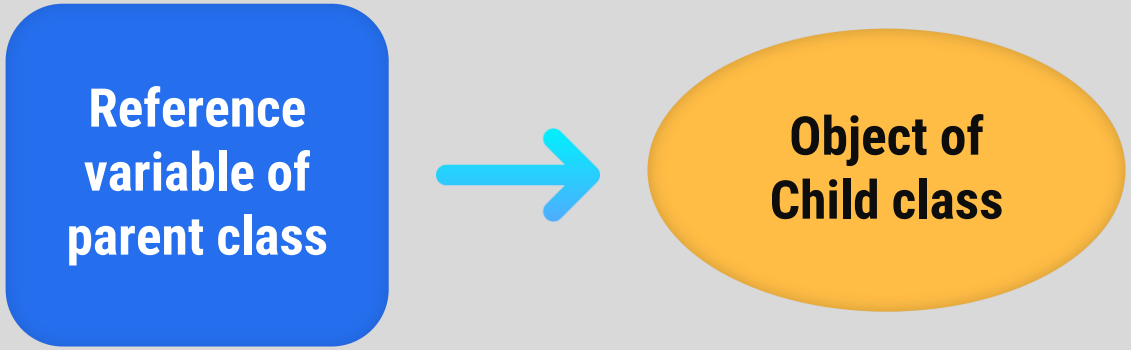
ĐA HÌNH VÀ TRỪU TƯỢNG

1. Đa hình:

Đa hình - Polymorphism là một trong 3 tính chất quan trọng của lập trình hướng đối tượng (bên cạnh Đóng gói - Encapsulation và Kế thừa - Inheritance). Đa hình cho phép bạn tham chiếu một biến thuộc kiểu dữ liệu của lớp cơ sở tới đối tượng của một lớp con.

Ví dụ: Lớp Student kế thừa từ lớp Person thì tất cả các thực thể (instance) của lớp Student đều là thực thể của lớp Person, nhưng ngược lại thì không.



1. Đa hình:

Ví dụ: Kế thừa lớp Person và nạp chồng phương thức display()

EXAMPLE

```
public class Person {
    public void display(){
        System.out.println("Person !");
    }
}

public class Student extends Person{
    public void display(){
        System.out.println("Student !");
    }
}

public class Staff extends Person {
    public void display(){
        System.out.println("Staff !");
    }
}
```

```
public class Lecturer extends Person {
    public void display(){
        System.out.println("Lecturer !");
    }
}

public class Main {
    public static void main(String[] args) {
        Person p1 = new Student();
        Person p2 = new Staff();
        Person p3 = new Lecturer();
        p1.display();
        p2.display();
        p3.display();
    }
}
```

OUTPUT

```
Student !
Staff !
Lecturer !
```

2. Dynamic Binding:

Trong kế thừa nhiều mức (Multilevel Inheritance) một phương thức có thể được ghi đè ở nhiều lớp trong chuỗi kế thừa, máy ảo Java (JVM) sẽ quyết định phương thức nào được gọi lúc Runtime.


```java
public class Person {
    public void display(){
        System.out.println("Person !");
    }
}

public class Student extends Person{
    public void display(){
        System.out.println("Student !");
    }
}

public class Pupil extends Student{
    public void display(){
        System.out.println("Pupil !");
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Person p1 = new Person();
        Person p2 = new Student();
        Person p3 = new Pupil();
        p1.display();
        p2.display();
        p3.display();
    }
}
```

OUTPUT

```
Person !
Student !
Pupil !
```

3. Ép kiểu đối tượng và toán tử instanceof:

Một biến tham chiếu của đối tượng có thể được ép sang tham chiếu của một đối tượng thuộc lớp khác, đây được gọi là **ép kiểu đối tượng**

Implicit casting và Explicit casting


```
public class Main {
    public static void main(String[] args) {
        Object ob = new Student(); //Implicit casting
        Student s = (Student)ob; // Explicit casting
    }
}
```

3. Ép kiểu đối tượng và toán tử instanceof:

Ta có thể ép một instance của lớp con sang một biến đối tượng của lớp cha **(upcasting)**, vì một đối tượng của lớp con bao giờ cũng là một đối tượng của lớp cha.

Khi ép kiểu một instance của lớp cha sang biến đối tượng của lớp con **(downcasting)** bạn phải tự đảm bảo rằng instance của lớp cha là một instance của lớp con, nếu không sẽ phát sinh lỗi ClassCastException. Downcasting được thực hiện bằng Explicit casting

Ví dụ lỗi ClassCastException

EXAMPLE

```
public class Main {
    public static void main(String[] args) {
        Person p1 = new Student(); // upcasting
        Student s1 = (Student)p1; // Downcasting
        Person p2 = new Student();
        Staff s2 = (Staff)p2; // ClassCastException
    }
}
```

OUTPUT

```
Exception in thread "main"
java.lang.ClassCastException:
Polymophism.Student cannot be
cast to Polymophism.Staff
```

3. Ép kiểu đối tượng và toán tử instanceof:

Sử dụng toán tử instanceof để tránh phát sinh lỗi:

EXAMPLE

```
public class Main {
    public static void main(String[] args) {
        Person p1 = new Student(); // upcasting
        if(p1 instanceof Student){
            System.out.println("OK1");
            Student s1 = (Student)p1; // Downcasting
        }
        else{
            System.out.println("Error");
        }
        Person p2 = new Student();
        if(p2 instanceof Staff){
            System.out.println("OK2");
            Staff s2 = (Staff)p2; // ClassCastException
        }
        else{
            System.out.println("Error");
        }
    }
}
```

OUTPUT

```
OK1
Error
```

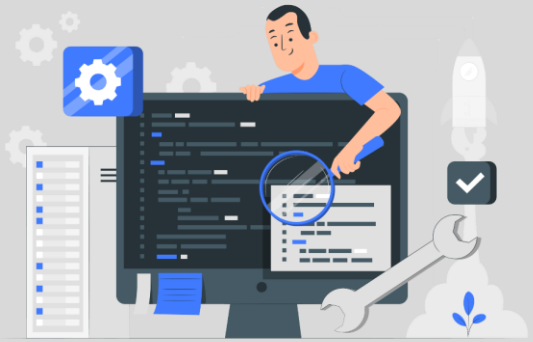
4. Lớp trừu tượng - Abstract Class:

Abstract class không thể sử dụng để tạo ra một đối tượng như các lớp thông thường, lớp trừu tượng có thể chứa các phương thức trừu tượng (abstract method) và sẽ được cài đặt chi tiết hơn ở các lớp con.

Khi một lớp con kế thừa từ 1 lớp trừu tượng, tất cả các phương thức trừu tượng ở lớp cha phải được cài đặt cụ thể ở lớp con.

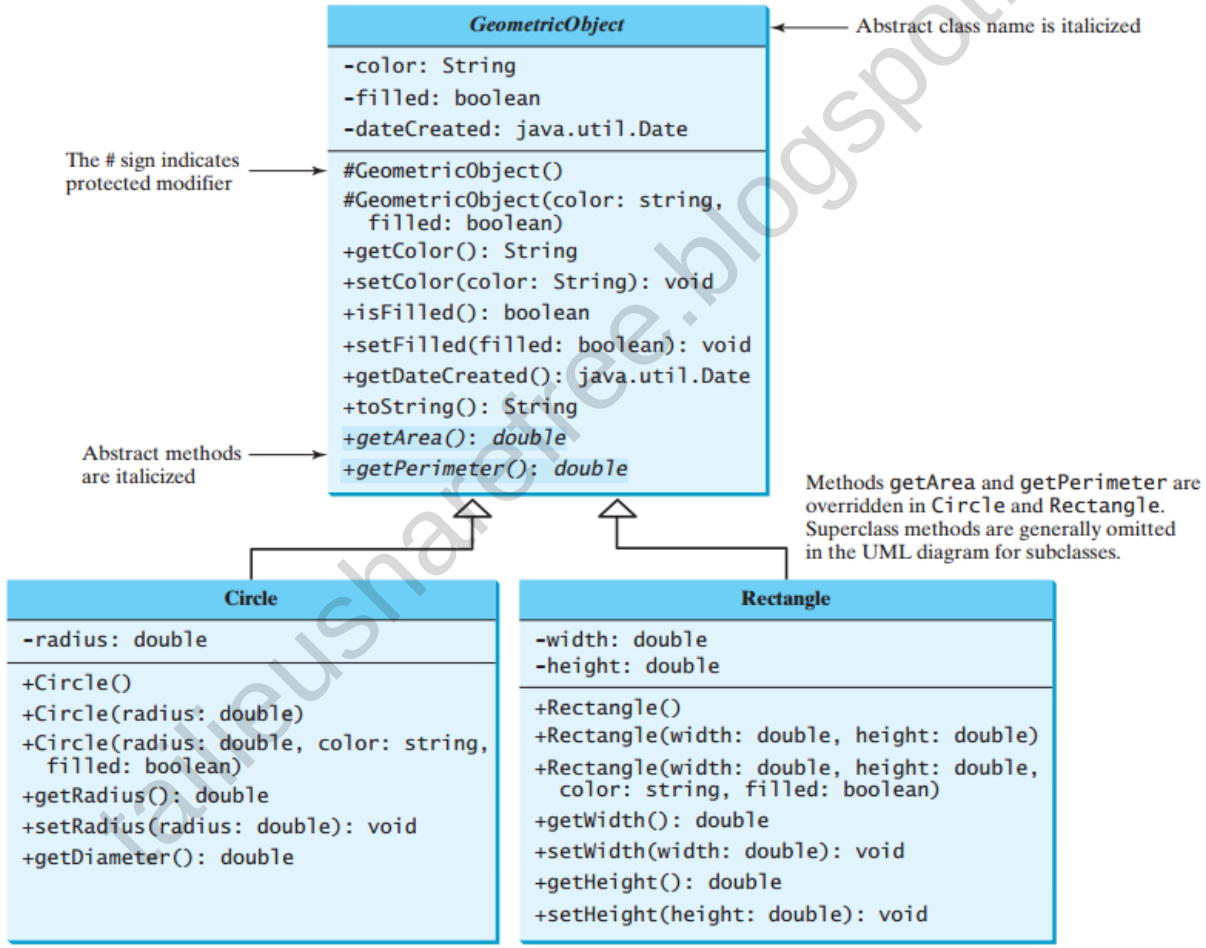
Ví dụ:

Lớp GeometricObject là lớp trừu tượng, có 2 lớp kế thừa từ nó là Circle và Rectangle, trong trường hợp này 2 phương thức getArea() và getPerimeter() không thể cài đặt ở lớp cha vì nó còn phụ thuộc vào loại hình (tam giác, vuông, tròn...) vì thế 2 phương thức này được khai báo là trừu tượng.



4. Lớp trừu tượng - Abstract Class:

Trong thiết kế lớp với UML lớp trừu tượng và phương thức trừu tượng được in nghiêng.



4. Lớp trừu tượng - Abstract Class:

Lớp GeometricObject

```
import java.util.Date;

public abstract class GeometricObject {
    private String color;
    private boolean filled;
    private java.util.Date dateCreated;

    protected GeometricObject(){}
    protected GeometricObject(String color, boolean filled){
        this.color = color;
        this.filled = filled;
        this.dateCreated = new Date(System.currentTimeMillis());
    }
    public abstract double getArea();
    public abstract double getPerimeter();
}
```

4. Lớp trừu tượng - Abstract Class:

Lớp Circle

```
public class Circle extends GeometricObject{
    private double radius;

    public Circle(String color, boolean filled, double radius){
        super(color, filled);
        this.radius = radius;
    }
    @Override
    public double getArea(){
        return Math.PI * radius * radius;
    }
    @Override
    public double getPerimeter(){
        return 2 * Math.PI * radius;
    }
}
```

4. Lớp trừu tượng - Abstract Class:

Lớp Rectangle

```
public class Rectangle extends GeometricObject{
    private double width, height;
    public Rectangle(double width, double height, String color, boolean filled){
        super(color, filled);
        this.width = width;
        this.height = height;
    }
    @Override
    public double getArea() {
        return this.width * this.height;
    }
    @Override
    public double getPerimeter() {
        return 2 * (this.width + this.height);
    }
}
```

4. Lớp trừu tượng - Abstract Class:

Ví dụ:

EXAMPLE

```
public class Main {

    public static boolean equalArea(GeometricObject o1, GeometricObject o2){
        return o1.getArea() == o2.getArea();
    }

    public static void display(GeometricObject o){
        System.out.println(o.getArea() + " " + o.getPerimeter());
    }
    public static void main(String[] args) {
        GeometricObject ob1 = new Circle("Red", true, 10);
        GeometricObject ob2 = new Rectangle(10, 20, "Green", true);
        System.out.println(equalArea(ob1, ob2));
        display(ob1); display(ob2);
    }
}
```

OUTPUT

```
false
314.15 62.83
200.0 60.0
```

4. Lớp trừu tượng - Abstract Class:

- Không thể tạo đối tượng từ lớp trừu tượng thông qua toán tử new, tuy nhiên bạn vẫn có thể tạo constructor cho lớp trừu tượng.
- Lớp chứa phương thức trừu tượng bắt buộc phải là một lớp trừu tượng, tuy nhiên một lớp trừu tượng có thể không chứa phương thức trừu tượng.
- Lớp cha của một lớp trừu tượng có thể không phải là một lớp trừu tượng, ví dụ lớp Object là cha của lớp GeometicObject

Lưu ý về lớp trừu tượng

4. Lớp trừu tượng - Abstract Class:

- Một phương thức trừu tượng chỉ có thể được chứa trong một lớp trừu tượng, nếu một lớp con không cài đặt mọi phương thức trừu tượng ở lớp cha thì nó phải được khai báo là một lớp trừu tượng. Các phương thức trừu tượng không thể khai báo với từ khóa static.

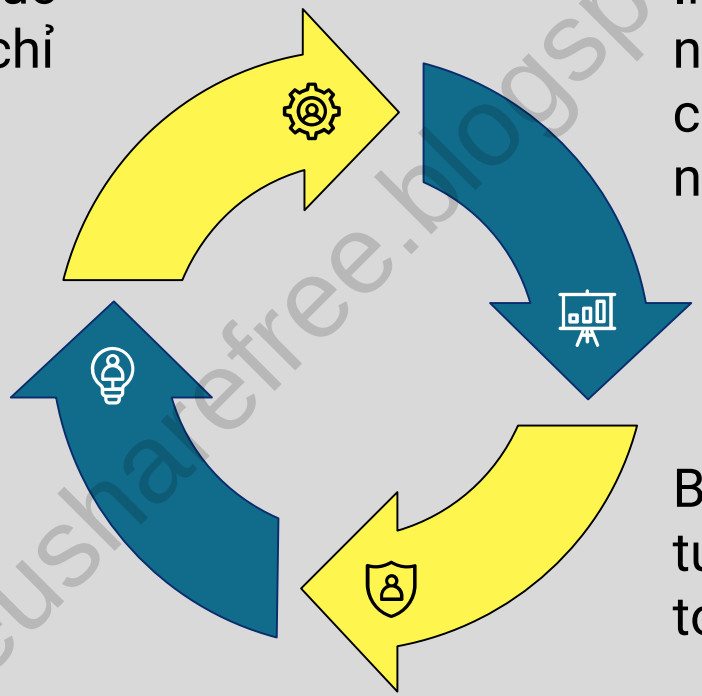
- Bạn không thể tạo đối tượng từ lớp trừu tượng nhưng có thể sử dụng nó như một kiểu dữ liệu.

Lưu ý về lớp trừu tượng

5. Interface:

Một số đặc điểm:

Interface trong Java có cấu trúc tương tự như một lớp nhưng chỉ chứa các hằng số và phương thức trừu tượng



Interface có vai trò tạo ra những phương thức, hành vi chung cho các lớp tương tự như Lớp trừu tượng.

Mối quan hệ giữa lớp và interface được gọi là kế thừa interface, để một lớp kế thừa interface ta sử dụng từ khóa implements

Bạn không thể tạo ra một đối tượng từ interface thông qua toán tử new.

5. Interface:

Cú pháp khai báo Interface:

```
modifier interface InterfaceName{
    //constant
    //abstract method
}

public class X implements InterfaceName{

}
```

Ví dụ:

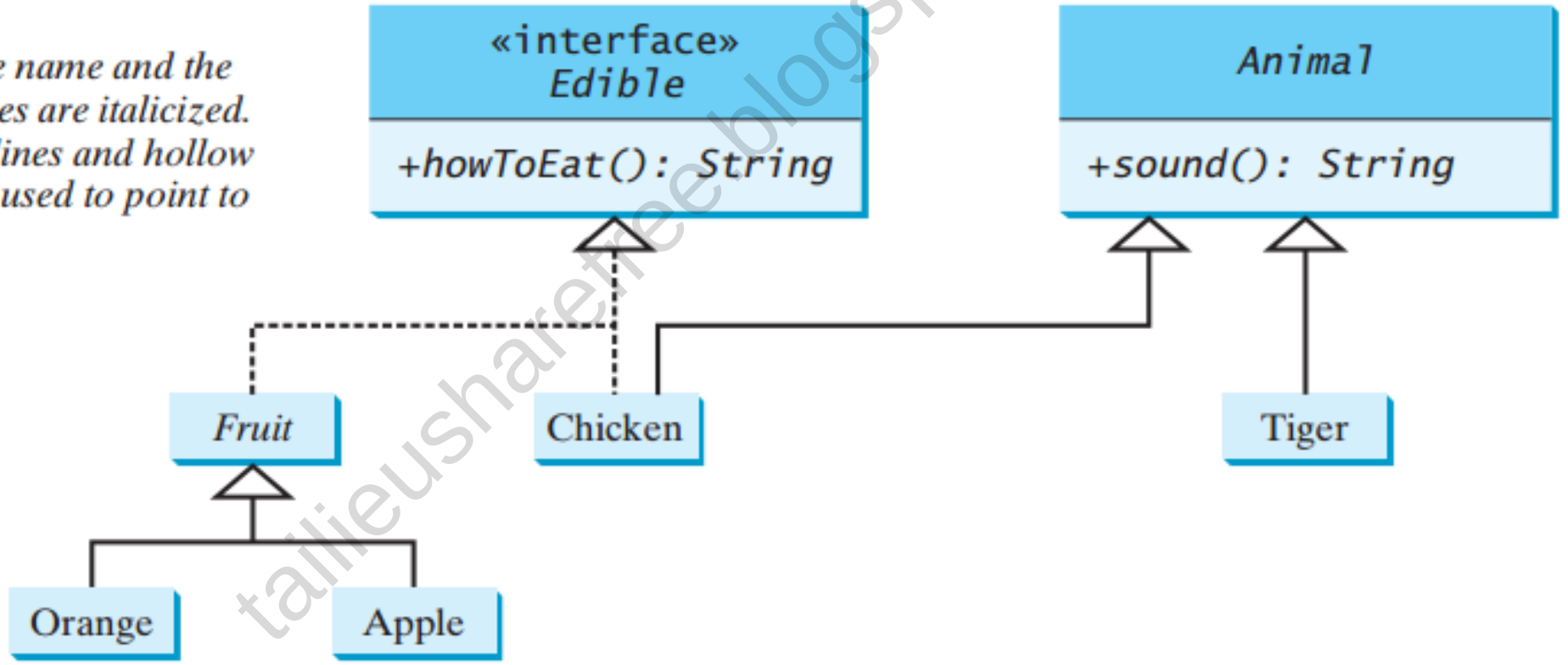
```
public interface Edible {
    public abstract String howToEat();
}

public Dog implements Edible{
    public String howToEat(){
        System.out.println("Dog eat");
    }
}
```

5. Interface:

Khi một lớp cài đặt một interface thì mối quan hệ này được biểu diễn bởi mũi tên nét đứt từ lớp tới interface.

Notation:
The interface name and the method names are italicized. The dashed lines and hollow triangles are used to point to the interface.



5. Interface:

Ví dụ:


```
public interface Drawable {
    public abstract void draw();
}
public class Circle implements Drawable{
    @Override
    public void draw() {
        System.out.println("Drawing circle ");
    }
}
public class Rectangle implements Drawable{
    @Override
    public void draw() {
        System.out.println("Drawing rectangle");
    }
}
```

```
public class Main {
    public static void main(String[] args){
        Circle c = new Circle();
        c.draw();
        Rectangle r = new Rectangle();
        r.draw();
    }
}
```

OUTPUT

```
Drawing circle
Drawing rectangle
```

6. Đa kế thừa trong Java:

Trong Java không hỗ trợ đa kế thừa thông qua lớp nhưng có thể thông qua interface, một lớp có thể cài đặt được nhiều interface.

Ví dụ:


```
public interface Flyable {
    public abstract void fly();
}

public interface Runable {
    public abstract void run();
}
```

```
public class Bird implements Runable, Flyable{

    @Override
    public void run() {
        System.out.println("Bird can run");
    }

    @Override
    public void fly() {
        System.out.println("Bird can fly");
    }
}
```

7. Comparable Interface:

Comparable Interface định nghĩa một phương thức là **compareTo** để so sánh object. Giả sử bạn cần một phương thức chung để tìm đối tượng lớn hơn trong 2 Sinh viên, 2 hình tròn... để có thể so sánh 2 đối tượng thì chúng cần phải so sánh được với nhau, Java hỗ trợ **interface comparable** giúp bạn đạt được điều này.

Định nghĩa:

```
package java.lang;
public interface Comparable<E> {
    public int compareTo(E o);
}
```

7. Comparable Interface:

Lớp sinh viên cài đặt Comparable interface

EXAMPLE

```
public class Student implements Comparable<Student>{
    private String name;
    private double gpa;

    public Student(String name, double gpa) {
        this.name = name;
        this.gpa = gpa;
    }
    public double getGpa(){
        return this.gpa;
    }
    @Override
    public int compareTo(Student o) {
        if(this.gpa < o.getGpa()) return -1;
        else if(this.gpa > o.getGpa()) return 1;
        else return 0;
    }
}
```

7. Comparable Interface:

Drive Class 1:

EXAMPLE

```
public class Main {
    public static void main(String[] args) {
        ArrayList<Student> a = new ArrayList<>();
        a.add(new Student("Teo", 3.1));
        a.add(new Student("Nam", 3.4));
        a.add(new Student("Ty", 2.7));
        a.add(new Student("Phuong", 2.1));
        Collections.sort(a);
        for(Student x : a){
            System.out.println(x.getName() + " " + x.getGpa());
        }
    }
}
```

OUTPUT

```
Phuong 2.1
Ty 2.7
Teo 3.1
Nam 3.4
```

7. Comparable Interface:

Drive Class 2:

EXAMPLE

```
public class Main {
    public static void main(String[] args) {
        ArrayList<Student> a = new ArrayList<>();
        a.add(new Student("Teo", 3.1));
        a.add(new Student("Nam", 3.4));
        a.add(new Student("Ty", 2.7));
        a.add(new Student("Phuong", 2.1));
        Collections.sort(a, Collections.reverseOrder());
        for(Student x : a){
            System.out.println(x.getName() + " " + x.getGpa());
        }
    }
}
```

OUTPUT

```
Nam 3.4
Teo 3.1
Ty 2.7
Phuong 2.1
```

8. Comparator Interface:

Bên cạnh comparable interface Java còn cung cấp cho bạn interface là **comparator** để có thể dễ dàng sắp xếp danh sách các object mà người dùng tự định nghĩa ra.

Lớp sinh viên:

```
public class Student{
    private String name;
    private double gpa;

    public Student(String name, double gpa) {
        this.name = name;
        this.gpa = gpa;
    }
    public double getGpa(){
        return this.gpa;
    }
    public String getName(){
        return this.name;
    }
}
```

8. Comparator Interface:

Lớp SortStudentByName implement comparator interface

```
public class SortStudentByName implements Comparator<Student>{
    @Override
    public int compare(Student o1, Student o2) {
        return o1.getName().compareTo(o2.getName());
    }
}
public class Main {
    public static void main(String[] args) {
        ArrayList<Student> a = new ArrayList<>();
        a.add(new Student("Teo", 3.1));
        a.add(new Student("Nam", 3.4));
        a.add(new Student("Ty", 2.7));
        a.add(new Student("Phuong", 2.1));
        Collections.sort(a, new SortStudentByName());
        for(Student x : a){
            System.out.println(x.getName() + " " + x.getGpa());
        }
    }
}
```

OUTPUT

```
Nam 3.4
Phuong 2.1
Teo 3.1
Ty 2.7
```