TECNICAS DE PROGRAMA.ORIE. OBJ.

UNIVERSIDAD PRIVADA DEL NORTE

FACULTAD DE INGENIERÍA

INGENIERÍA DE SISTEMAS COMPUTACIONALES

TEMA:

EVALUACION T1 – TALLER DE CASOS

CURSO:

TECNICAS DE PROGRAMACION ORIENTADO A OBJETOS

INTEGRANTES:

GRUPO 03			
Estudiante 01	LOURDES SHARON HUAMAN MAMANI	**ID**	N00518813
Estudiante 02	MIGUEL JOHNNY HUARCAYA PINEDO	**ID**	N00244468
Estudiante 03	MANUEL ALBERTO MIRANDA ARONES	**ID**	N00443714
Estudiante 04	JOSE CARLOS SOSA BELLODAS	**ID**	N00417302

PROFESOR:

MARTIN EDUARDO TORRES RODRIGUEZ

TECNICAS DE PROGRAMA.ORIE. OBJ.

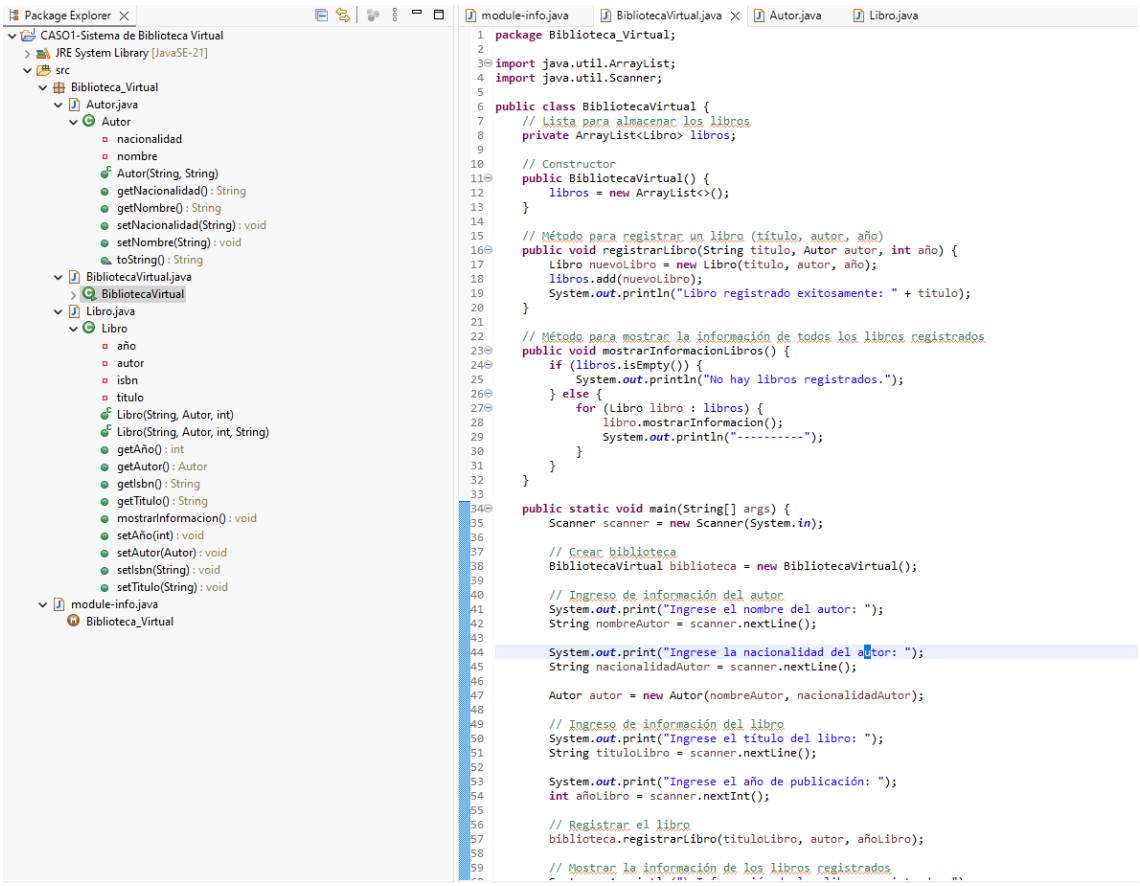
INDICE

CASO Nº 01 - SISTEMA DE BIBLIOTECA VIRTUAL 3
CASOS Nº 02 – GESTIÓN DE ESTUDIANTES.....5
CASOS Nº 03 – SISTEMA DE VENTAS EN LINEA 9
PREGUNTAS DE ANALISIS 13
LINK DE REPOSITORIO 13

TECNICAS DE PROGRAMA.ORIE. OBJ.

CASO Nº 01 - SISTEMA DE BIBLIOTECA VIRTUAL

Una universidad necesita un sistema básico para registrar libros y autores en una biblioteca virtual.



1. ¿Por qué se utilizó sobrecarga y qué ventajas tiene?

La sobrecarga se usó para tener dos formas de crear un libro. Si no tienes el ISBN, usas el constructor que no lo pide, y si lo tienes, usas el que lo incluye. La ventaja es que puedes crear un libro sin complicaciones, dependiendo de si tienes o no el ISBN. Así no tienes que andar metiendo datos innecesarios. Es como si te dieran dos opciones para pedir un café: con azúcar o sin azúcar. Tú eliges lo que más te convenga.

2. ¿Cómo protegerías la información con modificadores de acceso?

Ya en el código, los atributos (como el título, autor y año) son privados. Eso significa que nadie fuera de las clases puede cambiarlos directamente. Para poder modificarlos o leerlos, usas los métodos getter y setter.

Si quisieras hacer las cosas aún más seguras, podrías agregar validaciones dentro de los setter. Por ejemplo, que no te dejen poner un año negativo o un ISBN que no sea válido. Es como cuando pones una caja fuerte para guardar algo importante. La caja tiene una cerradura, y para abrirla, necesitas la llave (getter/setter).

3. ¿Qué colecciones usarías si tu sistema fuera a crecer mucho?

Ahora estás usando un ArrayList, y eso está bien cuando no tienes muchos libros. Pero si el sistema crece y tienes miles de libros, tal vez quieras algo más rápido para encontrar un libro.

Si quisieras buscar un libro rápidamente por su ISBN o algún identificador único, un HashMap sería perfecto, porque puedes buscar sin perder tiempo, incluso si tienes un montón de libros. Si no quieres libros duplicados, un HashSet sería lo ideal. Así, si intentas agregar el mismo libro dos veces, te lo impide automáticamente. Si tu necesidad es agregar o eliminar libros constantemente, una LinkedList sería mejor que el ArrayList, porque las inserciones y eliminaciones son más rápidas.

En resumen, la opción depende de lo que más necesites: ¿rapidez en la búsqueda?, ¿evitar duplicados?, ¿o agregar y quitar libros fácilmente?

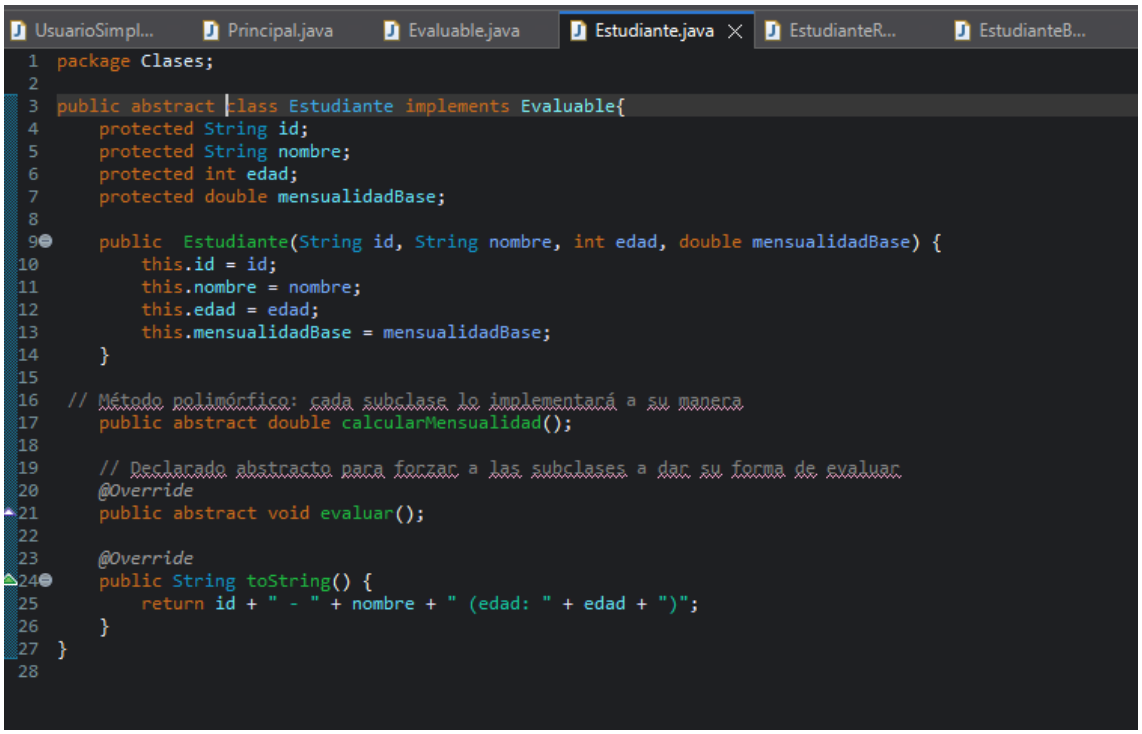
TECNICAS DE PROGRAMA.ORIE. OBJ.

CASOS Nº 02 – GESTIÓN DE ESTUDIANTES

Un colegio requiere un sistema para gestionar estudiantes y cursos.

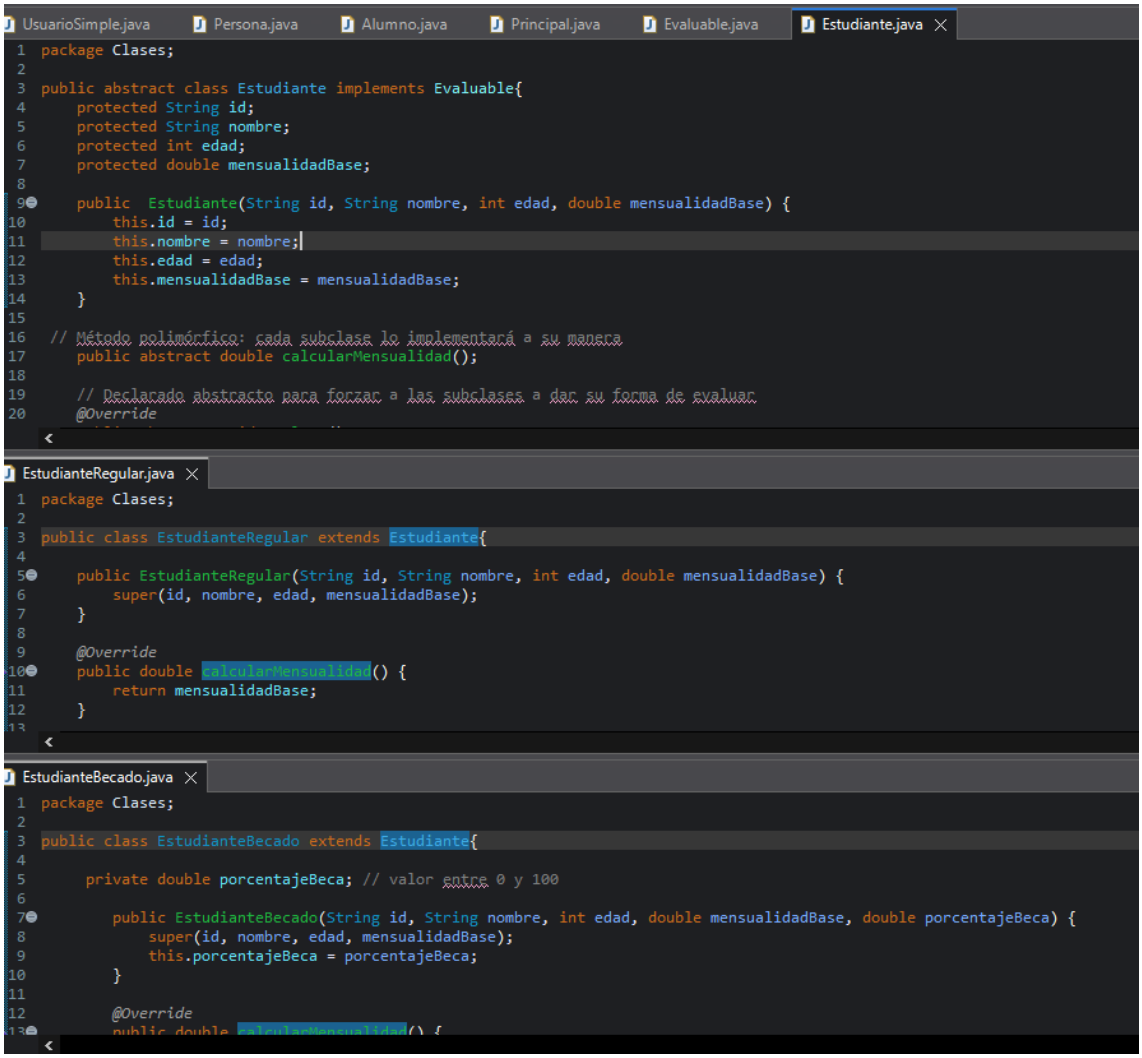
Requerimientos:

- Crear la clase Estudiante y la clase Curso.



- Establecer una relación entre ellas usando herencia y polimorfismo: - EstudianteRegular y EstudianteBecado deben heredar de Estudiante. - Cada uno debe implementar un método calcularMensualidad() (sobreescrito).

- En la imagen se evidencia la herencia al observar que la clase EstudianteBecado y la clase EstudianteRegular heredan de la clase Estudiante.

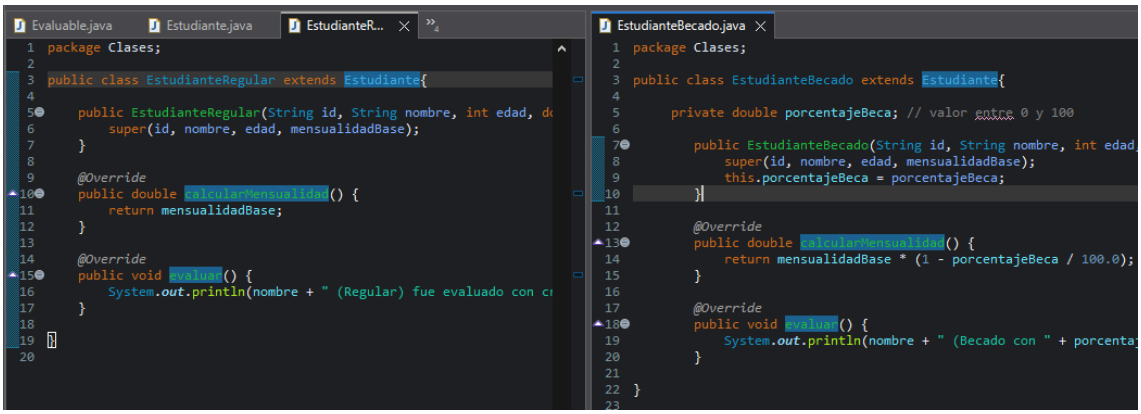


```java
// UsuarioSimple.java | Persona.java | Alumno.java | Principal.java | Evaluable.java | Estudiante.java
package Clases;

public abstract class Estudiante implements Evaluable{
    protected String id;
    protected String nombre;
    protected int edad;
    protected double mensualidadBase;

    public  Estudiante(String id, String nombre, int edad, double mensualidadBase) {
        this.id = id;
        this.nombre = nombre;
        this.edad = edad;
        this.mensualidadBase = mensualidadBase;
    }

 // Método polimórfico: cada subclase lo implementará a su manera
    public abstract double calcularMensualidad();

    // Declarado abstracto para forzar a las subclases a dar su forma de evaluar
    @Override
```

```java
// EstudianteRegular.java
package Clases;

public class EstudianteRegular extends Estudiante{

    public EstudianteRegular(String id, String nombre, int edad, double mensualidadBase) {
        super(id, nombre, edad, mensualidadBase);
    }

    @Override
    public double calcularMensualidad() {
        return mensualidadBase;
    }
```

```java
// EstudianteBecado.java
package Clases;

public class EstudianteBecado extends Estudiante{

     private double porcentajeBeca; // valor entre 0 y 100

        public EstudianteBecado(String id, String nombre, int edad, double mensualidadBase, double porcentajeBeca) {
            super(id, nombre, edad, mensualidadBase);
            this.porcentajeBeca = porcentajeBeca;
        }

        @Override
        public double calcularMensualidad() {
```

Y el polimorfismo se evidencia cuando se usa el método calcularMensualidad y para ambos casos tienen diferentes comportamientos.



```java
// Evaluable.java | Estudiante.java | EstudianteR...
package Clases;

public class EstudianteRegular extends Estudiante{

    public EstudianteRegular(String id, String nombre, int edad, d
        super(id, nombre, edad, mensualidadBase);
    }

    @Override
    public double calcularMensualidad() {
        return mensualidadBase;
    }

    @Override
    public void evaluar() {
        System.out.println(nombre + " (Regular) fue evaluado con cr
    }
```

```java
// EstudianteBecado.java
package Clases;

public class EstudianteBecado extends Estudiante{

     private double porcentajeBeca; // valor entre 0 y 100

        public EstudianteBecado(String id, String nombre, int edad,
            super(id, nombre, edad, mensualidadBase);
            this.porcentajeBeca = porcentajeBeca;
        }

        @Override
        public double calcularMensualidad() {
            return mensualidadBase * (1 - porcentajeBeca / 100.0);
        }

        @Override
        public void evaluar() {
            System.out.println(nombre + " (Becado con " + porcenta
        }

}
```

TECNICAS DE PROGRAMA.ORIE. OBJ.

UsuarioSimple.java | Persona.java | Alumno.java | Principal.java | Evaluable.java | Estudiante.java | EjeProcPrincipal.j... | Curso.java

```java
package Clases;
import java.util.ArrayList;
import java.util.List;

public class Curso {
    private String codigo;
    private String nombre;
    private List<Estudiante> estudiantes;

    public Curso(String codigo, String nombre) {
        this.codigo = codigo;
        this.nombre = nombre;
        this.estudiantes = new ArrayList<>();
    }

    public void agregarEstudiante(Estudiante e) {
        estudiantes.add(e);
    }

    public void mostrarEstudiantes() {
        System.out.println("Curso: " + codigo + " - " + nombre);
        for (Estudiante e : estudiantes) {
            System.out.println(e.toString() + " | Mensualidad: " + e.calcularMensualidad());
        }
    }

    public double calcularMensualidadTotal() {
        double total = 0;
        for (Estudiante e : estudiantes) {
            total += e.calcularMensualidad();
        }
        return total;
    }
}
```

- Usar interfaces para definir el contrato Evaluable, que incluya el método evaluar().

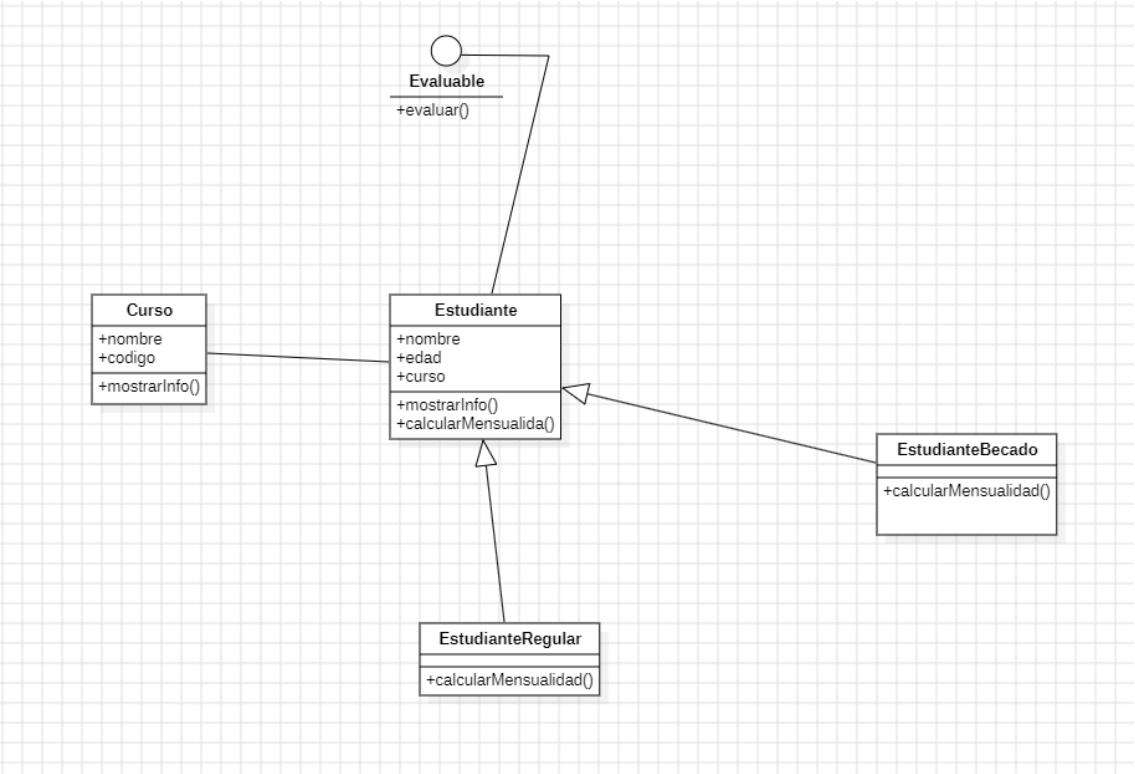
```java
package Clases;

public abstract class Estudiante implements Evaluable{
    protected String id;
    protected String nombre;
    protected int edad;
    protected double mensualidadBase;

    public  Estudiante(String id, String nombre, int edad, double mensualidadBase) {
        this.id = id;
        this.nombre = nombre;
        this.edad = edad;
        this.mensualidadBase = mensualidadBase;
    }

 // Método polimórfico: cada subclase lo implementará a su manera
    public abstract double calcularMensualidad();

    // Declarado abstracto para forzar a las subclases a dar su forma de evaluar
    @Override
    public abstract void evaluar();

    @Override
    public String toString() {
        return id + " - " + nombre + " (edad: " + edad + ")";
    }
}
```

UsuarioSimpl... | Principal.java | Evaluable.java | Estudiante.java

```java
package Clases;

public interface Evaluable {
    void evaluar();
}
```

TECNICAS DE PROGRAMA.ORIE. OBJ.

Representar las relaciones en un diagrama UML.



Preguntas de análisis:

- ¿Por qué es útil aplicar herencia y polimorfismo en este sistema?

Es útil porque no ayuda a trabajar reutilizando código y creando jerarquías lógicas y en cuanto al polimorfismo podemos de alguna manera también ahorrar líneas de código ya que usamos mismos métodos que se comportan de diferente manera dependiendo que objeto los use.

- ¿Qué problemas podrían surgir si no se usaran interfaces en el diseño?

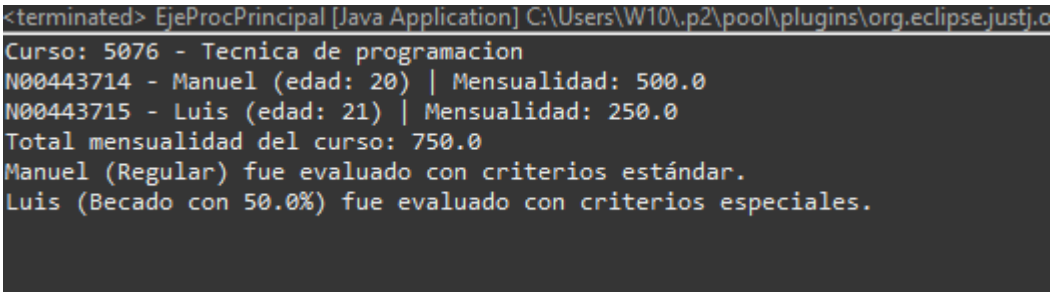
El codigo sería más rígido, habría lógica duplicada.

- ¿Qué diferencia hay entre atributos/métodos estáticos y no estáticos en este contexto?

Los atributos son las caracteristicas propias que tienen cada clase y los métodos no estáticos como calcularMensualidad() son aquellos que dependen de que objeto es llamado como en el caso de si es un estudiante

regular o becado quien invoca al método mientras que un método estático no depende de un objeto en específico estos métodos solo trabajan con atributos estáticos.

Ejecución final



CASOS Nº 03 – SISTEMA DE VENTAS EN LINEA

Una tienda en línea necesita un sistema sencillo para gestionar productos y ventas.

Requerimientos:

Crear la clase Producto (nombre, precio, stock).

```java
public class Producto {

    private String nombre;
    private double precio;
    private int stock;

    public Producto(String nombre, double precio, int stock) {
        this.nombre = nombre;
        this.precio = precio;
        this.stock = stock;
    }

    public String getNombre() {
        return nombre;
    }

    public double getPrecio() {
        return precio;
    }

    public int getStock() {
        return stock;
    }

    public void disminuirStock(int cantidad) throws Exception {
        if (cantidad > stock) {
            throw new Exception("Stock insuficiente para el producto: " +
        }
        stock -= cantidad;
    }

    @Override
    public String toString() {
        return nombre + " - Precio: " + precio + " - Stock: " + stock;
    }
}
```

Implementar un ArrayList para registrar múltiples productos.

caso3/src/caso03/Producto.java

```java
package caso03;
import java.util.ArrayList;
import java.util.Scanner;


public class SistemaVentas {

    private ArrayList<Producto> productos;
    private ArrayList<Venta> ventas;

    public SistemaVentas() {
        productos = new ArrayList<>();
        ventas = new ArrayList<>();
    }
```

TECNICAS DE PROGRAMA.ORIE. OBJ.

Crear un método que realice una compra:

- Disminuir el stock.

- Manejar errores si el stock es insuficiente.

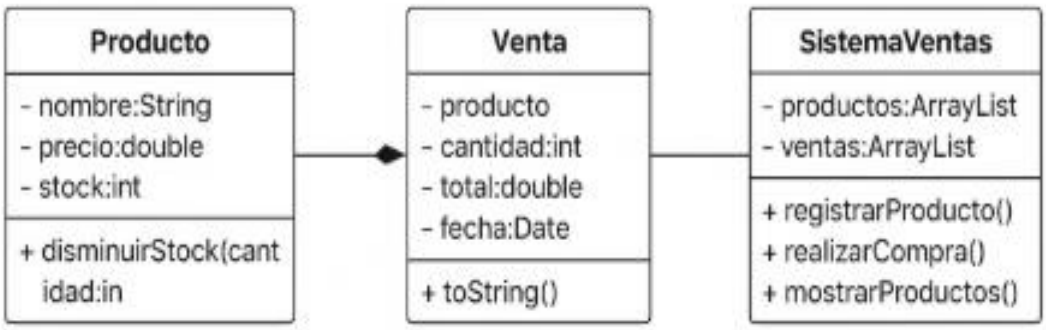
```java
public void registrarProducto(Producto p) {
    productos.add(p);
}

public void mostrarProductos() {
    System.out.println("\n--- Lista de productos ---");
    for (int i = 0; i < productos.size(); i++) {
        System.out.println(i + ". " + productos.get(i));
    }
}

public void realizarCompra(int index, int cantidad) {
    try {
        Producto p = productos.get(index);
        Venta v = new Venta(p, cantidad);
        ventas.add(v);
        System.out.println("✅ Compra realizada: " + v);
    } catch (Exception e) {
        System.out.println("⚠ Error en la compra: " + e.getMessage());
    }
}

public static void main(String[] args) {
    SistemaVentas sistema = new SistemaVentas();
    Scanner sc = new Scanner(System.in);


    sistema.registrarProducto(new Producto("Laptop", 2500.0, 5));
    sistema.registrarProducto(new Producto("Mouse", 50.0, 10));
    sistema.registrarProducto(new Producto("Teclado", 120.0, 7));


    sistema.mostrarProductos();
    System.out.print("\nIngrese el índice del producto a comprar: ");
    int index = sc.nextInt();
    System.out.print("Ingrese cantidad: ");
    int cantidad = sc.nextInt();

    sistema.realizarCompra(index, cantidad);

    sc.close();
}
```

TECNICAS DE PROGRAMA.ORIE. OBJ.

Incluir una clase Venta que guarde información de cada transacción.

```java
package caso03;
import java.util.Date;

public class Venta {

    private Producto producto;
    private int cantidad;
    private double total;
    private Date fecha;

    public Venta(Producto producto, int cantidad) throws Exception {
        this.producto = producto;
        this.cantidad = cantidad;
        producto.disminuirStock(cantidad);
        this.total = cantidad * producto.getPrecio();
        this.fecha = new Date();
    }

    public double getTotal() {
        return total;
    }

    @Override
    public String toString() {
        return "Venta de " + cantidad + " x " + producto.getNombre() +
               " | Total: " + total + " | Fecha: " + fecha;
    }
}
```

Representar las clases y relaciones en un diagrama UML.



TECNICAS DE PROGRAMA.ORIE. OBJ.

PREGUNTAS DE ANALISIS

- ¿Cómo se evidencia el buen uso de colecciones en este caso?

- Se utiliza ArrayList para almacenar múltiples productos y ventas.
- Permite agregar, recorrer y gestionar dinámicamente elementos sin necesidad de definir un tamaño fijo.
- Facilita la búsqueda y el manejo de información en el sistema.

- ¿Qué excepciones deberías manejar y por qué?

- *IndexOutOfBoundsException*: si el usuario ingresa un índice inválido de producto.
- Excepción personalizada o genérica en stock: cuando se intenta comprar más cantidad de la disponible.
- *InputMismatchException*: si el usuario ingresa datos no válidos en lugar de números.

- ¿Cómo se podría aplicar composición en este sistema?

- La clase Venta contiene un objeto Producto, lo cual significa que una venta no existe sin un producto asociado → relación de composición.
- Además, el SistemaVentas compone listas de Productos y Ventas, gestionando el ciclo de vida de estos objetos.

LINK DE REPOSITORIO

https://github.com/SharonHua0/T1-Taller-de-casos.git