UNIVERSIDAD TECNOLÓGICA DEL PERÚ

Facultad de Ingeniería

Programación Orientada a Objetos

Sesión 3:

Relaciones de asociación, agregación y composición

UNIVERSIDAD TECNOLÓGICA DEL PERÚ

Facultad de Ingeniería

Programación Orientada a Objetos

Indicador de Logro

Al finalizar la sesión, el estudiante demuestra la capacidad de explicar claramente las relaciones entre clases en un modelo orientado a objetos, identificando correctamente ejemplos de asociación, agregación y composición,en la solución de problemas usando Java.

UNIVERSIDAD TECNOLÓGICA DEL PERÚ

Facultad de Ingeniería

Programación Orientada a Objetos

Utilidad

Un **diagrama de clases** es importante porque nos permitirá representar gráficamente y de manera estática la estructura general de un sistema, mostrando cada una de las clases y sus interacciones (como herencias, asociaciones, agregaciones, composiciones). Los **diagramas de clases** son el pilar fundamental del *modelado con UML*, siendo ampliamente utilizados tanto para *análisis como para diseño de sistemas y software en general*.

UNIVERSIDAD TECNOLÓGICA DEL PERÚ

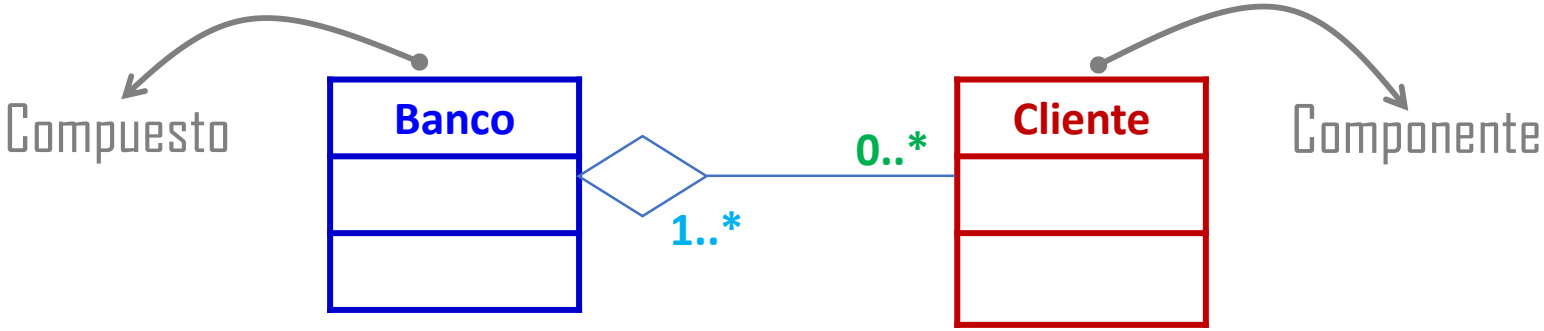
Facultad de Ingeniería

Programación Orientada a Objetos

RECORDANDO LA CLASE ANTERIOR

- **clases y objetos**
- **constructores**

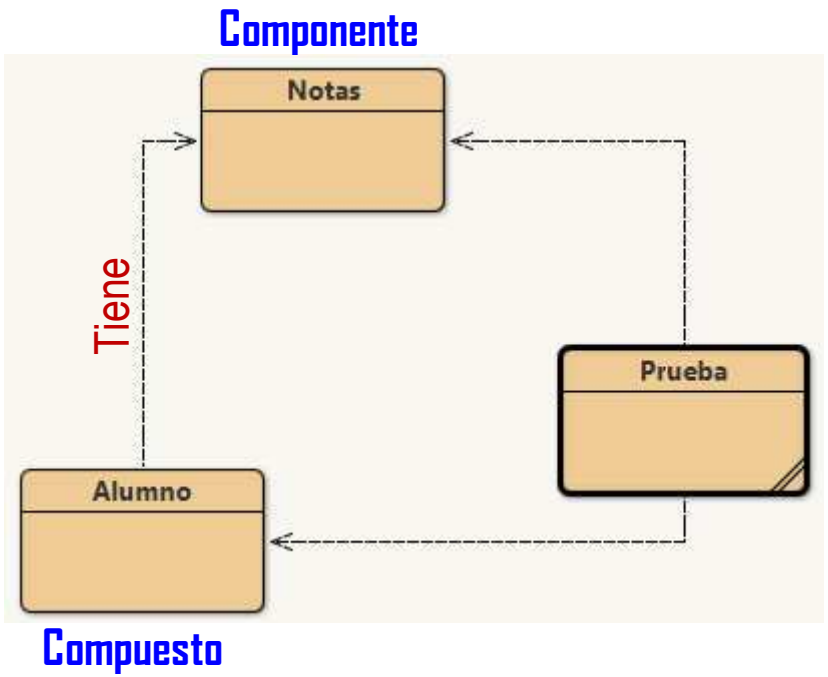
Agregación en JAVA



- Los *componentes* pueden ser compartidos por varios *compuestos*.
- La destrucción del compuesto **no** conlleva la destrucción de los componentes.
- Se lee: “Un Banco tiene cualquier número de Clientes”.
- Se lee: “Un cliente pertenece a por lo menos un Banco”.

Ejemplo01: Relación de Agregación

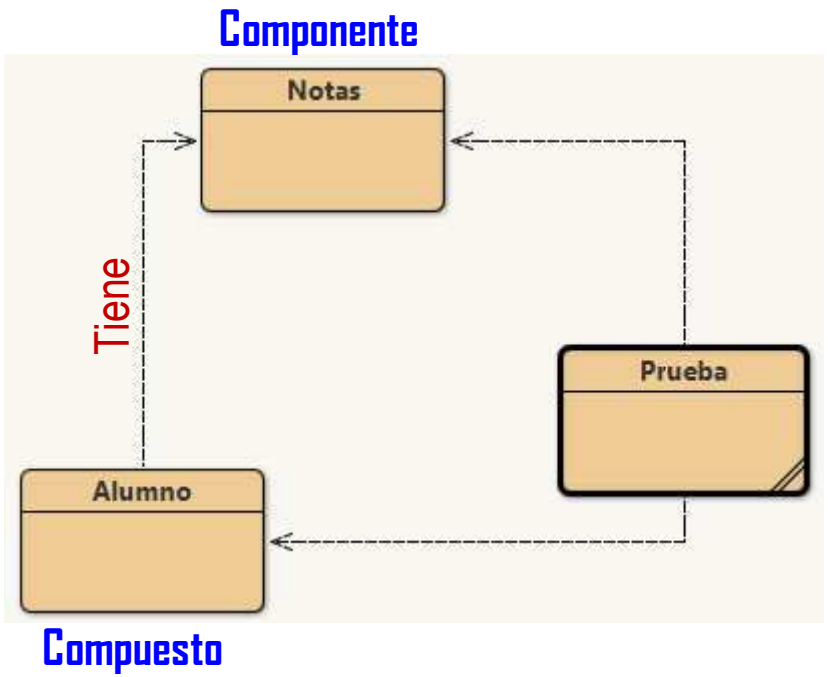
Implementar el siguiente diagrama de clases en BlueJ:




```java
public class Notas
{
    private int poo;
    private int estructura;
    private int algoritmos;

    public Notas(int poo, int estructura, int algoritmos){
      this.poo=poo;
      this.estructura=estructura;
      this.algoritmos=algoritmos;
    }

    public int getNotaPoo(){return poo;}
    public int getNotaEstructura(){return estructura;}
    public int getNotaAlgoritmos(){return algoritmos;}
}
```

Ejemplo01: Relación de Agregación

Implementar el siguiente diagrama de clases en BlueJ:

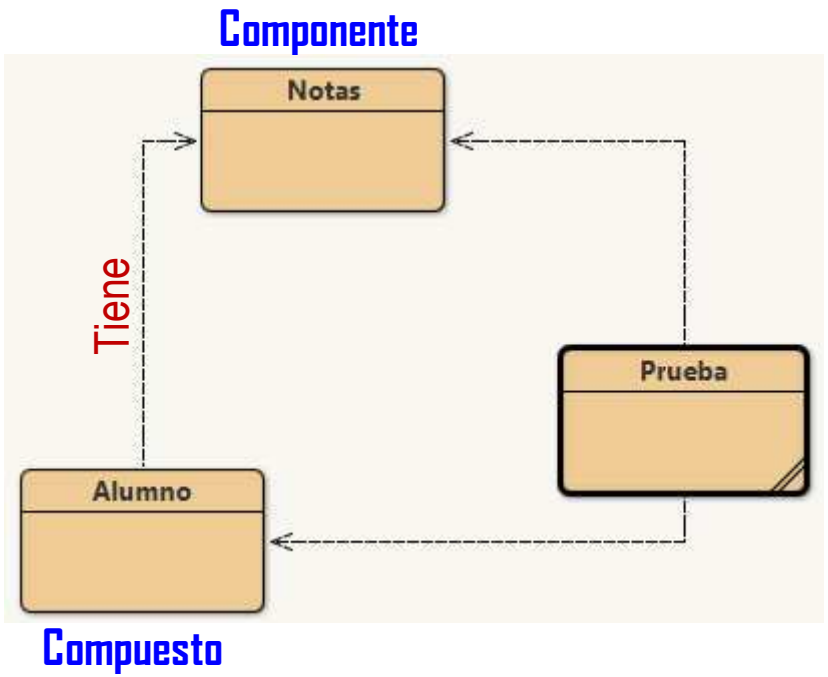



```java
public class Alumno
{
    private String id;
    private String nombre;
    private Notas notas;

    public Alumno(String id, String nombre, Notas notas){
        this.id = id;
        this.nombre = nombre;
        this.notas = notas;
    }

    public void cambiarNotas(Notas notas){this.notas=notas;}

    public void mostrar(){
      System.out.println("\nId:"+id);
      System.out.println("Nombre:"+nombre);
      System.out.println("Nota en POO:"+notas.getNotaPoo());
      System.out.println("Nota en Estructura:"+notas.getNotaEstructura());
      System.out.println("Nota en Algoritmos:"+notas.getNotaAlgoritmos());
    }
}
```

El mismo **conjunto de notas** lo puedo enviar a diferentes alumnos, eso significa que **el mismo componente puede ser compartido por diferentes compuestos** y esa una característica de la **agregación**

Ejemplo01: Relación de Agregación

Implementar el siguiente diagrama de clases en BlueJ:



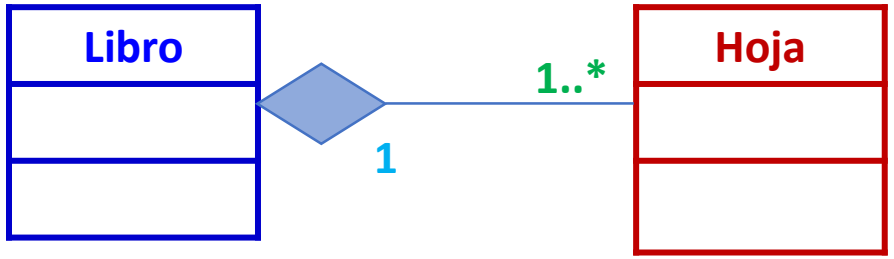

```
public class Prueba
{
    public static void main(){
        Notas notas = new Notas(12,15,20);
        Alumno a1 = new Alumno("A001","Roy",notas);
        a1.mostrar();


        //a1=null;
        //System.out.println("\n"+notas.getNotaPoo());
        //System.out.println(notas.getNotaEstructura());
        //System.out.println(notas.getNotaAlgoritmos());
    }
}
```

La destrucción del compuesto **no** conlleva la destrucción de los componentes.

Sigue existiendo el objeto notas con sus datos: 12, 15 y 20

Composición en JAVA

La Composición nos dice "***es parte de***".



- Se lee: "Un Libro está compuesto por una o varias Hojas".
- Se lee: "Una Hoja es parte de un Libro". (solamente de un Libro)
- El tiempo de vida de los objetos Hoja depende del tiempo de vida de Libro, ya que si no existe un Libro no pueden existir sus Hojas.

Composición en JAVA

Ventajas de la Composición:

1.Flexibilidad: Puedes cambiar fácilmente el comportamiento de una clase cambiando los objetos con los que se compone.

2.Reutilización: Las clases individuales pueden ser reutilizadas en múltiples composiciones.

3.Evita problemas de herencia múltiple: Java no admite herencia múltiple debido a problemas como el "diamante". La composición es una forma de obtener funcionalidad de múltiples fuentes sin heredar de múltiples clases.

Ejemplo de Composición en Java:

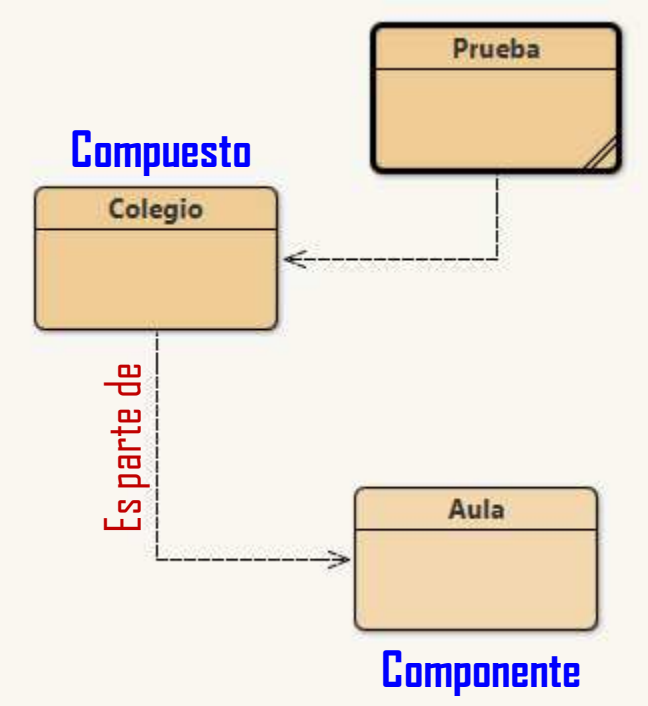
Supongamos que tenemos una clase Motor y una clase Carro. En lugar de decir que un Carro es un Motor, decimos que un Carro tiene un Motor.

En el ejemplo, la clase **Carro** no hereda de la clase **Motor**, sino que tiene una instancia de **Motor**. Esto es un ejemplo de composición.

```
class Motor {
    void encender() {
        System.out.println("El motor está encendido");
    }

    void apagar() {
        System.out.println("El motor está apagado");
    }
}

class Carro {
    // Composición: Carro "tiene-un" Motor
    private Motor motor;

    public Carro() {
        this.motor = new Motor();
    }

    void encenderMotor() {
        motor.encender();
    }

    void apagarMotor() {
        motor.apagar();
    }
}

public class Main {
    public static void main(String[] args) {
        Carro carro = new Carro();
        carro.encenderMotor();
        carro.apagarMotor();
    }
}
```

Ejemplo02: Relación de Composición

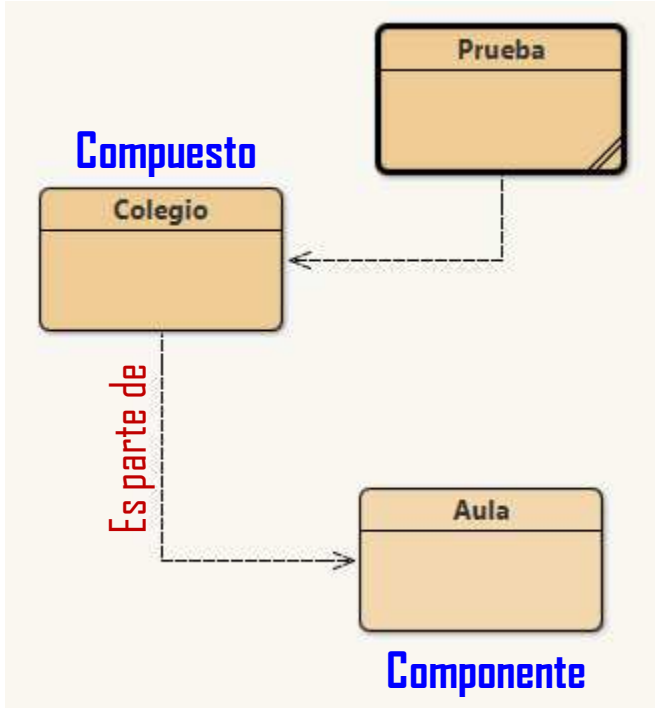
Implementar el siguiente diagrama de clases en BlueJ:




```java
public class Aula
{
    private String nombre;
    private int aforo;

    public Aula(String nombre, int aforo){
        this.nombre=nombre;
        this.aforo=aforo;
    }

    public String getNombre(){return nombre;}
    public int getAforo(){return aforo;}

}
```

Ejemplo02: Relación de Composición

Implementar el siguiente diagrama de clases en BlueJ:



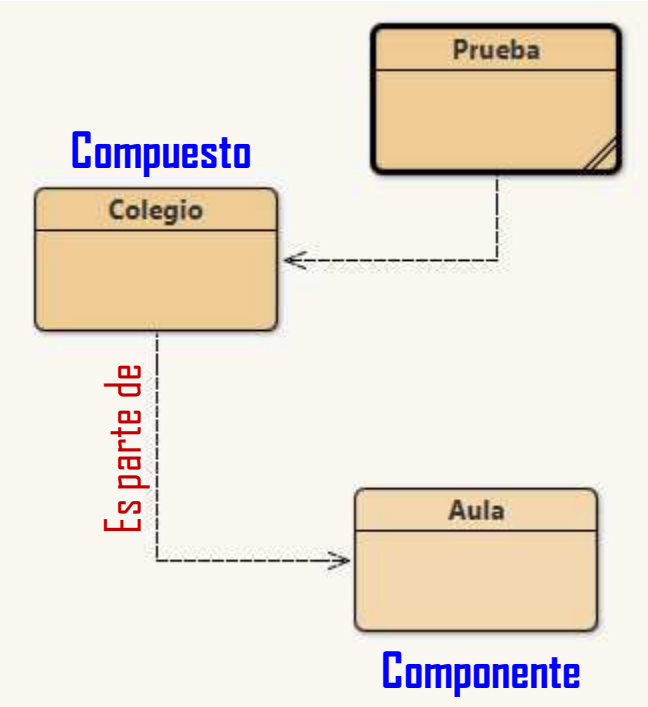

```
import java.util.Scanner;
public class Colegio
{
    private Aula[] aulas;

    public Colegio(){

    public void cargar_aulas(){
       Scanner obj = new Scanner(System.in);
       for(int i=0;i<aulas.length;i++){
          System.out.print("\nIngrese aula [Pab-N°]: ");
          String nombreAula = obj.next();
          System.out.print("Ingrese aforo del aula "+nombreAula+": ");
          int aforo = obj.nextInt();
          aulas[i]= new Aula(nombreAula,aforo);
       }
    }

    public void mostrar(){

    public void finalize() {
}
```

Los **componentes** se crean internamente, **no se envían como parámetro en el constructor del compuesto**. En otras palabras, todas las aulas que se creen al ejecutarse este método SERÁN PARTE DEL mismo colegio, no podrán ser compartidas por otro colegio.

Entonces, **cuando se elimine un colegio se eliminarán también sus aulas.**

Ejemplo02: Relación de Composición

Implementar el siguiente diagrama de clases en BlueJ:




```
public class Prueba
{
    public static void main(String args[]){
        Colegio c = new Colegio();
        c.cargar_aulas();
        c.mostrar();

        c=null;
        System.gc();
        c.mostrar();

    }
}
```

La destrucción del compuesto **SI** conlleva la destrucción de los componentes.

Esta última línea genera un error, pues **ya no existe las aulas del colegio c**. Al eliminar el compuesto se han eliminado también sus componentes. (hay que comentarla)

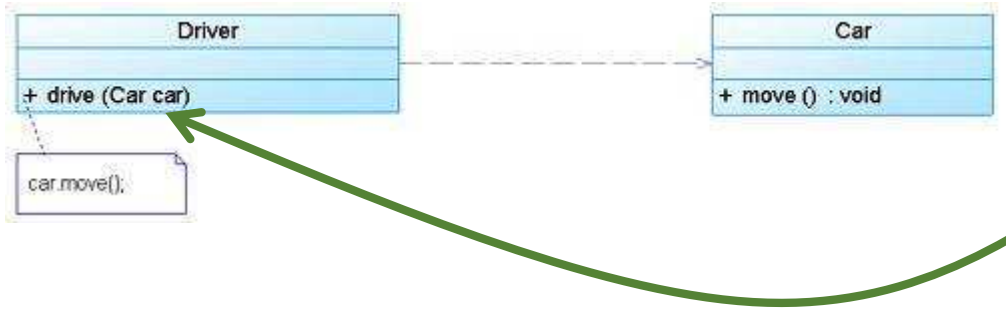
Relación de dependencia entre clases

La dependencia **es una relación de uso**. Muestra la **relación entre un cliente y el proveedor de un servicio** usado por el cliente.

Cliente es el objeto que solicita un servicio.

Servidor es el objeto que provee un servicio.

Gráficamente la dependencia se muestra como una línea discontinua con una punta de flecha que apunta del cliente al proveedor.



En muchos casos, la relación de dependencia se refleja en los métodos de una clase utilizando objetos de otra clase como parámetros.

Relación de dependencia entre clases

Relación de Dependencia: Definición

Una clase A tiene una relación de dependencia con una clase B si:

- Un método de A crea un objeto de B.
- Un método de A usa un objeto de B como parámetro.
- Un método de A devuelve un objeto de B.

La relación de dependencia es más débil que otras relaciones como la asociación, la agregación o la composición. Es una relación de "uso" más que de "tiene".

En este ejemplo, la clase Carro tiene una relación de dependencia con la clase Motor porque el método arrancar de Carro utiliza un objeto de Motor como parámetro.

```
class Motor {
    void encender() {
        System.out.println("El motor está encendido");
    }
}

class Carro {
    void arrancar(Motor motor) {
        motor.encender();
    }
}

public class Main {
    public static void main(String[] args) {
        Carro carro = new Carro();
        Motor motor = new Motor();
        carro.arrancar(motor);
    }
}
```

Ejemplo03: Relación de dependencia

Implementar las siguientes clases en Java:

```
1   package ejemplo4;
2   import java.util.Scanner;
3   public class MiClase
4   {
5       private int n;
6       public MiClase(){
7          pedirN();
8       }
9       public void pedirN(){
14      public void setN(int n){
17      public int getN(){
20      public void par_impar(){
28      public void positivo_negativo(){
39      public void primo_v1(){
58      public void primo_v2(){
73
74  }
```

```
package ejemplo4;
public class Prueba
{
    public static void main(String args[])
    {
        MiClase obj = new MiClase();
        obj.par_impar();
        obj.positivo_negativo();
        obj.primo_v1();
        obj.primo_v2();
        //supongamos que quiero modificar el
        obj.pedirN(); //o utilizo el setN(10)
        obj.par_impar();
        obj.positivo_negativo();
        obj.primo_v1();
        obj.primo_v2();
    }
}
```

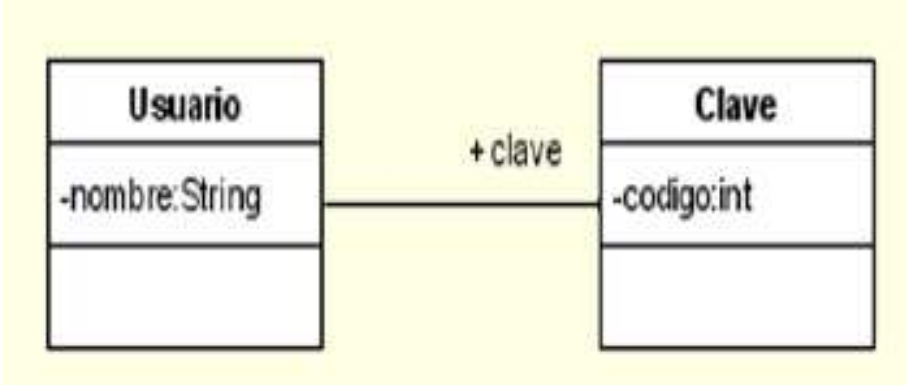
Relación de dependencia entre clases

Asociación

Diremos que dos (o más) clases tiene una relación de **asociación** cuando una de ellas tenga que requerir o utilizar alguno de los servicios (es decir, acceder a alguna de las propiedades o métodos) de las otras.

Como se puede observar, las relaciones de asociación son más débiles que las relaciones de agregación, en el sentido de que no requieren (aunque en ocasiones lo implementemos así) que creemos un objeto nuevo a partir de otros objetos, sino únicamente que los objetos interactúen entre sí.

Las relaciones de asociación crean enlaces entre objetos. Estos enlaces no tienen por qué ser permanentes (es más, en la mayoría de los casos, no lo serán). Los objetos deben tener entidad fuera de la relación (a diferencia de las relaciones de composición).

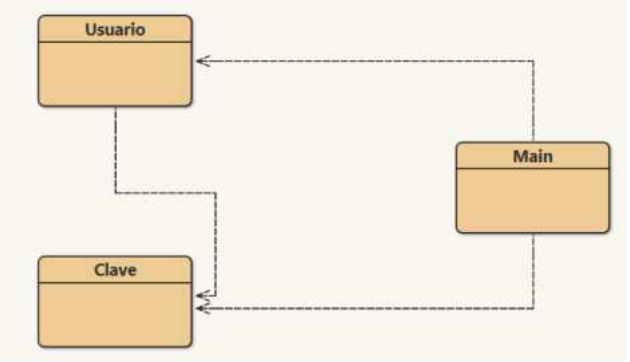


Relación de dependencia entre clases

```
class Usuario {
    private String nombre;
    private Clave clave;

    public Usuario(String nombre, Clave clave) {
        this.nombre = nombre;
        this.clave = clave;
    }

    // Métodos getter y setter para el nombre
    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    // Métodos getter y setter para la clave
    public Clave getClave() {
        return clave;
    }

    public void setClave(Clave clave) {
        this.clave = clave;
    }
}
```

```
public class Clave
{
    private String contraseña;

    public Clave(String contraseña) {
        this.contraseña = contraseña;
    }

    // Métodos getter y setter para la contraseña
    public String getContraseña() {
        return contraseña;
    }

    public void setContraseña(String contraseña) {
        this.contraseña = contraseña;
    }

}
```

```
public class Main
{
    public static void main(String[] args) {
        // Crear instancia de clave
        Clave claveUsuario1 = new Clave("contraseña123");

        // Crear instancia de usuario y asociar la clave
        Usuario usuario1 = new Usuario("Juan", claveUsuario1);

        // Acceder a la información del usuario y su clave utilizando getters
        System.out.println("Información del usuario:");
        System.out.println("------------------------");
        System.out.println("Nombre: " + usuario1.getNombre());
        System.out.println("Contraseña: " +
usuario1.getClave().getContraseña());
    }
}
```



Ejercicio Propuesto

Implementar un programa en Java que permita:

- Modelar un sistema de biblioteca
- Clases: Libro, Autor, Biblioteca
- Establecer relaciones de asociación entre las clases

UNIVERSIDAD TECNOLÓGICA DEL PERÚ

Facultad de Ingeniería

Programación Orientada a Objetos

Conclusiones:

- Las relaciones son fundamentales en POO.
- Permiten modelar sistemas complejos de manera eficiente.
- Un buen entendimiento mejora la calidad del diseño de software.

UNIVERSIDAD TECNOLÓGICA DEL PERÚ

Facultad de Ingeniería

Programación Orientada a Objetos

Actividad

Terminar de implementar los ejemplos 1,2,3 y el ejercicio propuesto.

UNIVERSIDAD TECNOLÓGICA DEL PERÚ

Facultad de Ingeniería

Programación Orientada a Objetos

Cierre

¿Qué hemos aprendido hoy?