Investigar diferencias entre clases, clases asbtractas e interfaces (Un ejemplo de cada una)

**Diferencias entre Clases, Clases Abstractas e Interfaces en Java**

En la programación orientada a objetos (OOP), las clases, las clases abstractas y las interfaces son pilares fundamentales que permiten organizar y estructurar el código de manera eficiente. Aunque estos conceptos pueden parecer similares, cada uno tiene un propósito específico que se adapta a diferentes necesidades de diseño y desarrollo. A continuación, se detallan las diferencias clave entre ellos, acompañadas de ejemplos prácticos.

**1. Clases**

Una **clase** en Java es una plantilla o modelo que define un conjunto de atributos y métodos que los objetos creados a partir de esa clase pueden tener. Las clases permiten encapsular datos y comportamientos, facilitando la creación de objetos con características comunes.

**Características**:

* **Instanciación**: Se pueden crear objetos directamente a partir de una clase.
* **Métodos y Atributos**: Puede contener tanto métodos como atributos.
* **Herencia**: Una clase puede heredar de otra clase (herencia simple)

class Animal {

String nombre;

public Animal(String nombre) {

this.nombre = nombre;

}

public void hacerSonido() {

System.out.println(nombre + " hace un sonido.");

}

}

public class EjemploClase {

public static void main(String[] args) {

Animal perro = new Animal("Perro");

perro.hacerSonido(); // Salida: Perro hace un sonido.

}

}

**2. Clases Abstractas**

Una **clase abstracta** es una clase que no se puede instanciar directamente, es decir, no se pueden crear objetos a partir de ella. Se utiliza como una plantilla para crear subclases que deben implementar los métodos abstractos que la clase abstracta define. Además, puede contener métodos concretos que pueden ser heredados por las subclases.

**Características**:

* **Instanciación**: No se pueden crear objetos directamente; es necesario extenderla.
* **Métodos Abstractos y Concretos**: Puede contener métodos abstractos (sin implementación) y métodos concretos (con implementación).
* **Herencia**: Facilita la herencia, obligando a las subclases a implementar ciertos métodos.

**Ejemplo**:

abstract class FiguraGeometrica {

private String nombre;

public FiguraGeometrica(String nombre) {

this.nombre = nombre;

}

public abstract double calcularArea();

public String getNombre() {

return nombre;

}

}

class Circulo extends FiguraGeometrica {

private double radio;

public Circulo(String nombre, double radio) {

super(nombre);

this.radio = radio;

}

@Override

public double calcularArea() {

return Math.PI \* radio \* radio;

}

}

class Rectangulo extends FiguraGeometrica {

private double longitud;

private double anchura;

public Rectangulo(String nombre, double longitud, double anchura) {

super(nombre);

this.longitud = longitud;

this.anchura = anchura;

}

@Override

public double calcularArea() {

return longitud \* anchura;

}

}

public class EjemploClaseAbstracta {

public static void main(String[] args) {

Circulo circulo = new Circulo("Círculo", 5.0);

Rectangulo rectangulo = new Rectangulo("Rectángulo", 4.0, 6.0);

System.out.println("Área de " + circulo.getNombre() + ": " + circulo.calcularArea());

System.out.println("Área de " + rectangulo.getNombre() + ": " + rectangulo.calcularArea());

}

}

**3. Interfaces**

Una **interfaz** es un contrato que define un conjunto de métodos que una clase debe implementar. Las interfaces no pueden contener implementaciones de métodos; solo pueden declarar métodos abstractos. Además, una clase puede implementar múltiples interfaces, lo que permite una mayor flexibilidad en la arquitectura del software.

**Características**:

* **Instanciación**: No se puede instanciar; se implementa en clases concretas.
* **Métodos**: Solo puede contener declaraciones de métodos (a partir de Java 8, pueden contener métodos por defecto con implementación).
* **Herencia Múltiple**: Una clase puede implementar múltiples interfaces, permitiendo la herencia múltiple de comportamientos.

**Ejemplo**:

interface ReproductorMultimedia {

void reproducir();

void pausar();

void detener();

void adelantar(int segundos);

void retroceder(int segundos);

}

class ReproductorAudio implements ReproductorMultimedia {

private String cancion;

public ReproductorAudio(String cancion) {

this.cancion = cancion;

}

@Override

public void reproducir() {

System.out.println("Reproduciendo audio: " + cancion);

}

@Override

public void pausar() {

System.out.println("Audio en pausa.");

}

@Override

public void detener() {

System.out.println("Audio detenido.");

}

@Override

public void adelantar(int segundos) {

System.out.println("Adelantando audio " + segundos + " segundos.");

}

@Override

public void retroceder(int segundos) {

System.out.println("Retrocediendo audio " + segundos + " segundos.");

}

}

class ReproductorVideo implements ReproductorMultimedia {

private String video;

public ReproductorVideo(String video) {

this.video = video;

}

@Override

public void reproducir() {

System.out.println("Reproduciendo video: " + video);

}

@Override

public void pausar() {

System.out.println("Video en pausa.");

}

@Override

public void detener() {

System.out.println("Video detenido.");

}

@Override

public void adelantar(int segundos) {

System.out.println("Adelantando video " + segundos + " segundos.");

}

@Override

public void retroceder(int segundos) {

System.out.println("Retrocediendo video " + segundos + " segundos.");

}

}

public class EjemploInterfacesMultimedia {

public static void main(String[] args) {

ReproductorMultimedia reproductorAudio = new ReproductorAudio("Canción1.mp3");

ReproductorMultimedia reproductorVideo = new ReproductorVideo("Video1.mp4");

reproductorAudio.reproducir();

reproductorAudio.adelantar(30);

reproductorAudio.pausar();

reproductorVideo.reproducir();

reproductorVideo.detener();

}

}