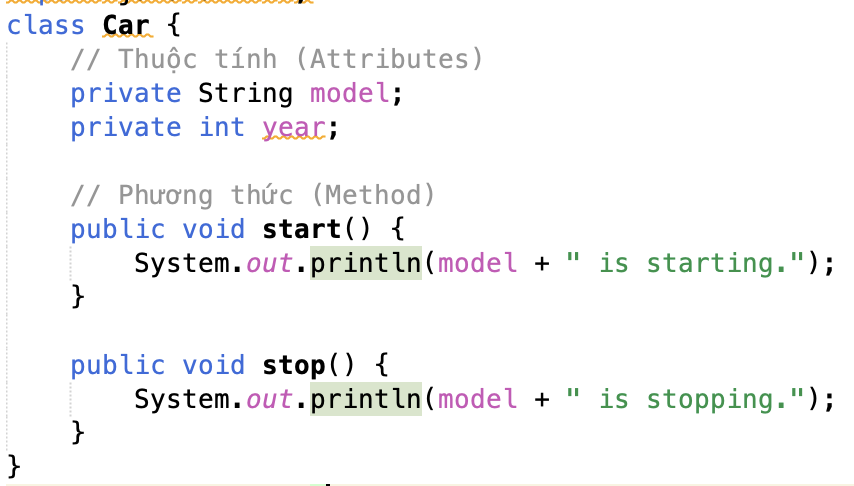
**1.Class**

- Định nghĩa : **Class**(lớp) trong Java là một bản thiết kế định nghĩa các trường dữ liệu và phương thức của đối tượng(Object) sẽ là gì. Một đối tượng là một thể hiện của một lớp. Mối quan hệ giữa lớp và đối tượng tương tự như mối quan hệ giữa công thức làm bánh táo và bánh táo: Bạn có thể làm bao nhiêu bánh táo tùy thích từ một công thức duy nhất.

- Đặc điểm của Class:

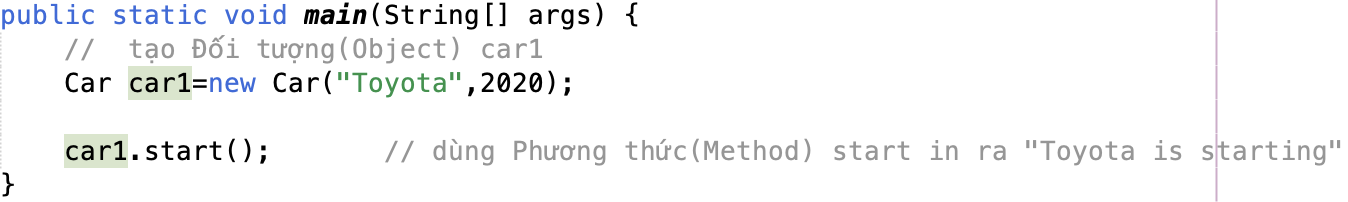
 + **Class** chứa các **thuộc tính** và **phương thức**

VD:



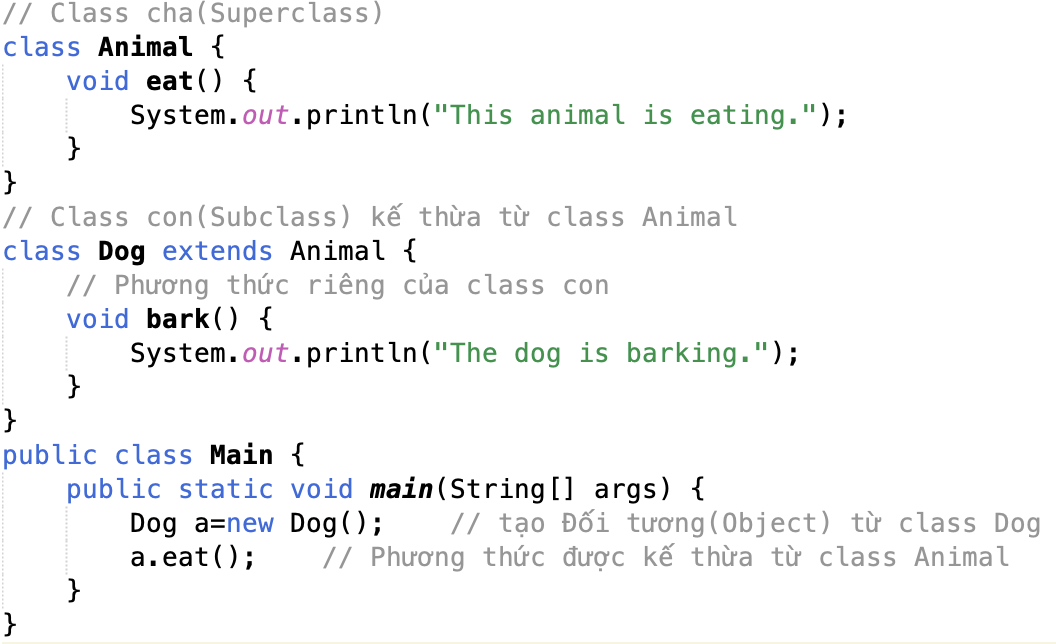
 + **Class** có thể khởi tạo một đối tượng(Object).(sử dụng từ khóa **new**)

VD:

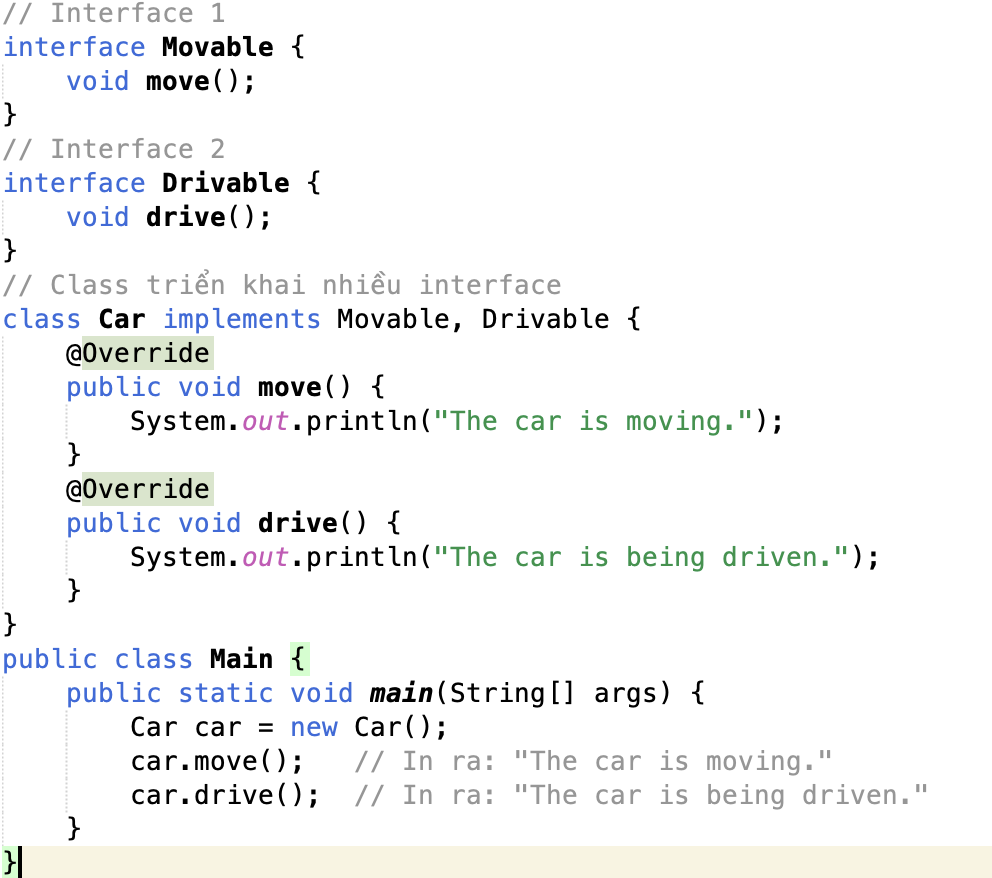


 + **Class** có thể kế thừa từ class khác (sử dụng từ khóa **extends**)(class chỉ hỗ trợ **đơn kế thừa**).

VD:



 + **Class** có thế triển khai một hoặc nhiều **Interface** (sử dụng từ khóa **Implements**)



**2.Abstract class**

### a. Tính trừu tượng:

**-** Tính trừu tượng là một khái niệm trong lập trình hướng đối tượng, nó cho phép ta chỉ ra những gì mà một đối tượng có thể làm, nhưng không nói ra làm thế nào để đối tượng đó làm được điều đó.

- Ví dụ, khi ta sử dụng điện thoại để gửi tin nhắn, ta chỉ biết soạn nội dung tin, nhập thông tin người dùng rồi ấn gửi đi nhưng ta không cần biết quy trình xử lý tin nhắn gửi đi như thế nào.

- Tính trừu tượng là một khái niệm quan trọng trong lập trình hướng đối tượng, nó cho phép ta tách biệt được phần logic của đối tượng và phần giao tiếp với đối tượng. Ta chỉ quan tâm đến phần giao tiếp với đối tượng, còn phần logic của đối tượng thì ta không cần quan tâm. Điều này giúp ta có thể dễ dàng thay đổi phần logic của đối tượng mà không ảnh hưởng đến phần giao tiếp với đối tượng.

**b. Abstract class:**

- Abstract Class – Lớp trừu tượng, là 1 lớp trong Java, nhưng có thể chứa cả các phương thức trừu tượng (abstract method) và các phương thức thường (non-abstract method).

- Abstract Class không thể khởi tạo đối tượng, nhưng có thể khai báo biến kiểu Abstract Class, và khởi tạo đối tượng của class con của Abstract Class.

Ví dụ:

abstract class Animal {  
 public abstract void eat();  
 public void travel() {  
 System.*out*.println("Animal is travelling");  
 }  
}

class Dog extends Animal {  
 @Override  
 public void eat() {  
 System.*out*.println("Dog is eating");  
 }  
}  
class Cat extends Animal {  
 @Override  
 public void eat() {  
 System.*out*.println("Cat is eating");  
 }  
}  
public class Main {  
 public static void main(String[] args) {  
 Animal dog = new Dog();  
 Animal cat = new Cat();  
  
 dog.eat();  
 dog.travel();  
  
 cat.eat();  
 cat.travel();  
 }  
}

Kết quả khi chạy chương trình:

Dog is eating

Animal is travelling

Cat is eating

Animal is travelling

**c. Tổng hợp:**

- Như vậy, Abstract Class nhìn chung giống Class, nhưng mà có thể tạo thêm phương thức trừu tượng, các class kế thừa từ Abstract Class này phải implement các phương thức trừu tượng này, nếu không sẽ bị lỗi compile.

- Các phương thức trừu tượng này, có thể hiểu là các phương thức mà Abstract Class này quy định, các class kế thừa từ Abstract Class này phải có, và phải implement các phương thức này.

- Các phương thức thường (non-abstract method) thì không cần implement, vì các class kế thừa từ Abstract Class này đã có sẵn các phương thức này.

- Abstract Class thì không thể khởi tạo đối tượng, nhưng có thể khai báo biến kiểu Abstract Class, và khởi tạo đối tượng của class con của Abstract Class.

Ví dụ:

abstract class Animal {  
 public abstract void eat();  
 public void travel() {  
 System.*out*.println("Animal is travelling");  
 }  
}  
class Dog extends Animal {  
 @Override  
 public void eat() {  
 System.*out*.println("Dog is eating");  
 }  
}  
public class Main {  
 public static void main(String[] args) {  
 Animal animal = new Animal(); *// Lỗi compile* Animal dog = new Dog(); *// OK* }  
}

- Giải thích lỗi compile, vì abstract class hay interface chứa những hàm trừu tượng, những hàm chỉ định nghĩa, chưa có logic và giao việc triển khai logic cho các nơi ghi đè nó, nên không thể khởi tạo đối tượng từ abstract class hay interface được. (Vì nó chưa có logic, chưa có gì để chạy cả)

1. **Interface**

Định nghĩa: Interface có cấu trúc giống như lớp, chỉ chứa các hằng số và phương thức trừu tượng.

Interface tương tự như một lớp trừu tượng, nhưng mục đích của nó là chỉ định các hành vi chung cho các đối tượng của các lớp liên quan hoặc không liên quan. Ví dụ sử dụng interface để chỉ định đối tượng có thể ăn được(the objects are edible)

A blue rectangle with black text

Description automatically generated

Interface giống như lớp trừu tượng, bạn không thể tạo một instance từ interface sử dụng toán tử **new**

A close-up of a computer screen

Description automatically generated

Có thể sử dụng interface chỉ ra rằng xem 1 đối tượng có **Edible hay không.**Ví dụ Chicken

A close up of a sign

Description automatically generated

A screen shot of a computer code

Description automatically generated

Lớp Animal là lớp trừu tượng có phương thức trưu tượng sound() được thực hiện bởi một concrete animal class.  
Lớp Chicken sẽ implements Edible để xem Chicken có ăn được.Khi implements howToEat() của interface Edible thì lớp Chicken phải implements tất cả các phương thức được định nghĩa trong interface đó.

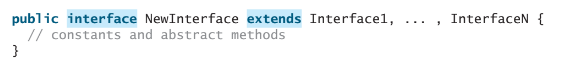
**Interfaces vs Abstract Classs**  
A close-up of a text

Description automatically generated

Một class có thể triển khai nhiều interfaces nhưng nó chỉ có thể kế thừa từ 1 lớp cha.

A close-up of a text

Description automatically generated

Một interface có thể extends nhiều interface khác  


Interfaces được ưu tiên sử dụng hơn abstract classes bời vì interface có thể định nghĩa common supertype cho lớp không liên quan. Interface linh hoạt hơn  
Ví dụ có **howToEat()** method được định nghĩa trong **Animal** class

A close-up of a computer code

Description automatically generated

Hai lớp **Chicken và Duck kế thừa từ Animal**:  
A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

Khi bạn định nghĩa 1 lớp con của Animal. Điều này là hạn chế.Khi đó lớp con phải là một another animal.  
Còn Interfaces không có hạn chế, linh hoạt hơn classes.

A close up of a text

Description automatically generated

A screenshot of a computer code

Description automatically generated

A screenshot of a computer code

Description automatically generated

**\* So sánh Interface và Abstract Class**

**-** Giống nhau:

+ Đều được sử dụng để đạt được tính trừu tượng trong OOP.

+ Đều được sử dụng trong kế thừa.

+ Đều có thể chứa cả phương thức và các trường dữ liệu(variables).

+ Đều không thể tạo đối tượng cụ thể của chúng với keyword new.

- Khác nhau:

|  |  |  |
| --- | --- | --- |
| Tiêu chí | Abstract Class | Interface |
| Loại phương thức | Abstract Class có thể có các phương thức abstract , non-abstract, constructor, static | Phiên bản Java < 8, Interface chỉ có thể có phương thức abstract. Phiên bản Java 8, có thể thêm **default**và**static methods**. Phiên bản Java 9, có thể thêm **private methods**. |
| Đa kế thừa | Abstract Class **không** hỗ trợ đa kế thừa,  Abstract Class chỉ có thể kế thừa 1 abstract class khác | Interface hỗ trợ **đa kế thừa,**  Một interface có thể kế thừa nhiều interface khác |
| Đa thực thi | Abstract class có thể extends lớp khác và implements nhiều interface tùy ý | Interface chỉ có thể extends các interface khác |
| Tính trừu tượng | Abstract Class không đạt tính trừu tượng hoàn toàn (100%) | Interface đạt tính trừu tượng hoàn toàn (100%) |
| Các kiểu biến | Abstract Class có thể có các biến **final, non-final, static và non-static** | Interface chỉ có các biến **static final, b**iến trong interface mặc định là public static final |
| Access modifiers | Hỗ trợ tất cả các loại access modifier bao gồm private, protected và public | Mặc định các thành phần của interface là public. Java 9+ hỗ trợ thêm private |
| Sự triển khai lẫn nhau | Abstract class có thể cung cấp các triển khai cho interface | Interface không thể cung cấp triển khai cho abstract class |
| Constructor | Có thể chứa các constructor | Không hỗ trợ |
| Mục đích | Làm lớp cha chung của một nhóm các lớp có liên quan | Định ra hành động có thể dùng chung cho các lớp |
| Tốc độ | Nhanh | Chậm hơn vì phải tốn thêm thao tác điều hướng |

Ví dụ 1: **Loại phương thức**

**Abstract Class** có thể có phương thức abstract, non-abstract, constructor và static:

abstract class Animal {  
 *// Phương thức abstract* abstract void makeSound();  
 *// Phương thức non-abstract* public void sleep() {  
 System.*out*.println("Sleeping...");  
 }  
 *// Phương thức static* public static void breathe() {  
 System.*out*.println("Breathing...");  
 }  
 *// Constructor* public Animal() {  
 System.*out*.println("Animal created.");  
 }  
}

**Interface** trong các phiên bản Java 8, 9 có thể có default, static, và private methods:  
interface Bird {  
 *// Phương thức abstract* void fly();  
 *// Phương thức default (Java 8+)* default void layEggs() {  
 System.*out*.println("Laying eggs...");  
 }  
 *// Phương thức static (Java 8+)* static void migrate() {  
 System.*out*.println("Migrating...");  
 }  
 *// Phương thức private (Java 9+)* private void rest() {  
 System.*out*.println("Resting...");  
 }  
}

Ví dụ 2: **Đa kế thừa và Đa thực thi**

**Abstract Class** không hỗ trợ đa kế thừa, nhưng có thể implement nhiều interface:

abstract class Mammal {  
 abstract void walk();  
}  
class Dolphin extends Mammal implements Swimmer, Jumper {  
 @Override  
 void walk() {  
 System.*out*.println("Dolphins can't walk, but they swim.");  
 }  
 @Override  
 public void swim() {  
 System.*out*.println("Dolphin is swimming.");  
 }  
  
 @Override  
 public void jump() {  
 System.*out*.println("Dolphin is jumping.");  
 }  
}

**Interface** hỗ trợ đa kế thừa:

interface Swimmer {  
 void swim();  
}  
interface Jumper {  
 void jump();  
}  
class Dolphin implements Swimmer, Jumper {  
 @Override  
 public void swim() {  
 System.*out*.println("Dolphin is swimming.");  
 }  
  
 @Override  
 public void jump() {  
 System.*out*.println("Dolphin is jumping.");  
 }  
}

**Ví dụ 3: Tính trừu tượng**

**Abstract Class** có thể chứa cả phương thức trừu tượng và không trừu tượng:

abstract class Vehicle {  
 abstract void start(); *// Phương thức trừu tượng* public void stop() { *// Phương thức không trừu tượng* System.*out*.println("Vehicle stopped.");  
 }  
}

**Interface** chỉ chứa các phương thức trừu tượng (trong các phiên bản trước Java 8):

interface Movable {  
 void move(); *// Phương thức trừu tượng*}

**Ví dụ 4: Biến trong Abstract Class và Interface**

**Abstract Class** có thể chứa các biến final, non-final, static, và non-static:

abstract class Creature {  
 static final int *LIFESPAN* = 100; *// Biến static final* int age; *// Biến non-static* public Creature() {  
 this.age = 0;  
 }  
}

**Interface** chỉ có các biến public static final:

interface Edible {  
 public static final String FOOD\_TYPE= "Plants"; *// Biến mặc định là public static final*}

**Ví dụ 5: Access Modifiers**

**Abstract Class** hỗ trợ tất cả các access modifiers (private, protected, public):

abstract class Machine {  
 protected abstract void operate(); *// Phương thức protected*}

**Interface** mặc định mọi thành phần là public (từ Java 9, có thêm private cho các phương thức):

interface Drivable {  
 public void drive(); *// Mặc định là public* private void checkFuel() { *// Phương thức private (Java 9+)* System.*out*.println("Checking fuel...");  
 }  
}

Ví dụ 6: **Constructor**

**Abstract Class** có thể có constructor:

abstract class Appliance {  
 public Appliance() {  
 System.*out*.println("Appliance is created.");  
 }  
}

**Interface** không có constructor:

interface Device {  
 void turnOn();  
}

* Note: Sau này interface có thể có các phương thức thường, nhưng mà vẫn không thể khởi tạo đối tượng từ interface được.
* Về biến trong interface, mặc định là **public static final**, nên khi khai báo biến trong interface, không cần khai báo public static final nữa. Tức là các biến trong interface là tồn tại duy nhất, không đổi được, ở dạng static.
* Về biến trong abstract class, không có mặc định gì cả, nên có thể tạo các “biến thường”, tức là mỗi object tạo ra bởi 1 class chứa giá trị khác nhau.

-> Nói về Abtract Class và Interface, đôi khi bạn sẽ gặp một số cách gọi: Khi một class **extend**một class/ abtract class thì có nghĩa là ta đang thể hiện mối quan hệ **is-a** (là), còn khi **implement**một interface, thì ta đang thể hiện mối quan hệ **has-a** (có, hay thực hiện).

Ví dụ:

// Programmer là Person, thực hiện việc Programming, Debugging

**class** Programmer **extends** Person **implements** Programming, Debugging {}

 // Rectangle là Shape, có Width, Height

**class** Rectangle **extends** Shape **implements** Width, Height {}

### \* Khi nào dùng interface, khi nào dùng abstract class?

**Sử dụng Abstract class:**

- Khi bạn có một nhóm các lớp liên quan cần chia sẻ chung một đoạn code hay tính năng nào đó. Bạn đưa các thành phần dùng chung vào lớp abstract và các lớp con liên quan sẽ kế thừa lớp cha abstract này.

- Khi bạn mong muốn rằng các lớp kế thừa lớp abstract có chung nhiều trường, phương thức hoặc muốn dùng chung cả các access modifier khác chứ không phải chỉ là access modifier public.

- Khi bạn muốn khai báo các trường non-static, non-final. Nhờ đó bạn có thể định nghĩa các phương thức có thể truy cập và sửa đổi trạng thái của từng đối tượng.

- Khi chúng ta chỉ có thể hoàn thành một vài chức năng (method/ function) chuẩn của hệ thống, một vài chức năng còn lại các lớp **extends** phải hoàn thành. Những tính năng đã hoàn thành này vẫn sử dụng như bình thường, đây là những tính năng chung.

Ví dụ:

*// Định nghĩa abstract class Shape*abstract class Shape {  
 protected String color;  
  
 public Shape(String color) {  
 this.color = color;  
 }  
 *// Phương thức trừu tượng để tính diện tích* public abstract double area();  
  
 *// Phương thức chung* public void displayColor() {  
 System.*out*.println("Color: " + color);  
 }  
}  
  
*// Lớp Circle kế thừa Shape*class Circle extends Shape {  
 private final double radius;  
  
 public Circle(String color, double radius) {  
 super(color);  
 this.radius = radius;  
 }  
  
 @Override  
 public double area() {  
 return Math.*PI* \* radius \* radius;  
 }  
}  
  
*// Lớp Rectangle kế thừa Shape*class Rectangle extends Shape {  
 private final double width;  
 private final double height;  
  
 public Rectangle(String color, double width, double height) {  
 super(color);  
 this.width = width;  
 this.height = height;  
 }  
  
 @Override  
 public double area() {  
 return width \* height;  
 }  
}  
public class ShapeTest {  
 public static void main(String[] args) {  
 Shape circle = new Circle("Red", 5.0);  
 Shape rectangle = new Rectangle("Blue", 4.0, 6.0);  
  
 circle.displayColor(); System.*out*.println("Circle area: " + circle.area());  
  
 rectangle.displayColor(); System.*out*.println("Rectangle area: " + rectangle.area());  
 }  
}

Kết quả khi chạy chương trình:

Color: Red

Circle area: 78.53981633974483

Color: Blue

Rectangle area: 24.0

**Sử dụng Interface:**

**-** Khi bạn muốn đạt được tính trừu tượng hoàn toàn. Tất cả các phương thức được nêu ra trong interface chưa có phần thân triển khai chi tiết và cần được triển khai cụ thể trong các lớp con implements interface đó.

- Khi bạn muốn đạt được tính đa kế thừa.

- Khi bạn muốn cho các lớp không liên quan gì đến nhau cũng có thể sử dụng chức năng của interface.

- Khi bạn muốn chỉ định các hành vi cần thực hiện nhưng không quan tâm các hành vi đó được thực hiện bởi ai, thực hiện như thế nào.

- Khi bạn muốn tạo dựng một bộ khung chuẩn gồm các chức năng (method/ function) mà tất cả module/ project cần phải có. Các module phải implements tất cả chức năng đã được định nghĩa.

Ví dụ:

*// Định nghĩa interface Transport*interface Transport {  
 void move();  
 double getSpeed();  
}  
  
*// Lớp Ship thực hiện interface Transport*class Ship implements Transport {  
 @Override  
 public void move() {  
 System.*out*.println("Ship is sailing.");  
 }  
  
 @Override  
 public double getSpeed() {  
 return 30.5; *// Vận tốc bằng hải trình* }  
}  
  
*// Lớp Airplane thực hiện interface Transport*class Airplane implements Transport {  
 @Override  
 public void move() {  
 System.*out*.println("Airplane is flying.");  
 }  
  
 @Override  
 public double getSpeed() {  
 return 900.0; *// Vận tốc bằng không trung* }  
}  
  
public class TransportTest {  
 public static void main(String[] args) {  
 Transport ship = new Ship();  
 Transport airplane = new Airplane();  
  
 ship.move();   
 airplane.move();   
  
 System.*out*.println("Ship speed: " + ship.getSpeed() + " knots.");  
 System.*out*.println("Airplane speed: " + airplane.getSpeed() + " km/h.");  
 }  
}

Kết quả khi chạy chương trình:

Ship is sailing.

Airplane is flying.

Ship speed: 30.5 knots.

Airplane speed: 900.0 km/h.