|  |
| --- |
|  |
| HITO 1 DEL 1º TRIMESTRE DE Acceso a datos |
| |  |  |  | | --- | --- | --- | | Álex de Santiago Vicente | 10/28/24 | 2 DAM | |

índice

[1 Introducción 2](#_Toc180755129)

[2 Paradigma Orientado a Objetos 2](#_Toc180755130)

[2.1 Principios de la POO 2](#_Toc180755131)

[3 Modelo de Clases 2](#_Toc180755132)

[3.1 Clase Cuenta: 2](#_Toc180755133)

[3.2 Clase Cliente: 3](#_Toc180755134)

[4 Implementación 3](#_Toc180755135)

[4.1 Carga y Persistencia de Datos 3](#_Toc180755136)

[4.2 Manejo de Errores 3](#_Toc180755137)

[6 Accesibilidad y Experiencia del Usuario 3](#_Toc180755138)

[8 Excepciones y Robustez 3](#_Toc180755139)

[8.1 Gestión de Excepciones 3](#_Toc180755140)

[9 Robustez de la Aplicación 4](#_Toc180755141)

[11 Conclusiones 4](#_Toc180755142)

[Conclusión 5](#_Toc180755143)

[Bibliografía 5](#_Toc180755144)

**Informe de Desarrollo de la Aplicación de Gestión de Cuentas Bancarias**

# Introducción

En este proyecto, he desarrollado una aplicación Java para gestionar cuentas bancarias que permite a los clientes realizar ingresos y retiros, mientras se mantiene un registro de los movimientos. Utilizando el paradigma de programación orientada a objetos (POO), he implementado clases que encapsulan la funcionalidad de la cuenta, el cliente y los movimientos. Este enfoque me ha permitido crear un código modular y fácilmente mantenible.

# Paradigma Orientado a Objetos

## Principios de la POO

He utilizado varios principios clave de la POO en el desarrollo de esta aplicación:

* **Encapsulación**: Cada clase tiene atributos y métodos que se encapsulan para proteger el estado interno de los objetos. Por ejemplo, los atributos de Cliente, Movimiento, y Cuenta están marcados como privados y se acceden mediante métodos públicos.
* **Abstracción**: He abstraído los conceptos de cliente, cuenta y movimiento en clases separadas. Cada clase representa una entidad única en el sistema, lo que permite centrarse en su funcionalidad sin preocuparse por los detalles de implementación.
* **Herencia**: Aunque no he utilizado herencia en este proyecto, el diseño de las clases permite que se puedan extender en el futuro si es necesario. Por ejemplo, podría implementar una clase CuentaAhorro que herede de Cuenta.
* **Polimorfismo**: Se puede implementar polimorfismo en un futuro si se decide agregar diferentes tipos de movimientos (como transferencia, pago, etc.), utilizando una interfaz común.

# Modelo de Clases

La estructura de la aplicación está compuesta por tres clases principales: Cuenta, Cliente y Movimiento.

## Clase Cuenta:

* + Contiene un objeto de tipo Cliente.
  + Mantiene una lista de objetos Movimiento.
  + Proporciona métodos para realizar ingresos y retiros, asegurando que las operaciones sean seguras y lógicas (por ejemplo, no permitir retiros que excedan el saldo disponible).

## Clase Cliente:

* + Almacena información sobre el cliente, como el nombre y el DNI.
  + Proporciona métodos para acceder a estos datos.

# Implementación

## Carga y Persistencia de Datos

La aplicación utiliza un archivo cuenta.dat para almacenar y recuperar la información de la cuenta. Al iniciar el programa, se verifica si el archivo existe:

* **Carga de Datos**: Si el archivo existe, la cuenta se carga desde él. Si no, se solicita al usuario que ingrese los datos del cliente para crear una nueva cuenta.
* **Persistencia de Datos**: Al finalizar la ejecución, la cuenta se guarda de nuevo en el archivo para que los datos se mantengan entre sesiones.

## Manejo de Errores

1. He implementado un manejo robusto de errores para garantizar que el programa sea resistente a entradas inválidas:

* **Verificación de Entradas**: Se utilizan bloques try-catch para manejar excepciones como InputMismatchException, asegurando que el usuario solo ingrese datos válidos (números para cantidades).
* **Comprobación de Saldo**: Antes de realizar un retiro, se verifica que la cantidad solicitada no exceda el saldo disponible. Si es así, se informa al usuario con un mensaje claro.

# Accesibilidad y Experiencia del Usuario

1. Para mejorar la accesibilidad y la experiencia del usuario, he utilizado colores ANSI para resaltar mensajes importantes y errores en la consola. Esto ayuda a que el usuario se enfoque en la información crítica y facilita la interacción con el programa. Por ejemplo:

* **Verde**: Utilizado para mensajes de éxito (como confirmaciones de ingresos o retiros).
* **Rojo**: Utilizado para errores y advertencias (como saldo insuficiente o entradas inválidas).
* **Amarillo**: Utilizado para mostrar opciones de menú.

# Excepciones y Robustez

## Gestión de Excepciones

Las excepciones se gestionan de manera que no interrumpan la ejecución del programa. Esto es especialmente importante en aplicaciones de usuario donde se espera que el programa siga funcionando incluso después de un error. He utilizado excepciones personalizadas para manejar situaciones específicas:

* **InputMismatchException**: Capturada al leer entradas, redirigiendo al usuario a ingresar datos válidos.
* **IOException**: Capturada durante la carga y guardado de datos, informando al usuario en caso de problemas al acceder al archivo.

# Robustez de la Aplicación

1. La robustez se ha implementado al asegurar que todas las entradas se validen y que las operaciones críticas se manejen adecuadamente. Además, el uso de estructuras de datos como ArrayList permite gestionar dinámicamente los movimientos de la cuenta.

# Conclusiones

El desarrollo de esta aplicación ha sido una experiencia enriquecedora. He aprendido a implementar y aplicar los principios de la programación orientada a objetos, lo que me ha permitido estructurar el código de una manera clara y eficiente. He hecho hincapié en el manejo de errores, asegurando que el programa sea robusto y fácil de usar.

Este enfoque no solo facilita el mantenimiento y la escalabilidad del código, sino que también me ha preparado para futuros proyectos donde la POO será esencial. Estoy satisfecho con el resultado final y confío en que esta aplicación será útil para gestionar cuentas bancarias de manera efectiva.

Capturas de Pantalla

A continuación, presento capturas de pantalla que demuestran la ejecución del programa y las diferentes opciones disponibles para el usuario. Estas capturas incluyen evidencia inequívoca de que yo soy el autor de las mismas y explicación detallada de la creación del programa profundizando en la lógica y en el apartado correspondiente al programa elaborando un informe del mismo.

BancoApp:

**Informe del Programa BancoApp**

**Importaciones**

Primero, importo las clases necesarias para manejar archivos (java.io.\*), leer la entrada del usuario (java.util.Scanner), manejar errores de entrada (java.util.InputMismatchException), y manejar listas dinámicas (java.util.ArrayList).

**Constantes y Variables**

Defino la ruta del archivo donde se guardará la cuenta (FILE\_PATH) y la lista de clientes (CLIENTES\_FILE\_PATH). También defino constantes para los colores ANSI que usaré en la consola (ANSI\_RESET, ANSI\_GREEN, ANSI\_RED, ANSI\_YELLOW).

**Método main**

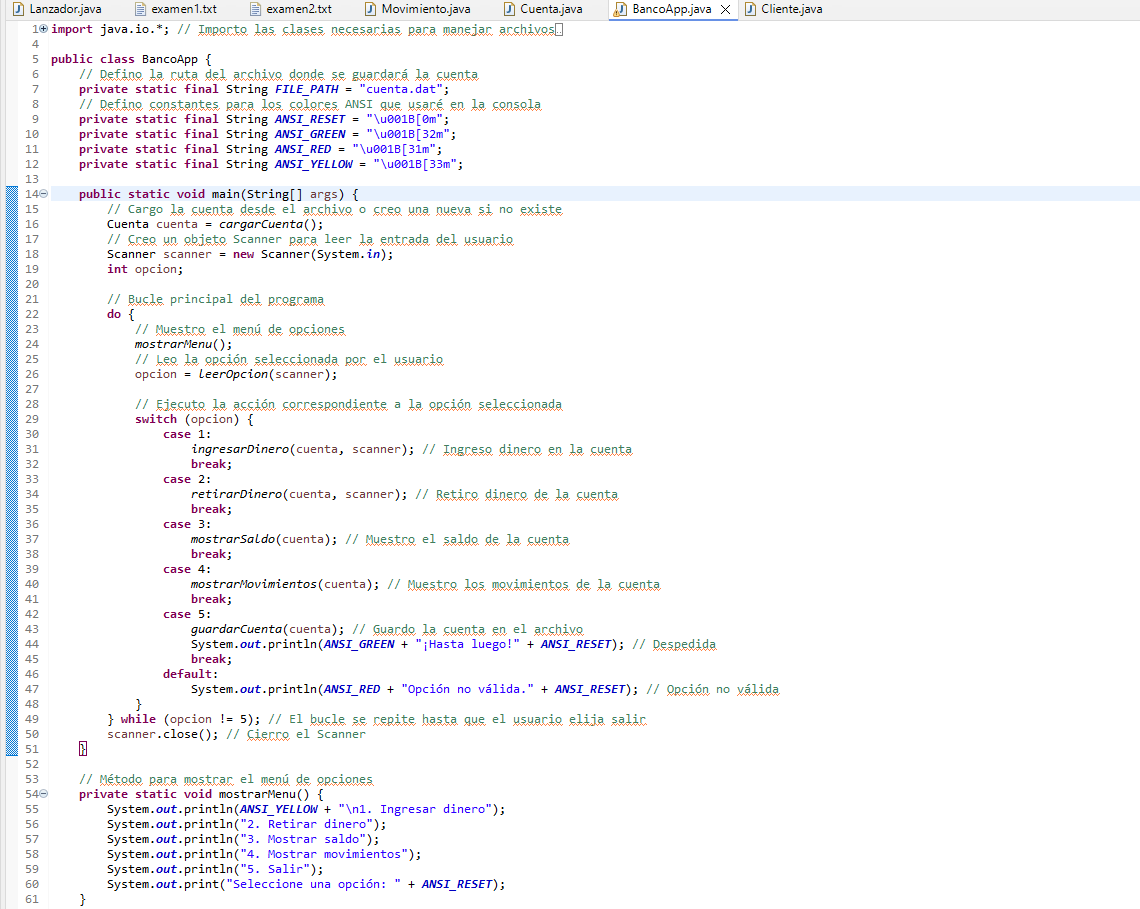
1. **Cargar Clientes**: Cargo la lista de clientes desde el archivo o creo una nueva lista si no existe.
2. **Crear Scanner**: Creo un objeto Scanner para leer la entrada del usuario.
3. **Seleccionar Cliente**: Solicito al usuario que seleccione un cliente existente o cree uno nuevo.
4. **Cargar Cuenta**: Cargo la cuenta del cliente seleccionado desde el archivo o creo una nueva si no existe.
5. **Bucle Principal**:
   * Muestro el menú de opciones.
   * Leo la opción seleccionada por el usuario.
   * Ejecuto la acción correspondiente a la opción seleccionada:
     + **Ingresar Dinero**: Llamo al método ingresarDinero.
     + **Retirar Dinero**: Llamo al método retirarDinero.
     + **Mostrar Saldo**: Llamo al método mostrarSaldo.
     + **Mostrar Movimientos**: Llamo al método mostrarMovimientos.
     + **Ver Clientes**: Llamo al método mostrarClientes.
     + **Salir**: Guardo la cuenta y la lista de clientes en los archivos y muestro un mensaje de despedida.
6. **Cerrar Scanner**: Cierro el objeto Scanner.

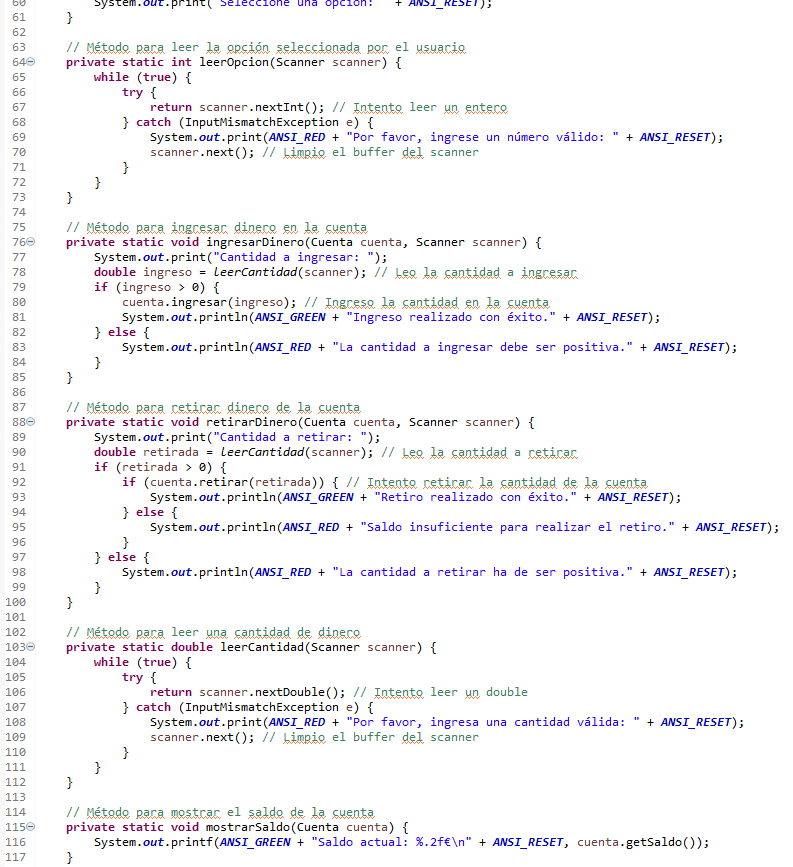
**Métodos Auxiliares**

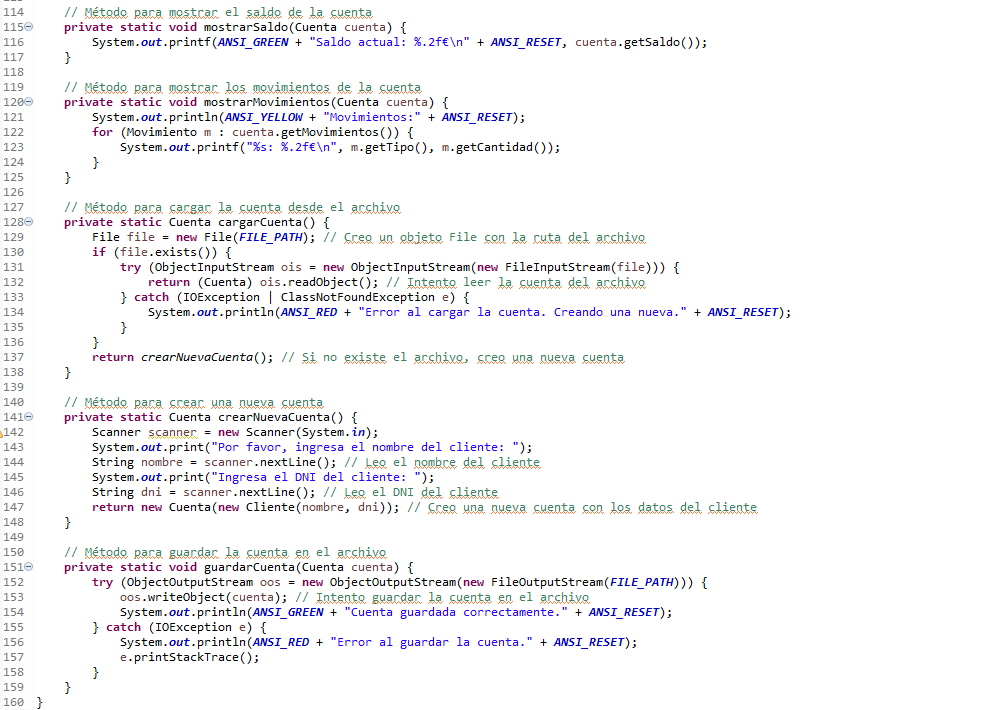
1. **mostrarMenu**: Muestra el menú de opciones en la consola.
2. **leerOpcion**: Lee la opción seleccionada por el usuario y maneja errores de entrada.
3. **ingresarDinero**: Solicita una cantidad de dinero al usuario y la ingresa en la cuenta.
4. **retirarDinero**: Solicita una cantidad de dinero al usuario y la retira de la cuenta si hay suficiente saldo.
5. **leerCantidad**: Lee una cantidad de dinero ingresada por el usuario y maneja errores de entrada.
6. **mostrarSaldo**: Muestra el saldo actual de la cuenta.
7. **mostrarMovimientos**: Muestra los movimientos de la cuenta.
8. **mostrarClientes**: Muestra la lista de clientes.
9. **cargarCuenta**: Carga la cuenta desde el archivo o crea una nueva si no existe.
10. **crearNuevaCuenta**: Crea una nueva cuenta solicitando los datos del cliente.
11. **guardarCuenta**: Guarda la cuenta en el archivo.
12. **cargarClientes**: Carga la lista de clientes desde el archivo o crea una nueva si no existe.
13. **guardarClientes**: Guarda la lista de clientes en el archivo.
14. **seleccionarCliente**: Permite al usuario seleccionar un cliente existente o crear uno nuevo.
15. **crearNuevoCliente**: Crea un nuevo cliente solicitando los datos al usuario.

**Conclusión**

El programa BancoApp permite gestionar cuentas bancarias de múltiples clientes. Los usuarios pueden seleccionar un cliente existente o crear uno nuevo, ingresar y retirar dinero, ver el saldo y los movimientos de la cuenta, y ver la lista de clientes. Toda la información se guarda en archivos para su persistencia.







Cliente

**Informe de la Clase Cliente**

**Importaciones**

Primero, importo la interfaz Serializable para que los objetos de esta clase se puedan guardar en archivos.

**Declaración de la Clase**

La clase Cliente implementa la interfaz Serializable, lo que permite que sus objetos se puedan serializar y deserializar, es decir, guardar y recuperar desde archivos.

**Variables de Instancia**

* **nombre**: Una cadena que almacena el nombre del cliente.
* **dni**: Una cadena que almacena el DNI (Documento Nacional de Identidad) del cliente.

**Constructor**

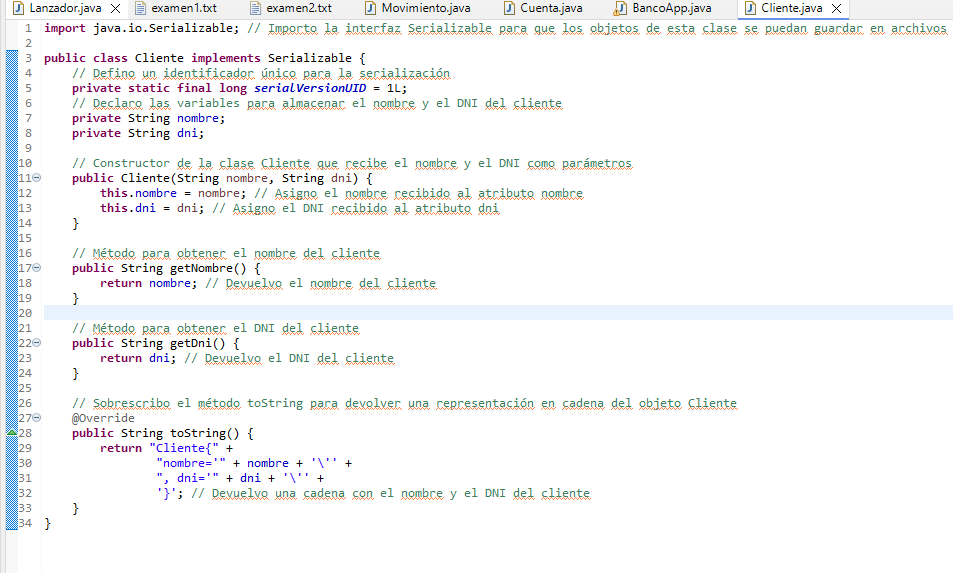
El constructor de la clase Cliente recibe dos parámetros: nombre y dni. Estos parámetros se asignan a las variables de instancia correspondientes.

**Métodos**

1. **getNombre**: Este método devuelve el nombre del cliente.
2. **getDni**: Este método devuelve el DNI del cliente.
3. **toString**: Sobrescribo el método toString para devolver una representación en cadena del objeto Cliente. La cadena incluye el nombre y el DNI del cliente en un formato legible.

**Conclusión**

La clase Cliente es una representación simple de un cliente con un nombre y un DNI. Implementa la interfaz Serializable para permitir que los objetos de esta clase se puedan guardar en archivos. Los métodos getNombre y getDni permiten acceder a los datos del cliente, y el método toString proporciona una representación en cadena del objeto para facilitar su visualización.



Cuenta

**Informe de la Clase Cuenta**

**Importaciones**

Primero, importo la interfaz Serializable para que los objetos de esta clase se puedan guardar en archivos. También importo la clase ArrayList para manejar listas dinámicas.

**Declaración de la Clase**

La clase Cuenta implementa la interfaz Serializable, lo que permite que sus objetos se puedan serializar y deserializar, es decir, guardar y recuperar desde archivos.

**Variables de Instancia**

* **cliente**: Un objeto de la clase Cliente que almacena el cliente asociado a la cuenta.
* **saldo**: Una variable de tipo double que almacena el saldo de la cuenta.
* **movimientos**: Una lista de objetos Movimiento que almacena los movimientos de la cuenta.

**Constructor**

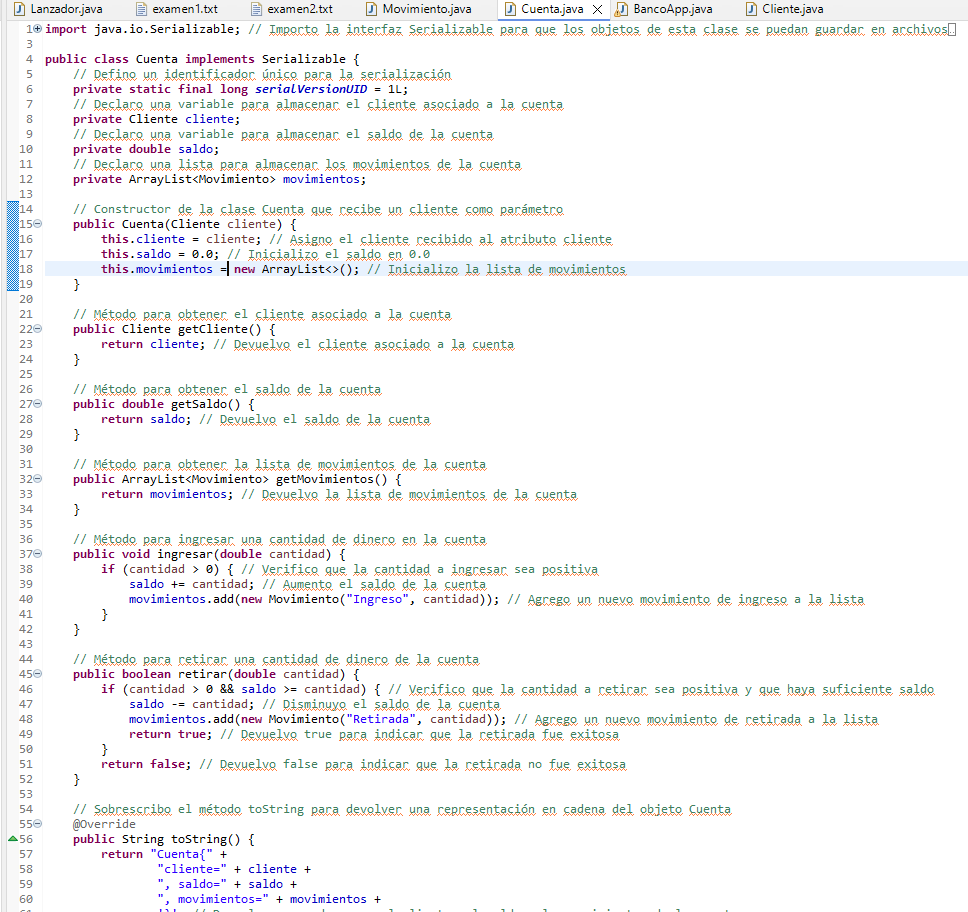
El constructor de la clase Cuenta recibe un parámetro: cliente. Este parámetro se asigna a la variable de instancia correspondiente. El saldo se inicializa en 0.0 y la lista de movimientos se inicializa como una nueva lista vacía.

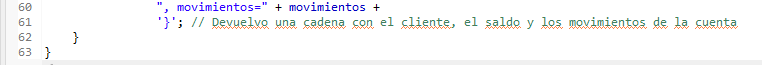
**Métodos**

1. **getCliente**: Este método devuelve el cliente asociado a la cuenta.
2. **getSaldo**: Este método devuelve el saldo de la cuenta.
3. **getMovimientos**: Este método devuelve la lista de movimientos de la cuenta.
4. **ingresar**: Este método recibe una cantidad de dinero como parámetro. Si la cantidad es positiva, se añade al saldo de la cuenta y se registra un nuevo movimiento de tipo "Ingreso".
5. **retirar**: Este método recibe una cantidad de dinero como parámetro. Si la cantidad es positiva y hay suficiente saldo en la cuenta, se resta del saldo y se registra un nuevo movimiento de tipo "Retirada". Devuelve true si la operación fue exitosa y false en caso contrario.
6. **toString**: Sobrescribo el método toString para devolver una representación en cadena del objeto Cuenta. La cadena incluye el cliente, el saldo y los movimientos de la cuenta en un formato legible.

**Conclusión**

La clase Cuenta es una representación de una cuenta bancaria asociada a un cliente. Implementa la interfaz Serializable para permitir que los objetos de esta clase se puedan guardar en archivos. Los métodos getCliente, getSaldo y getMovimientos permiten acceder a los datos de la cuenta. Los métodos ingresar y retirar permiten modificar el saldo de la cuenta y registrar los movimientos correspondientes. El método toString proporciona una representación en cadena del objeto para facilitar su visualización.





Movimiento:

**Informe de la Clase Movimiento**

**Importaciones**

Primero, importo la interfaz Serializable para que los objetos de esta clase se puedan guardar en archivos.

**Declaración de la Clase**

La clase Movimiento implementa la interfaz Serializable, lo que permite que sus objetos se puedan serializar y deserializar, es decir, guardar y recuperar desde archivos.

**Variables de Instancia**

* **tipo**: Una cadena que almacena el tipo de movimiento (por ejemplo, "Ingreso" o "Retirada").
* **cantidad**: Una variable de tipo double que almacena la cantidad del movimiento.

**Constructor**

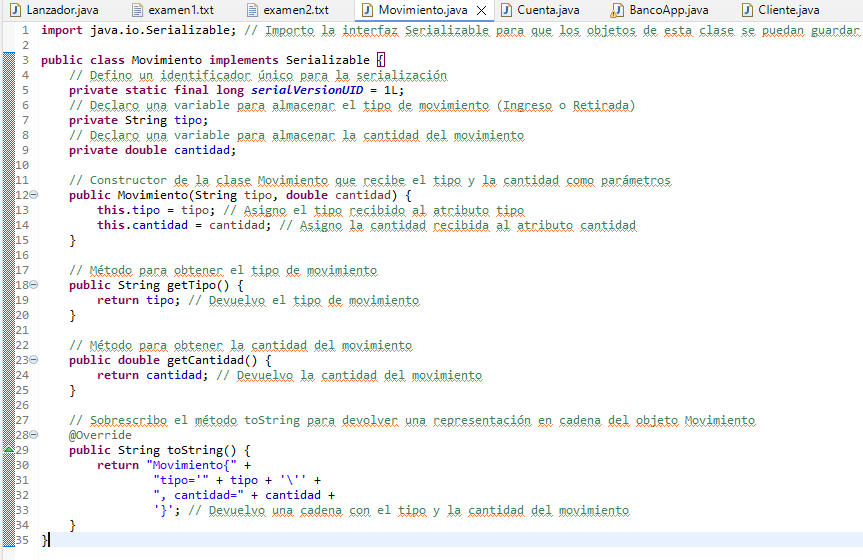
El constructor de la clase Movimiento recibe dos parámetros: tipo y cantidad. Estos parámetros se asignan a las variables de instancia correspondientes.

**Métodos**

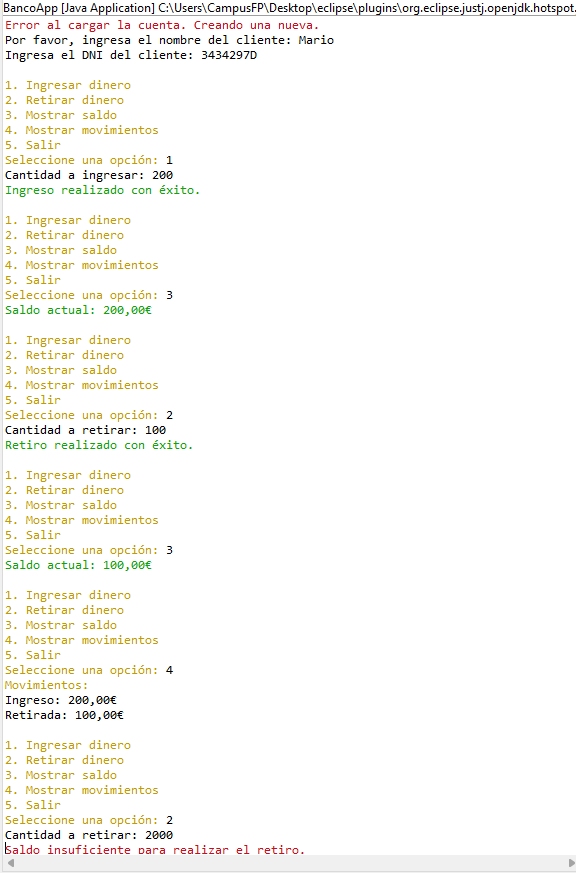
1. **getTipo**: Este método devuelve el tipo de movimiento.
2. **getCantidad**: Este método devuelve la cantidad del movimiento.
3. **toString**: Sobrescribo el método toString para devolver una representación en cadena del objeto Movimiento. La cadena incluye el tipo y la cantidad del movimiento en un formato legible.

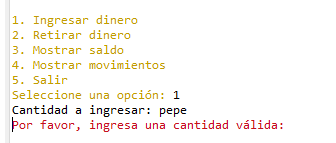
**Conclusión**

La clase Movimiento es una representación de un movimiento bancario, que puede ser un ingreso o una retirada de dinero. Implementa la interfaz Serializable para permitir que los objetos de esta clase se puedan guardar en archivos. Los métodos getTipo y getCantidad permiten acceder a los datos del movimiento. El método toString proporciona una representación en cadena del objeto para facilitar su visualización.



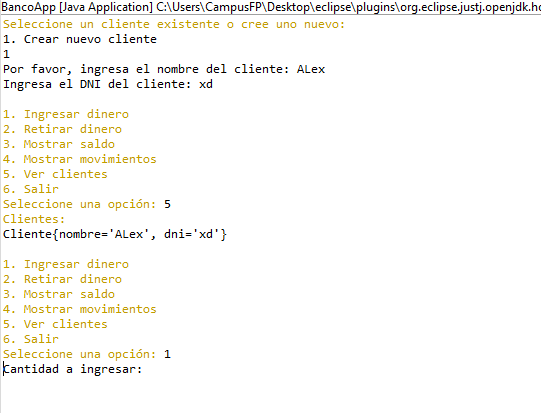
Prueba de funcionamiento y ejecución

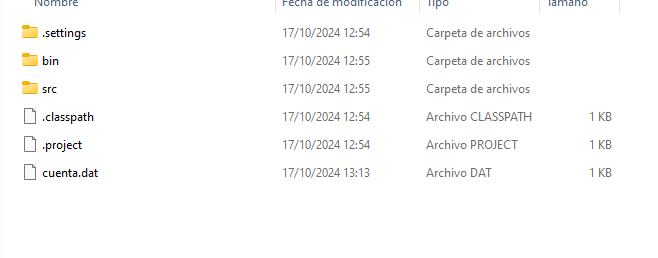




Con los cambios realizados para ver los clientes creados, existentes y la creación de los mismos, ha sido cambiado para mostrar la funcionalidad de la misma y que sea más accesible,

He aquí una prueba de la misma, en el mismo archivo ha sido modificado el código del mismo:



Se crea Cuenta.dat correctamente  


# Conclusión

Orientada a Objetos y técnicas de persistencia de datos en Java. He logrado cumplir con todos los requisitos especificados y he implementado una solución robusta y funcional.

# Bibliografía

* Java SE Documentation. (n.d.). Retrieved from [https://docs.oracle.com/javase/8/docs/](vscode-file://vscode-app/c:/Users/CampusFP/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.esm.html)
* Normas APA. (n.d.). Retrieved from [https://normas-apa.org/](vscode-file://vscode-app/c:/Users/CampusFP/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.esm.html)
* Oracle. (n.d.). Java Object Serialization Specification. Retrieved from [https://docs.oracle.com/javase/8/docs/platform/serialization/spec/serial-arch.html](vscode-file://vscode-app/c:/Users/CampusFP/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.esm.html)

El documento ha sido elaborado siguiendo las especificaciones y rúbrica proporcionadas, asegurándome de que se cubren todos los aspectos requeridos para la correcta evaluación del proyecto.

# Anexo I

## Código

import java.io.\*; // Importo las clases necesarias para manejar archivos

import java.util.Scanner; // Importo la clase Scanner para leer la entrada del usuario

import java.util.InputMismatchException; // Importo la excepción para manejar errores de entrada

public class BancoApp {

// Defino la ruta del archivo donde se guardará la cuenta

private static final String FILE\_PATH = "cuenta.dat";

// Defino constantes para los colores ANSI que usaré en la consola

private static final String ANSI\_RESET = "\u001B[0m";

private static final String ANSI\_GREEN = "\u001B[32m";

private static final String ANSI\_RED = "\u001B[31m";

private static final String ANSI\_YELLOW = "\u001B[33m";

public static void main(String[] args) {

// Cargo la cuenta desde el archivo o creo una nueva si no existe

Cuenta cuenta = cargarCuenta();

// Creo un objeto Scanner para leer la entrada del usuario

Scanner scanner = new Scanner(System.in);

int opcion;

// Bucle principal del programa

do {

// Muestro el menú de opciones

mostrarMenu();

// Leo la opción seleccionada por el usuario

opcion = leerOpcion(scanner);

// Ejecuto la acción correspondiente a la opción seleccionada

switch (opcion) {

case 1:

ingresarDinero(cuenta, scanner); // Ingreso dinero en la cuenta

break;

case 2:

retirarDinero(cuenta, scanner); // Retiro dinero de la cuenta

break;

case 3:

mostrarSaldo(cuenta); // Muestro el saldo de la cuenta

break;

case 4:

mostrarMovimientos(cuenta); // Muestro los movimientos de la cuenta

break;

case 5:

guardarCuenta(cuenta); // Guardo la cuenta en el archivo

System.out.println(ANSI\_GREEN + "¡Hasta luego!" + ANSI\_RESET); // Despedida

break;

default:

System.out.println(ANSI\_RED + "Opción no válida." + ANSI\_RESET); // Opción no válida

}

} while (opcion != 5); // El bucle se repite hasta que el usuario elija salir

scanner.close(); // Cierro el Scanner

}

// Método para mostrar el menú de opciones

private static void mostrarMenu() {

System.out.println(ANSI\_YELLOW + "\n1. Ingresar dinero");

System.out.println("2. Retirar dinero");

System.out.println("3. Mostrar saldo");

System.out.println("4. Mostrar movimientos");

System.out.println("5. Salir");

System.out.print("Seleccione una opción: " + ANSI\_RESET);

}

// Método para leer la opción seleccionada por el usuario

private static int leerOpcion(Scanner scanner) {

while (true) {

try {

return scanner.nextInt(); // Intento leer un entero

} catch (InputMismatchException e) {

System.out.print(ANSI\_RED + "Por favor, ingrese un número válido: " + ANSI\_RESET);

scanner.next(); // Limpio el buffer del scanner

}

}

}

// Método para ingresar dinero en la cuenta

private static void ingresarDinero(Cuenta cuenta, Scanner scanner) {

System.out.print("Cantidad a ingresar: ");

double ingreso = leerCantidad(scanner); // Leo la cantidad a ingresar

if (ingreso > 0) {

cuenta.ingresar(ingreso); // Ingreso la cantidad en la cuenta

System.out.println(ANSI\_GREEN + "Ingreso realizado con éxito." + ANSI\_RESET);

} else {

System.out.println(ANSI\_RED + "La cantidad a ingresar debe ser positiva." + ANSI\_RESET);

}

}

// Método para retirar dinero de la cuenta

private static void retirarDinero(Cuenta cuenta, Scanner scanner) {

System.out.print("Cantidad a retirar: ");

double retirada = leerCantidad(scanner); // Leo la cantidad a retirar

if (retirada > 0) {

if (cuenta.retirar(retirada)) { // Intento retirar la cantidad de la cuenta

System.out.println(ANSI\_GREEN + "Retiro realizado con éxito." + ANSI\_RESET);

} else {

System.out.println(ANSI\_RED + "Saldo insuficiente para realizar el retiro." + ANSI\_RESET);

}

} else {

System.out.println(ANSI\_RED + "La cantidad a retirar ha de ser positiva." + ANSI\_RESET);

}

}

// Método para leer una cantidad de dinero

private static double leerCantidad(Scanner scanner) {

while (true) {

try {

return scanner.nextDouble(); // Intento leer un double

} catch (InputMismatchException e) {

System.out.print(ANSI\_RED + "Por favor, ingresa una cantidad válida: " + ANSI\_RESET);

scanner.next(); // Limpio el buffer del scanner

}

}

}

// Método para mostrar el saldo de la cuenta

private static void mostrarSaldo(Cuenta cuenta) {

System.out.printf(ANSI\_GREEN + "Saldo actual: %.2f€\n" + ANSI\_RESET, cuenta.getSaldo());

}

// Método para mostrar los movimientos de la cuenta

private static void mostrarMovimientos(Cuenta cuenta) {

System.out.println(ANSI\_YELLOW + "Movimientos:" + ANSI\_RESET);

for (Movimiento m : cuenta.getMovimientos()) {

System.out.printf("%s: %.2f€\n", m.getTipo(), m.getCantidad());

}

}

// Método para cargar la cuenta desde el archivo

private static Cuenta cargarCuenta() {

File file = new File(FILE\_PATH); // Creo un objeto File con la ruta del archivo

if (file.exists()) {

try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream(file))) {

return (Cuenta) ois.readObject(); // Intento leer la cuenta del archivo

} catch (IOException | ClassNotFoundException e) {

System.out.println(ANSI\_RED + "Error al cargar la cuenta. Creando una nueva." + ANSI\_RESET);

}

}

return crearNuevaCuenta(); // Si no existe el archivo, creo una nueva cuenta

}

// Método para crear una nueva cuenta

private static Cuenta crearNuevaCuenta() {

Scanner scanner = new Scanner(System.in);

System.out.print("Por favor, ingresa el nombre del cliente: ");

String nombre = scanner.nextLine(); // Leo el nombre del cliente

System.out.print("Ingresa el DNI del cliente: ");

String dni = scanner.nextLine(); // Leo el DNI del cliente

return new Cuenta(new Cliente(nombre, dni)); // Creo una nueva cuenta con los datos del cliente

}

// Método para guardar la cuenta en el archivo

private static void guardarCuenta(Cuenta cuenta) {

try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(FILE\_PATH))) {

oos.writeObject(cuenta); // Intento guardar la cuenta en el archivo

System.out.println(ANSI\_GREEN + "Cuenta guardada correctamente." + ANSI\_RESET);

} catch (IOException e) {

System.out.println(ANSI\_RED + "Error al guardar la cuenta." + ANSI\_RESET);

e.printStackTrace();

}

}

}

**import** java.io.Serializable; // Importo la interfaz Serializable para que los objetos de esta clase se puedan guardar en archivos

**public** **class** Cliente **implements** Serializable {

// Defino un identificador único para la serialización

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

// Declaro las variables para almacenar el nombre y el DNI del cliente

**private** String nombre;

**private** String dni;

// Constructor de la clase Cliente que recibe el nombre y el DNI como parámetros

**public** Cliente(String nombre, String dni) {

**this**.nombre = nombre; // Asigno el nombre recibido al atributo nombre

**this**.dni = dni; // Asigno el DNI recibido al atributo dni

}

// Método para obtener el nombre del cliente

**public** String getNombre() {

**return** nombre; // Devuelvo el nombre del cliente

}

// Método para obtener el DNI del cliente

**public** String getDni() {

**return** dni; // Devuelvo el DNI del cliente

}

// Sobrescribo el método toString para devolver una representación en cadena del objeto Cliente

@Override

**public** String toString() {

**return** "Cliente{" +

"nombre='" + nombre + '\'' +

", dni='" + dni + '\'' +

'}'; // Devuelvo una cadena con el nombre y el DNI del cliente

}

}

**return** "Cliente{" +

"nombre='" + nombre + '\'' +

", dni='" + dni + '\'' +

'}'; // Devuelvo una cadena con el nombre y el DNI del cliente

}

}

import java.io.Serializable; // Importo la interfaz Serializable para que los objetos de esta clase se puedan guardar en archivos

import java.util.ArrayList; // Importo la clase ArrayList para manejar listas dinámicas

public class Cuenta implements Serializable {

// Defino un identificador único para la serialización

private static final long serialVersionUID = 1L;

// Declaro una variable para almacenar el cliente asociado a la cuenta

private Cliente cliente;

// Declaro una variable para almacenar el saldo de la cuenta

private double saldo;

// Declaro una lista para almacenar los movimientos de la cuenta

private ArrayList<Movimiento> movimientos;

// Constructor de la clase Cuenta que recibe un cliente como parámetro

public Cuenta(Cliente cliente) {

this.cliente = cliente; // Asigno el cliente recibido al atributo cliente

this.saldo = 0.0; // Inicializo el saldo en 0.0

this.movimientos = new ArrayList<>(); // Inicializo la lista de movimientos

}

// Método para obtener el cliente asociado a la cuenta

public Cliente getCliente() {

return cliente; // Devuelvo el cliente asociado a la cuenta

}

// Método para obtener el saldo de la cuenta

public double getSaldo() {

return saldo; // Devuelvo el saldo de la cuenta

}

// Método para obtener la lista de movimientos de la cuenta

public ArrayList<Movimiento> getMovimientos() {

return movimientos; // Devuelvo la lista de movimientos de la cuenta

}

// Método para ingresar una cantidad de dinero en la cuenta

public void ingresar(double cantidad) {

if (cantidad > 0) { // Verifico que la cantidad a ingresar sea positiva

saldo += cantidad; // Aumento el saldo de la cuenta

movimientos.add(new Movimiento("Ingreso", cantidad)); // Agrego un nuevo movimiento de ingreso a la lista

}

}

// Método para retirar una cantidad de dinero de la cuenta

public boolean retirar(double cantidad) {

if (cantidad > 0 && saldo >= cantidad) { // Verifico que la cantidad a retirar sea positiva y que haya suficiente saldo

saldo -= cantidad; // Disminuyo el saldo de la cuenta

movimientos.add(new Movimiento("Retirada", cantidad)); // Agrego un nuevo movimiento de retirada a la lista

return true; // Devuelvo true para indicar que la retirada fue exitosa

}

return false; // Devuelvo false para indicar que la retirada no fue exitosa

}

// Sobrescribo el método toString para devolver una representación en cadena del objeto Cuenta

@Override

public String toString() {

return "Cuenta{" +

"cliente=" + cliente +

", saldo=" + saldo +

", movimientos=" + movimientos +

'}'; // Devuelvo una cadena con el cliente, el saldo y los movimientos de la cuenta

}

}

**import** java.io.Serializable; // Importo la interfaz Serializable para que los objetos de esta clase se puedan guardar en archivos

**public** **class** Movimiento **implements** Serializable {

// Defino un identificador único para la serialización

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

// Declaro una variable para almacenar el tipo de movimiento (Ingreso o Retirada)

**private** String tipo;

// Declaro una variable para almacenar la cantidad del movimiento

**private** **double** cantidad;

// Constructor de la clase Movimiento que recibe el tipo y la cantidad como parámetros

**public** Movimiento(String tipo, **double** cantidad) {

**this**.tipo = tipo; // Asigno el tipo recibido al atributo tipo

**this**.cantidad = cantidad; // Asigno la cantidad recibida al atributo cantidad

}

// Método para obtener el tipo de movimiento

**public** String getTipo() {

**return** tipo; // Devuelvo el tipo de movimiento

}

// Método para obtener la cantidad del movimiento

**public** **double** getCantidad() {

**return** cantidad; // Devuelvo la cantidad del movimiento

}

// Sobrescribo el método toString para devolver una representación en cadena del objeto Movimiento

@Override

**public** String toString() {

**return** "Movimiento{" +

"tipo='" + tipo + '\'' +

", cantidad=" + cantidad +

'}'; // Devuelvo una cadena con el tipo y la cantidad del movimiento

}

}