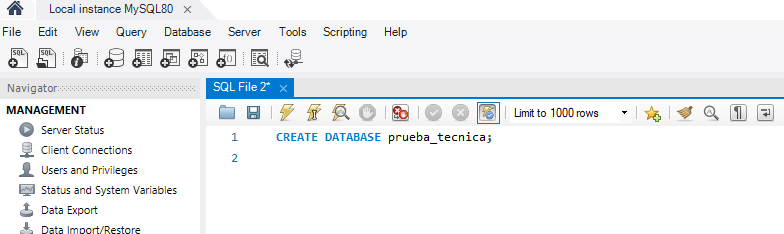
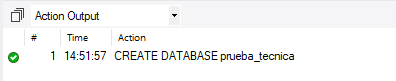
**Documentación practica Bancolombia**

**Sección 1: SQL**

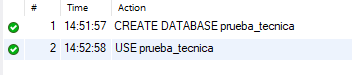
**Primero debemos crear la base de datos**

****

****

**Ahora debemos seleccionarla para poder usarla**

****

****

**Ejercicio 1: Consulta SQL**

Dada la siguiente estructura de base de datos:

CREATE TABLE Clientes (

id INT PRIMARY KEY,

nombre VARCHAR(100),

email VARCHAR(100),

fecha\_registro DATE

);

CREATE TABLE Pedidos (

id INT PRIMARY KEY,

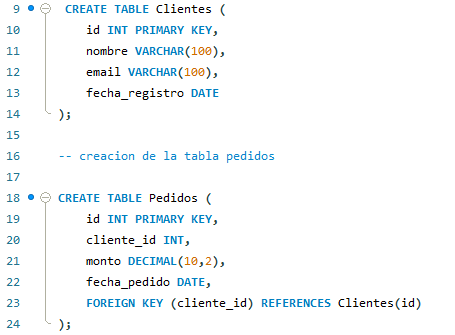
cliente\_id INT,

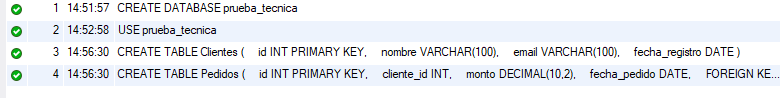
monto DECIMAL(10,2),

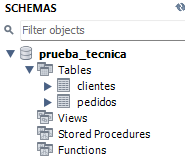
fecha\_pedido DATE,

FOREIGN KEY (cliente\_id) REFERENCES Clientes(id)

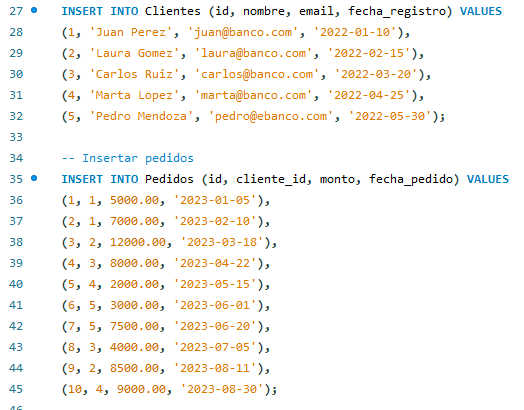
);

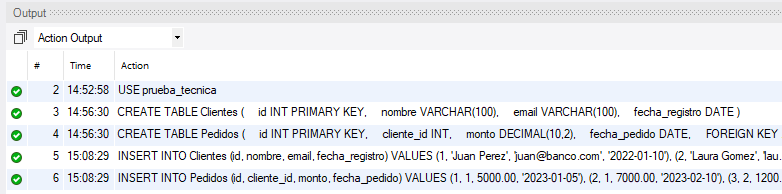


****



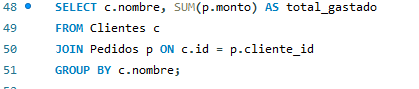
**Ahora debemos llenar las tablas con datos de prueba, debemos asegurarnos que cumplimos con los parámetros**



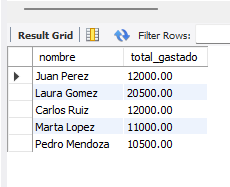


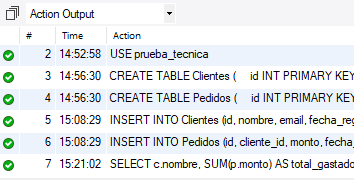
1. Escribe una consulta SQL para obtener el nombre de cada cliente con el total de dinero que ha gastado en pedidos. Solo incluir clientes que han hecho al menos un pedido.

Consulta SQL primera condición obtener el “nombre”, segunda condición total de dinero que ha gastado en sus pedidos, tercera condición solo mostraremos los clientes que tienen al menos un pedido

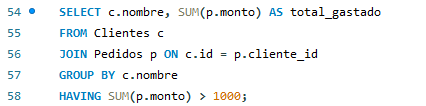


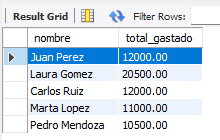
Porque usar **join** en vez de una condición. nos ahorra el trabajo debido a que join nos devuelve solo coincidencias entre clientes y pedidos. Nos ahorramos una condición de traer todos los clientes.





2. Agrega una condición para mostrar solo los clientes que hayan gastado más de 1000.





3. Ejecuta la consulta en un entorno SQL y toma una captura de pantalla con los resultados

**Ejercicio 2: Corrección de consulta**

Se tiene la siguiente consulta:

SELECT nombre, SUM(monto)

FROM Clientes, Pedidos

WHERE Clientes.id = Pedidos.cliente\_id

1. **Identifica y explica los errores en la consulta.**

**SELECT nombre, SUM(monto)**

Para el uso de **sum** debe ir un **group by** porque se está haciendo una agregación sin agrupar por su nombre

Hace un **select** de la columna **nombre** y una suma de monto, pero No se especifica de qué tabla proviene monto, lo que puede causar ambigüedad, podríamos usar “p” para la tabla Pedidos para mejorar la claridad

**Select nombre, sum(p.monto)**

Ahora no es claro donde se mostrará dicho resultado por lo que debemos agregar un alias para que sea claro como total\_gastado, el resultado aparecería con un nombre poco legible como SUM(monto) en lugar de algo más claro como total\_gastado

**Select nombre, sum(p.monto) AS total\_gastado**

Luego nos dicen que:

**FROM Clientes, Pedidos**

este join ya es obsoleto por lo cual debe usarse **join on**  que es mucho más claro y preciso.

**Where Clientes.id = Pedidos.cliente\_id;**

Si bien la sentencia anterior funcionaria es obsoleta La cláusula WHERE se debe reservar para filtros sobre los datos, no para unir tablas. Como filtrar monto, fechas etc.

**FROM Clientes c**

**JOIN Pedidos p ON c.id = p.cliente\_id**

Por último nos faltaría el uso de group by c

GROUP BY c.nombre;

Cuando se usa SUM() junto a otra columna, como nombre, es obligatorio agrupar con GROUP BY

1. Corrige la consulta para que funcione correctamente.

**SELECT c.nombre, SUM(p.monto) AS total\_gastado**

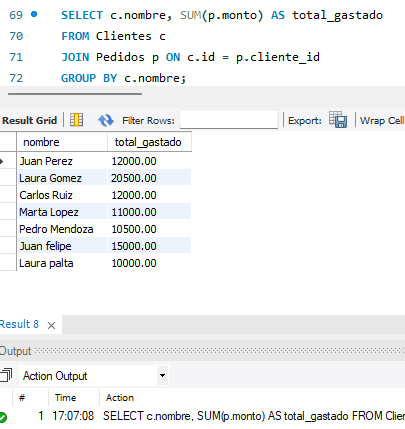
**FROM Clientes c**

**JOIN Pedidos p ON c.id = p.cliente\_id**

**GROUP BY c.nombre;**

En la consulta anterior es similar a la del primer punto, la que tenía errores es una versión funcional, pero genera errores por lo obsoleta y difícil legibilidad

1. Ejecuta la consulta corregida y toma una captura de pantalla con los resultados.



**Sección 2: Java con Programación Orientada a Objetos**

Ejercicio 1: Modelado de clases

Diseña una clase en Java para representar un "Producto" con los siguientes atributos:

• ID (entero)

• Nombre (cadena)

• Precio (decimal)

• Cantidad en stock (entero)

Incluye:

• Constructor con todos los atributos

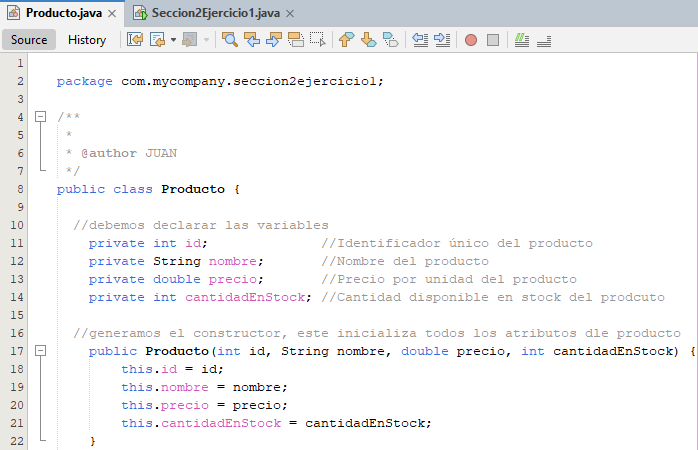
• Métodos getters y setters

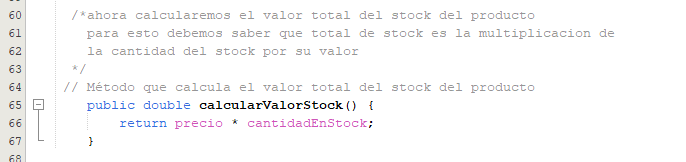
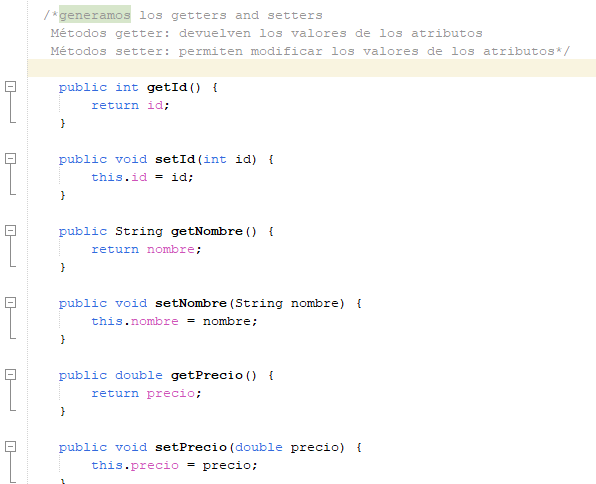
• Un método para calcular el valor total del stock del producto

• Implementa la clase en un entorno de desarrollo, prueba su funcionamiento y toma una captura de pantalla con

los resultados.

**Crearemos nuestra clase Producto y sus respectivos atributos**

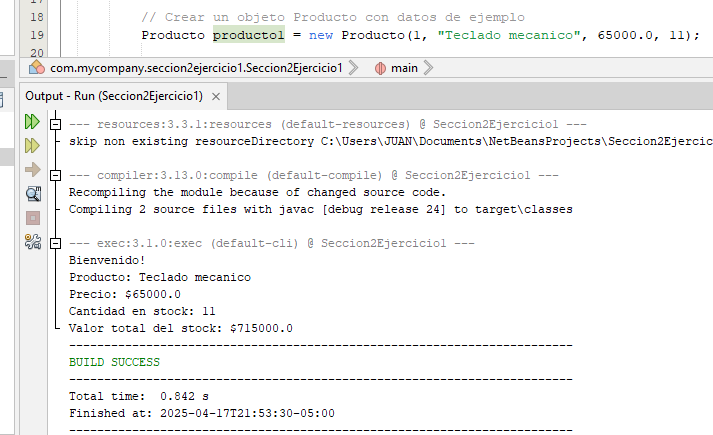




**Nuestro principal**



**Respectiva compilación o salida**



**Ejercicio 2: Análisis de código**

Analiza el siguiente código en Java:

public class Cliente {

private String nombre;

private int edad;

public Cliente(String nombre, int edad) {

nombre = nombre;

edad = edad;

}public void mostrarInformacion() {

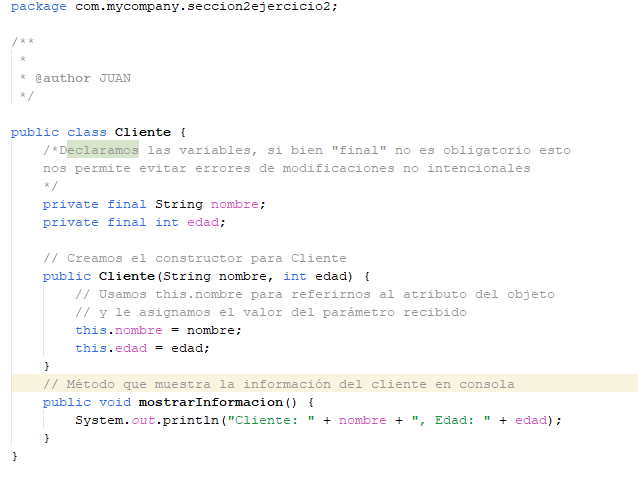
System.out.println("Cliente: " + nombre + ", Edad: " + edad);

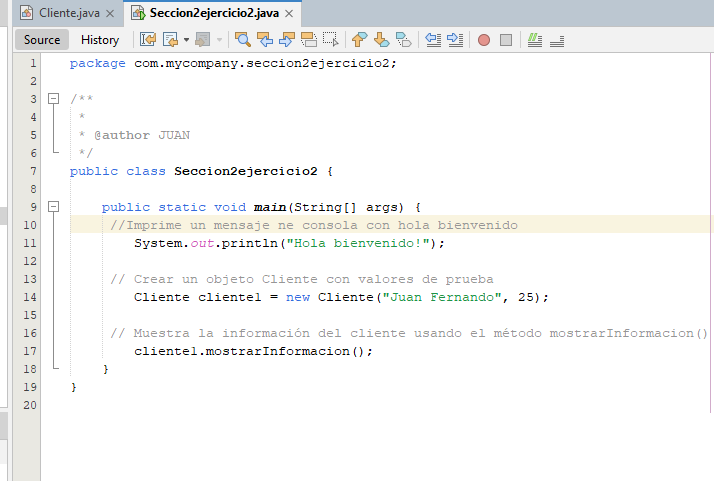
}

}

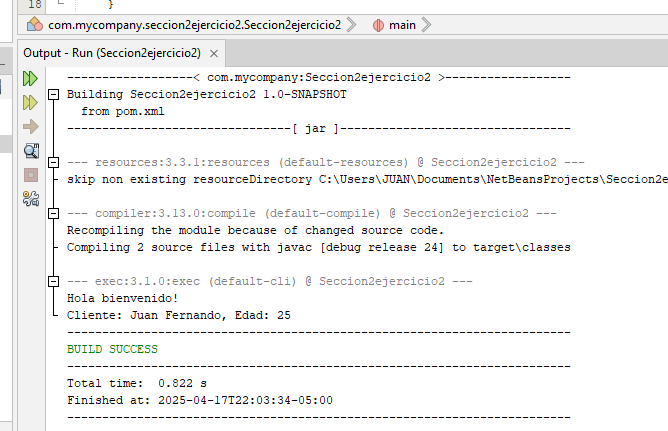
1. **Identifica los errores y explica por qué ocurren.**

R/ como podemos observar cuando en el constructor hace falta **el this.** para indicar que estás asignando el parámetro al atributo del objeto.

1. **Corrige el código y proporciona una versión funcional.** 



1. **Ejecuta el código corregido y toma una captura de pantalla con la salida en consola.**



**Sección 3: Análisis de Problemas**

Ejercicio 1: Lógica de programación

Escribe un algoritmo en Java que reciba una lista de enteros y devuelva el segundo número más grande de la lista.

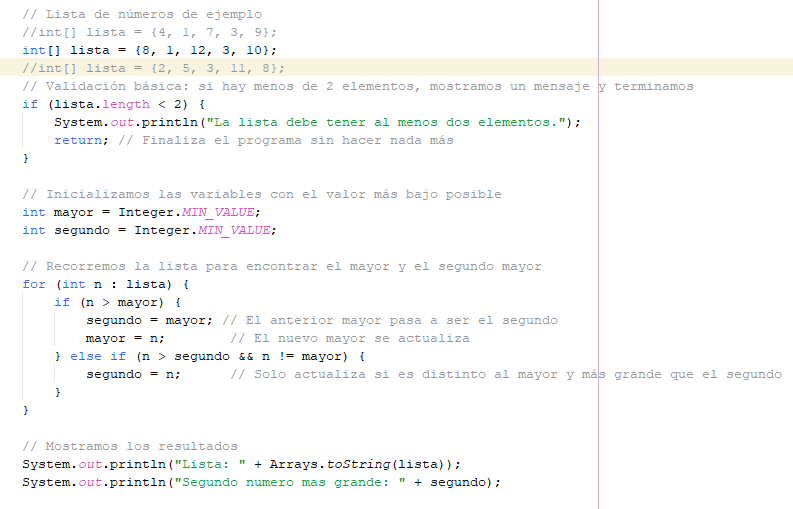
Ejemplo:

Entrada: [4, 1, 7, 3, 9]

Salida: 7

1. Implementa el algoritmo en Java y prueba diferentes entradas.

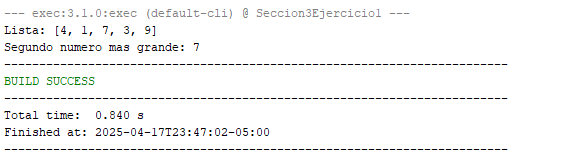
2. Toma capturas de pantalla de la ejecución con distintos casos de prueba.

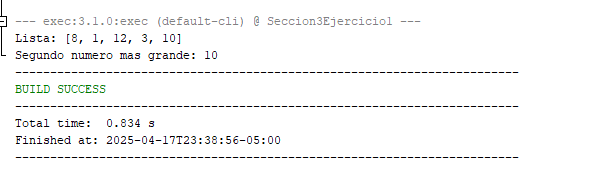
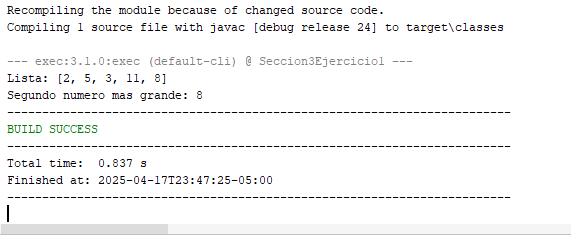


Ejemplo:

Entrada: [4, 1, 7, 3, 9]

Salida: 7





**Ejercicio 2: Resolución de problema**

Se está desarrollando un sistema de reservas para un cine. Un cliente solo puede reservar si hay al menos 2 asientos

disponibles en la misma fila. Diseña un método en Java que reciba una matriz 2D de asientos (1 = ocupado, 0 = libre) y

devuelva si un cliente puede hacer una reserva.

Ejemplo:

Entrada:

[

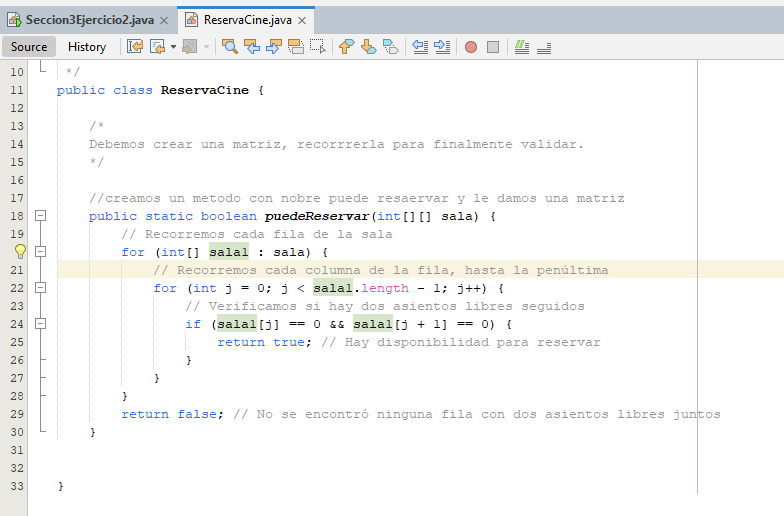
[1, 0, 0, 1],

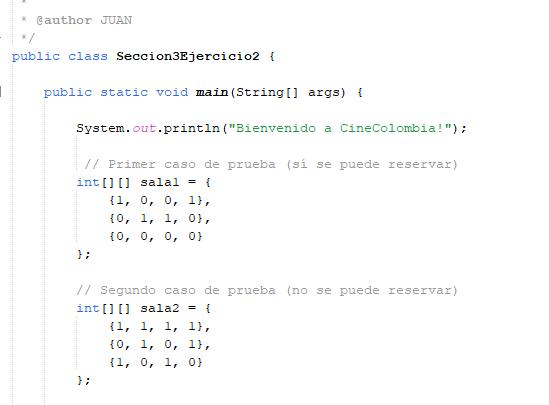
[0, 1, 1, 0],

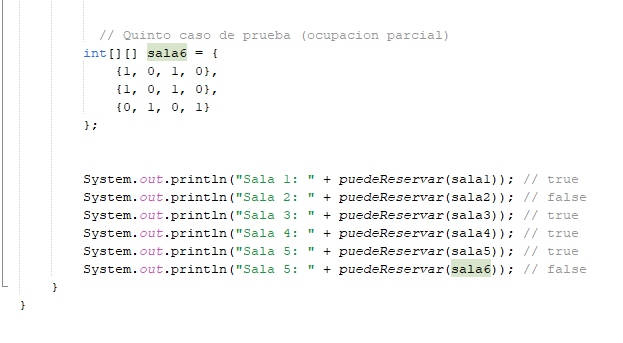
[0, 0, 0, 0]

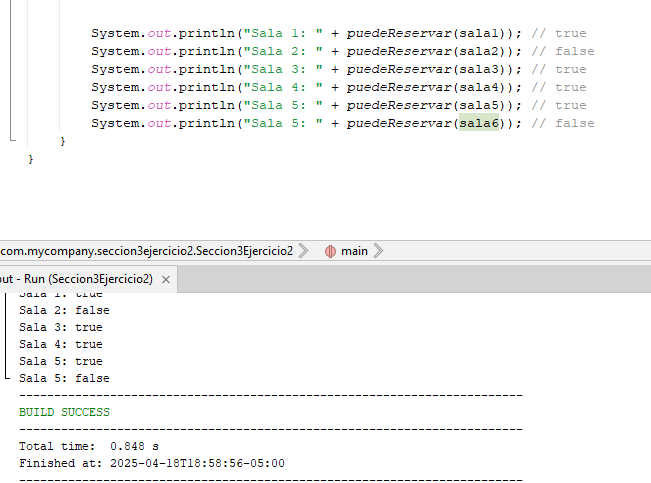
]

Salida: true (Hay una fila con al menos dos asientos libres juntos)









**Sección 4: Desarrollo de Interfaz Gráfica y Generación de Archivos**

**Ejercicio 1: Creación de una Interfaz Gráfica**

Desarrolla una aplicación en Java con Swing o JavaFX que contenga:

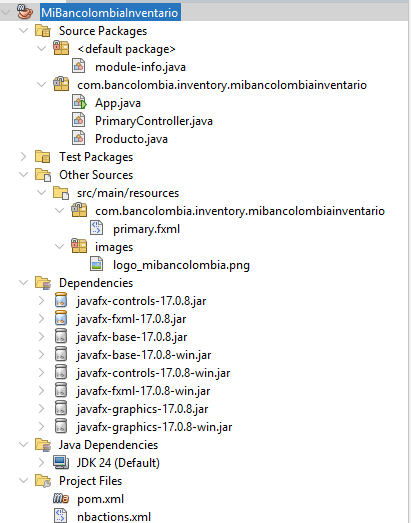
• Un diseño visual atractivo con un logo de la empresa.

• Un formulario donde el usuario pueda ingresar información de un producto (nombre, precio, cantidad en stock).

• Un botón para guardar la información ingresada en memoria.

• Ejecuta la aplicación y toma capturas de pantalla mostrando la interfaz en funcionamiento

**Estructura del proyecto JavaFX**



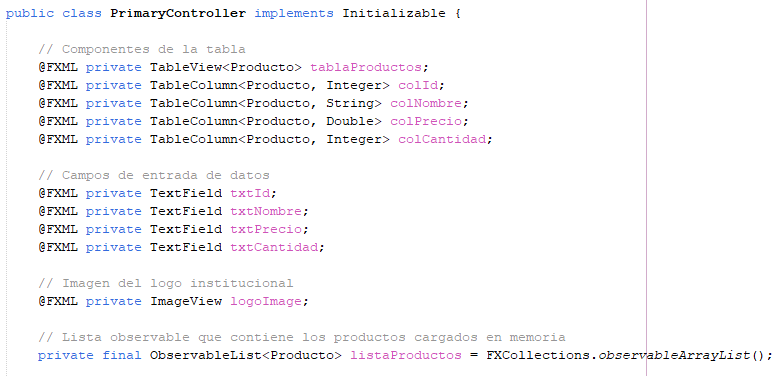
**Creación de la clase producto**

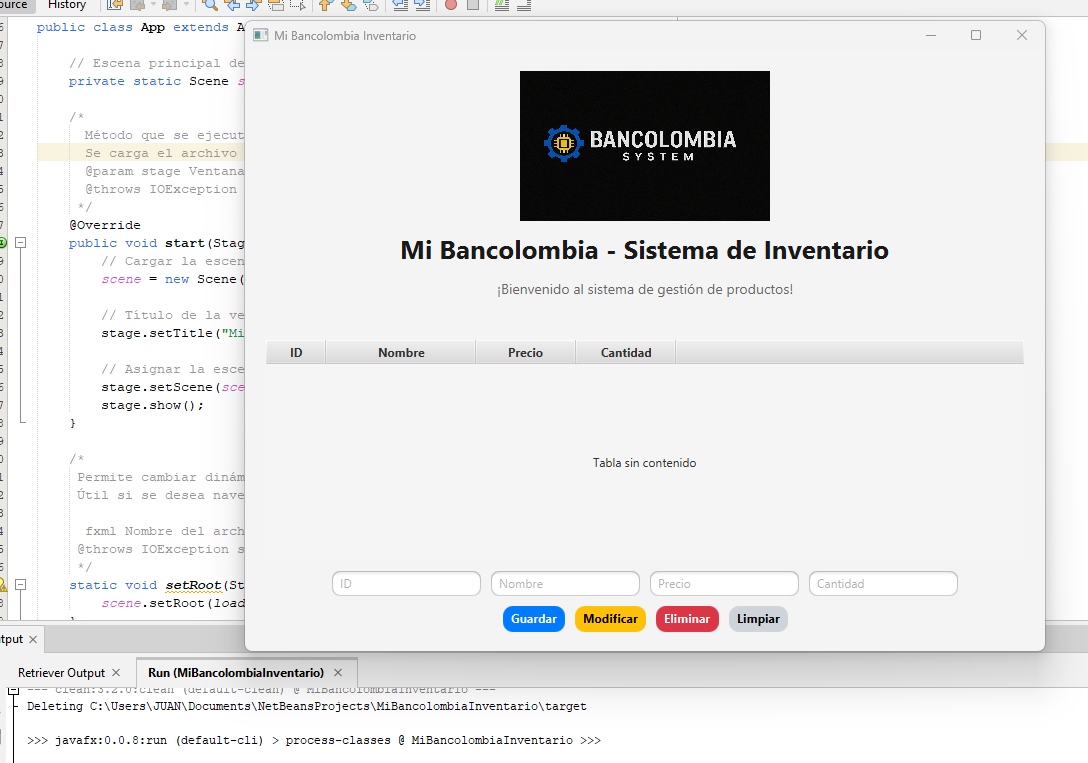


**Creación del fxml**

****

**Creación de controlador**

****

****

****

**Ejercicio 2: Generación de Archivo XLSX con Contraseña**

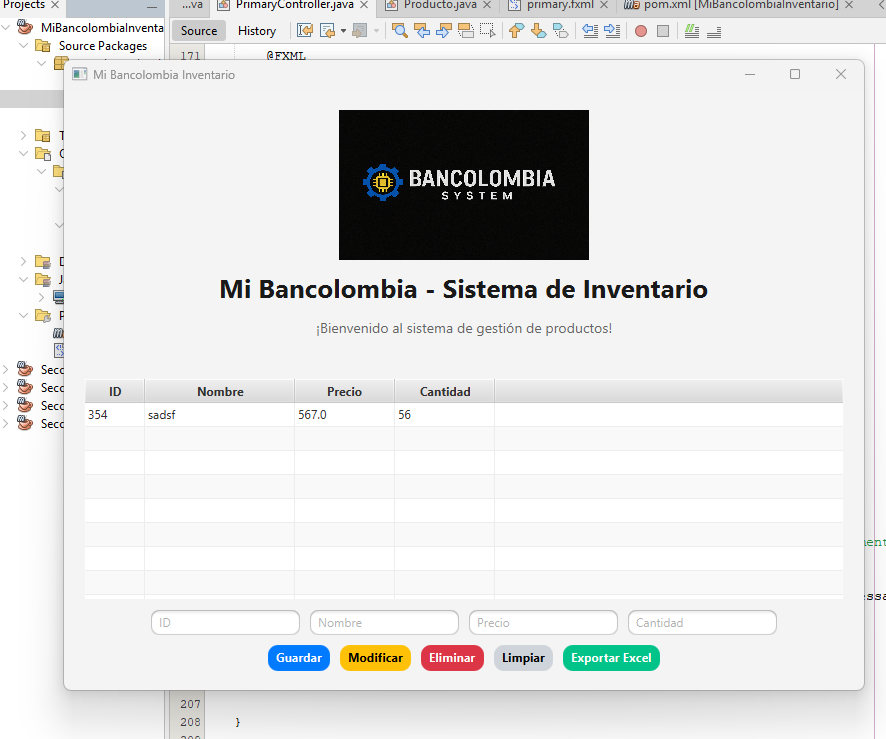
Modifica la aplicación del ejercicio anterior para que, al presionar el botón de guardar, se genere un archivo Excel

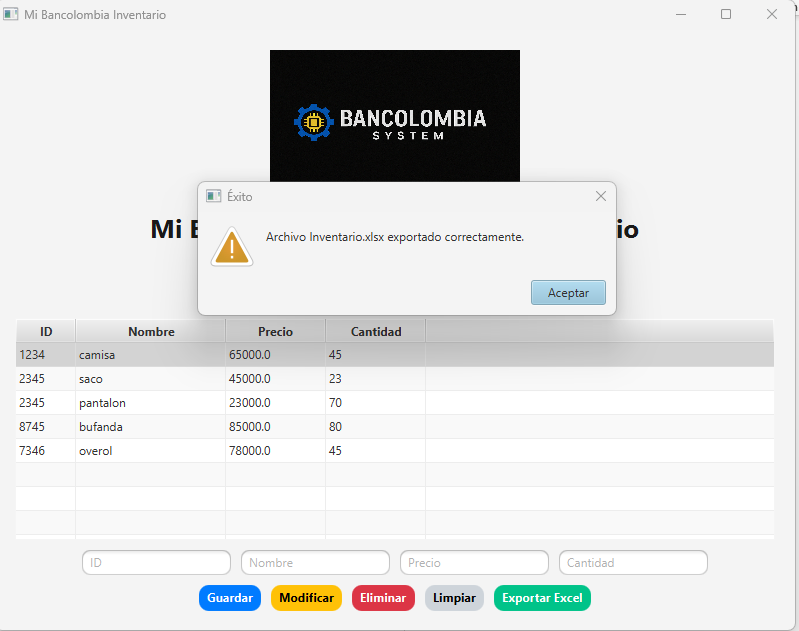
(.xlsx) con la información ingresada.

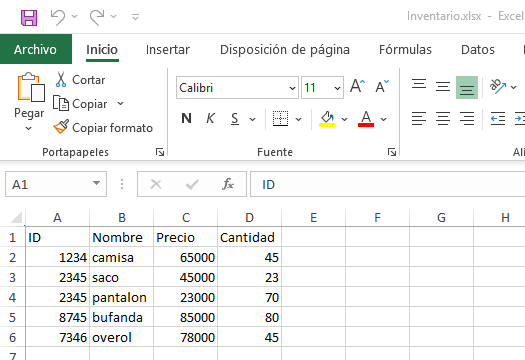
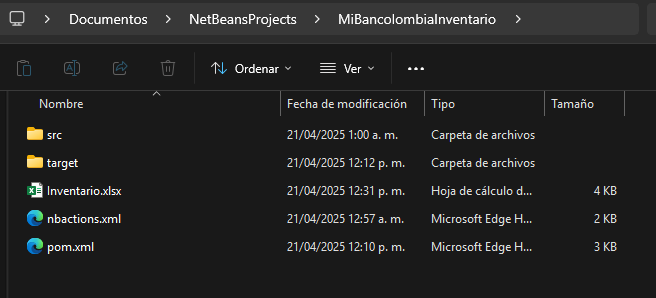
• El archivo debe estar bloqueado con una contraseña definida en el código.

• Usa la librería Apache POI para la generación del archivo Excel.

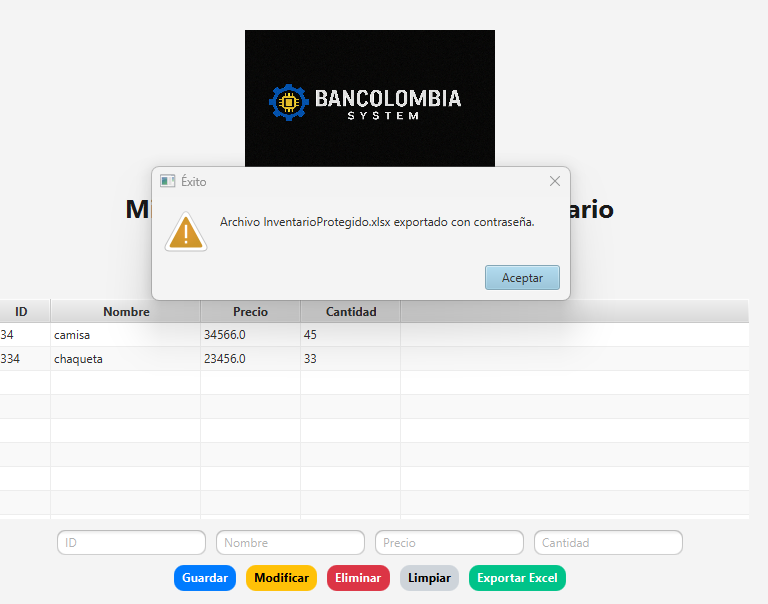
• Genera el archivo y toma una captura de pantalla mostrando su contenido bloqueado con la contraseña.

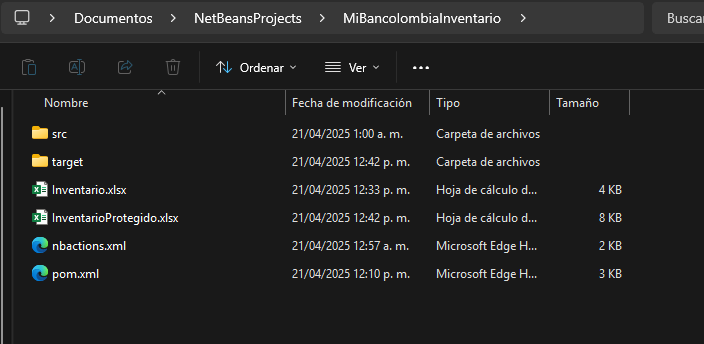


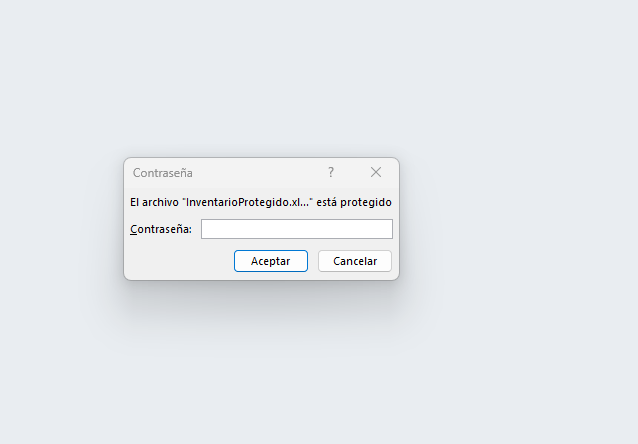


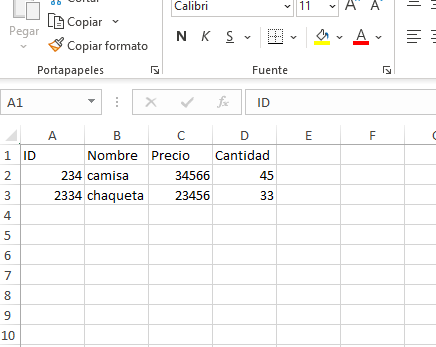


**Archivo Excel con contraseña **

****

****

****

****