Día 5 – Fundamentos de Programación Orientada a Objetos Teoría: Clases y objetos. Encapsulamiento, herencia, polimorfismo, abstracción. Práctica: Crear una clase Animal y subclases Perro y Gato. Ejercicio clave: Usar herencia para reutilizar código.

🧠 Teoría – Fundamentos de POO

◇ Clase y Objeto

- **Clase**: plantilla o molde (define atributos y comportamientos).
- **Objeto**: instancia de una clase (elemento real con esos atributos/comportamientos).

◇ Principios de POO

1. **Encapsulamiento**
    a. Ocultar los detalles internos.
    b. Se logra con modificadores de acceso (`private`, `public`) y getters/setters.
2. **Herencia**
    a. Una clase hereda atributos y métodos de otra.
    b. Uso de `extends`.
3. **Polimorfismo**
    a. Un mismo método puede tener diferentes comportamientos según el objeto.
    b. Se logra por sobreescritura (`@Override`) y sobrecarga.
4. **Abstracción**
    a. Enfocarse solo en lo importante; ocultar lo complejo.
    b. Se logra con clases abstractas o interfaces.

🐾 Práctica – Animal, Perro y Gato

1. Clase `Animal`

```
public class Animal {

    protected String nombre;


    public Animal(String nombre) {
```

```
        this.nombre = nombre;
    }

    public void hacerSonido() {
        System.out.println("El animal hace un sonido");
    }

    public String getNombre() {
        return nombre;
    }
}
```

2. Subclase Perro

```
public class Perro extends Animal {
    public Perro(String nombre) {
        super(nombre);
    }

    @Override
    public void hacerSonido() {
        System.out.println(nombre + " dice: ¡Guau!");
    }
}
```

3. Subclase Gato

```
public class Gato extends Animal {
    public Gato(String nombre) {
```

```
        super(nombre);
    }

    @Override
    public void hacerSonido() {
        System.out.println(nombre + " dice: ¡Miau!");
    }
}
```

4. Clase Main

```
public class Main {
    public static void main(String[] args) {
        Animal miPerro = new Perro("Firulais");
        Animal miGato = new Gato("Michi");

        miPerro.hacerSonido(); // Firulais dice: ¡Guau!
        miGato.hacerSonido();  // Michi dice: ¡Miau!
    }
}
```

⭐ Ejercicio clave: Reutilización de código con herencia

La clase Animal encapsula los atributos y comportamientos comunes. Las subclases Perro y Gato **heredan** de Animal y personalizan el comportamiento (hacerSonido()), evitando repetir el código de nombre y constructor.

Día 6 – POO Avanzado y Práctica Mixta Teoría: Métodos estáticos, clases abstractas, interfaces. Principios SOLID (introductorio). Práctica: Implementar una interfaz y múltiples clases que la usen. Ejercicio clave: Simular un sistema de transporte con clases abstractas.

🧠 Teoría

◇ Métodos estáticos

- Se definen con la palabra clave `static`.
- No requieren instancia de la clase para usarse.
- Ejemplo: `Math.sqrt()`, `System.out.println()`.

```
public class Util {
    public static int sumar(int a, int b) {
        return a + b;
    }
}
```

◇ Clase abstracta

- No se puede instanciar.
- Puede tener métodos implementados y abstractos (sin cuerpo).
- Se usa para definir una estructura común.

```
public abstract class Transporte {
    protected String marca;

    public Transporte(String marca) {
        this.marca = marca;
    }

    public abstract void moverse();

    public void mostrarMarca() {
        System.out.println("Marca: " + marca);
    }
}
```

◇ Interfaz

- Define un contrato (métodos que deben implementarse).
- No tiene estado (no guarda datos) y sus métodos son por defecto `abstract` y `public`.

```
public interface Conducible {
    void conducir();
}
```

◇ Principios SOLID (introductorio)

1. **S** - *Single Responsibility Principle*: una clase debe tener una única responsabilidad.
2. **O** - *Open/Closed Principle*: abierto a extensión, cerrado a modificación.
3. **L** - *Liskov Substitution Principle*: una subclase debe poder reemplazar a su superclase sin alterar el programa.
4. **I** - *Interface Segregation Principle*: no forzar a implementar métodos que no necesita.
5. **D** - *Dependency Inversion Principle*: depender de abstracciones, no de clases concretas.

🛠️ Práctica – Interfaz + Clases

Interfaz `Conducible`

```
public interface Conducible {
    void conducir();
}
```

Clase `Auto`

```
public class Auto implements Conducible {
    public void conducir() {
        System.out.println("El auto está conduciendo en la carretera.");
    }
}
```

Clase `Bicicleta`

```java
public class Bicicleta implements Conducible {
    public void conducir() {
        System.out.println("La bicicleta se mueve por el carril bici.");
    }
}
```

⭐ Ejercicio clave: Sistema de transporte con clase abstracta

Clase abstracta `Transporte`

```java
public abstract class Transporte {
    protected String tipo;

    public Transporte(String tipo) {
        this.tipo = tipo;
    }

    public abstract void moverse();

    public void mostrarTipo() {
        System.out.println("Tipo de transporte: " + tipo);
    }
}
```

Subclases

```java
public class Tren extends Transporte {
    public Tren() {
        super("Tren");
    }

    @Override
    public void moverse() {
        System.out.println("El tren se mueve sobre rieles.");
    }
}

public class Avion extends Transporte {
    public Avion() {
        super("Avión");
    }

    @Override
    public void moverse() {
        System.out.println("El avión vuela por el aire.");
    }
}
```

Clase Principal

```java
public class Main {
    public static void main(String[] args) {
        Transporte t1 = new Tren();
        Transporte t2 = new Avion();

        t1.moverse();
        t2.moverse();

        Conducible c1 = new Auto();
        Conducible c2 = new Bicicleta();

        c1.conducir();
        c2.conducir();
    }
}
```

Esto mezcla **clases abstractas** (para estructura base) e **interfaces** (para comportamientos independientes), una combinación muy usada en sistemas reales.