inherited from the types Interface2 and Interface1

2 quick fixes available:

- [Override default method in 'Interface1'](#)
- [Override default method in 'Interface2'](#)

Để giải quyết vấn đề này chúng ta có thể sử dụng một trong hai cách để giải quyết bên dưới:

- Override lại phương thức doSomething từ lớp con.
- Gọi default method của một interface cụ thể bằng cách sử dụng từ khóa super.

```
public class CarServiceImpl implements CarService, Car{
    @Override
    public void cleanCar() {

    }

    @Override
    public void fix() {

    }

    @Override
    public void checkDefault() {
        Car.super.checkDefault();
        //
        System.out.println("Đây là CarServiceImpl khi được override");
    }
}
```



## Một số interface phổ biến

- Serializable
- Cloneable
- Comparable
- Comparator
- ...

```
public class Student implements Serializable {  
    int id;  
    String name;  
  
    public Student(int id, String name) {  
        this.id = id;  
        this.name = name;  
    }  
}
```

```
class Multi3 implements Runnable {  
    public void run() {  
        System.out.println("thread is running...");  
    }  
  
    public static void main(String args[]) {  
        Multi3 m1 = new Multi3();  
        Thread t1 = new Thread(m1);  
        t1.start();  
    }  
}
```

```

public class Student implements Cloneable {
    int rollno;
    String name;

    Student(int rollno, String name) {
        this.rollno = rollno;
        this.name = name;
    }

    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

    public static void main(String args[]) {
        try {
            Student s1 = new Student(101, "An");
            Student s2 = (Student) s1.clone();
            System.out.println(s1.rollno + " " + s1.name);
            System.out.println(s2.rollno + " " + s2.name);
        } catch (CloneNotSupportedException c) {
        }
    }
}

```

```

class Student implements Comparable<Student> {
    private int id;
    private String name;
    private int age;
    private String address;

    public Student() {
    }

    public Student(int id, String name, int age, String address) {
        super();
        this.id = id;
        this.name = name;
        this.age = age;
        this.address = address;
    }

    // getter & setter

    @Override
    public String toString() {
        return "Student@id=" + id + ",name=" + name
            + ",age=" + age + ",address=" + address;
    }

    @Override
    public int compareTo(Student student) {
        // sort student's name by ASC
        return this.getName().compareTo(student.getName());
    }
}

```



## BTVN: Tài khoản ngân hàng

- Tạo class Account: name, accountNumber, email, accountBalance
- Tạo interface IAccount: recharge(double amount); changeEmail(String email); displayInfo();
- Account implements IAccount và thực hiện các hàm trên

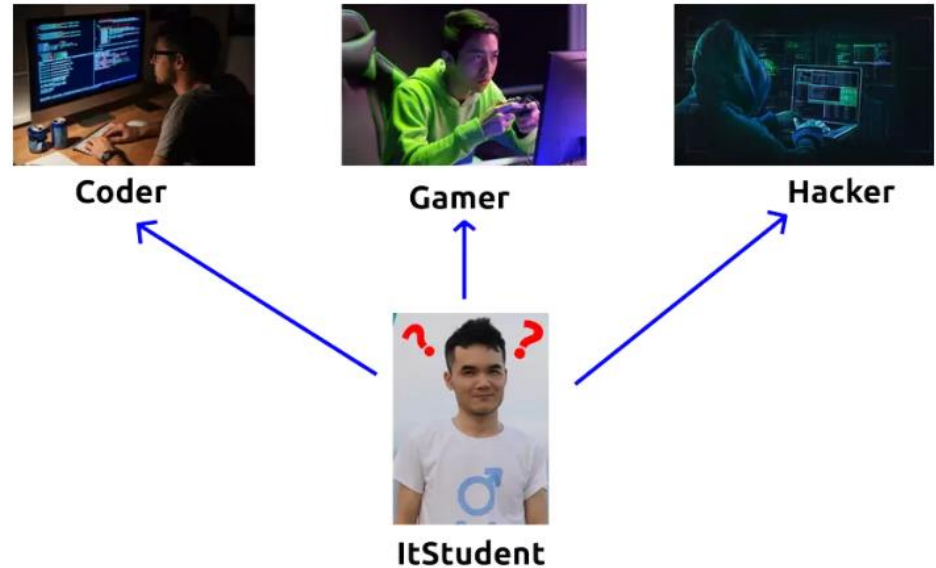
---

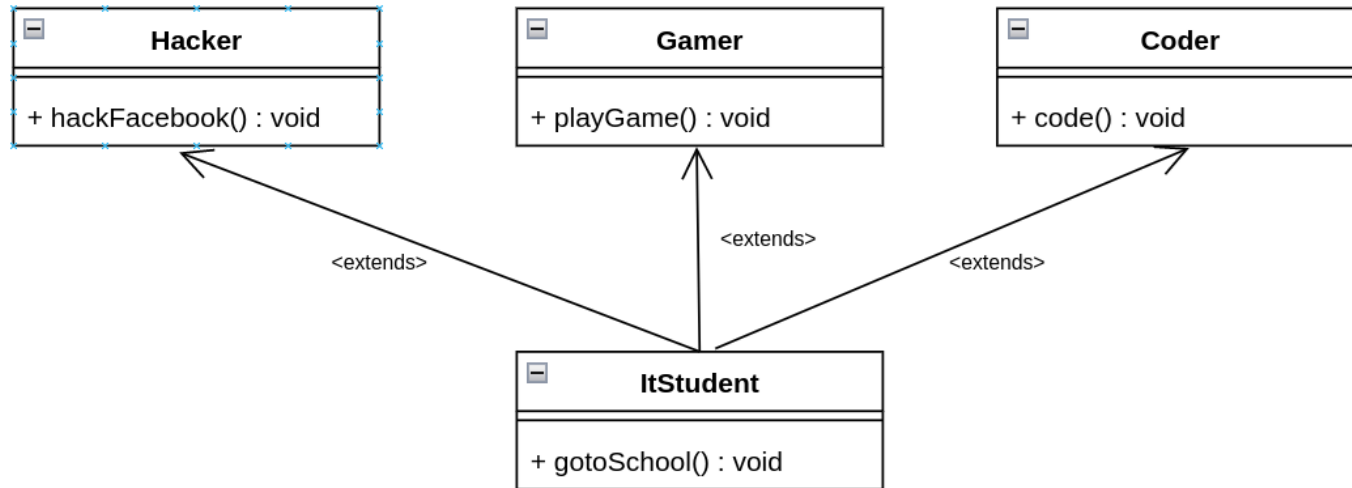
# Sử dụng Interface để “mô phỏng” Đa kế thừa



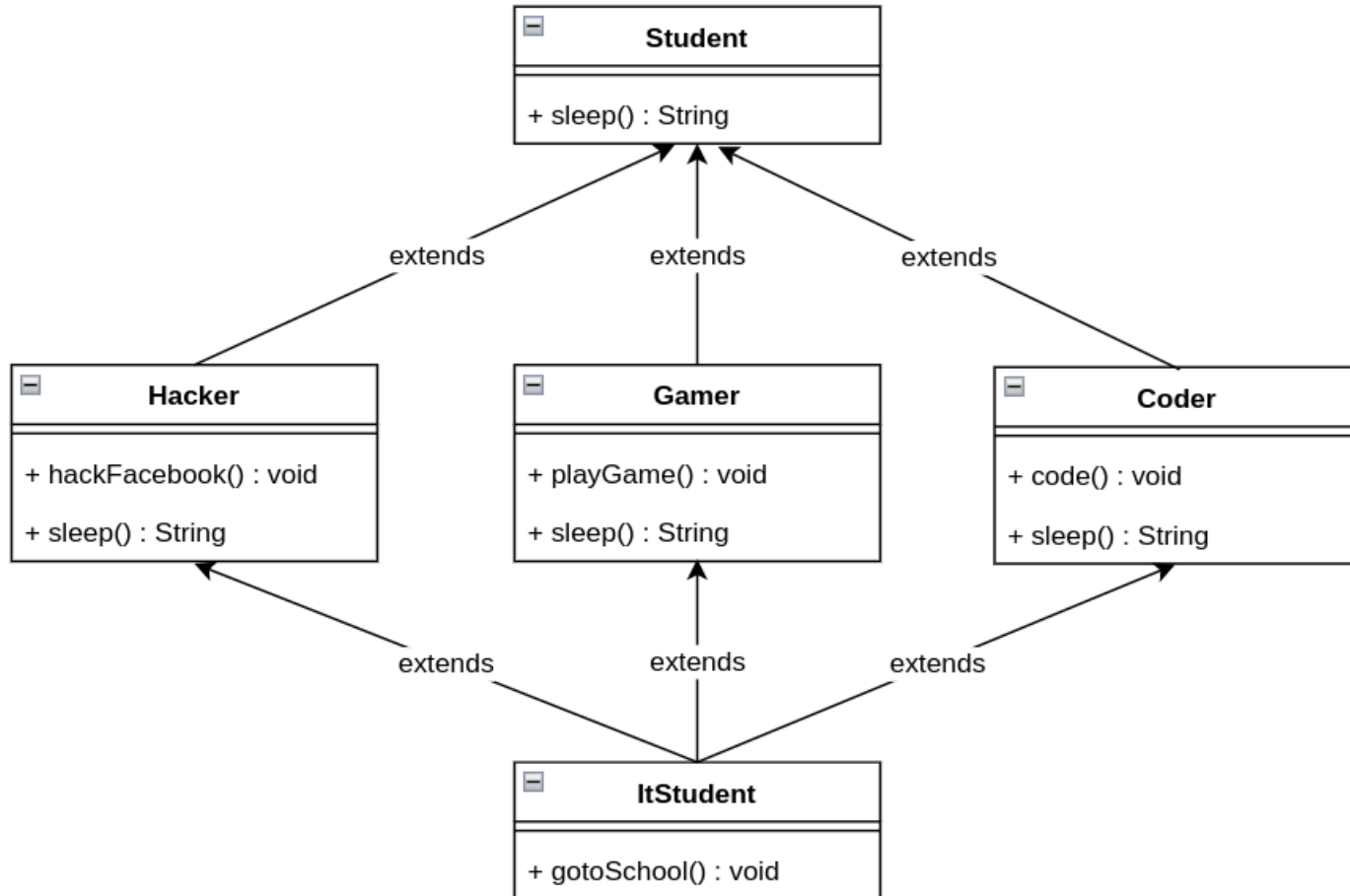
# Diamond Problem

Đa kế thừa là trường hợp một class kế thừa **nhều 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, nhưng hãy tưởng tượng nếu cho ba lớp Hacker, Gamer, Coder cùng kế thừa từ một lớp là Person



Giả sử Hacker sẽ ngủ ngày thức đêm, Gamer sẽ thức cả ngày, còn Coder thức ngày ngủ đêm. Vậy câu hỏi trong một ngày thẳng 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!


```
public class ItStudent extends Hacker, Coder, Gamer {
```



```
}
```

Class cannot extend multiple classes

```
public class ItStudent  
extends Hacker, Coder, Gamer
```

 optional\_example



Một class không thể kế thừa từ nhiều class khác nhau nhưng có thể implements nhiều interface khác nhau cùng lúc.

Vậy để giải quyết vấn đề này, thay vì tạo 3 class Hacker, Coder, 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();  
}
```

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

Ở đây ta thấy các interface ICoder, IHacker và IGamer đều 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 hành vi ở runtime. Và cũng là lý do tại sao chúng ta dùng interface để đạt được mục đích "*đa kế thừa*"

