# Trabajo Práctico 8 – Interfaces y Excepciones en Java

Tecnicatura Universitaria en Programación – UTN

Materia: Programación II

Alumno/a: Marina Giselle Cordero

Comisión: 07

## Parte 1 – Interfaces en un Sistema de E-commerce

En esta primera parte se desarrolló un sistema orientado a objetos que aplica los conceptos de interfaces, herencia múltiple, polimorfismo y delegación de responsabilidades. El objetivo es comprender cómo las interfaces permiten definir contratos de comportamiento y reducir el acoplamiento entre clases.

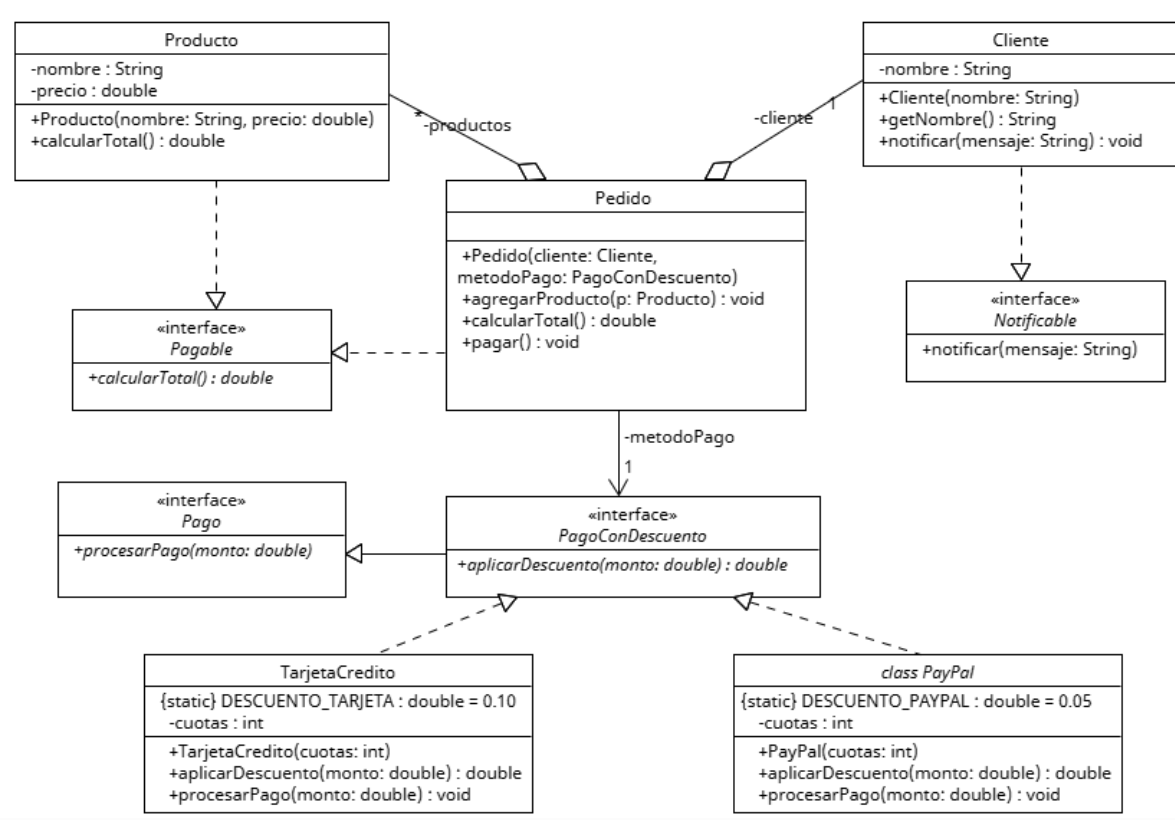
**Enlace Git Hub:**

<https://github.com/Marigi84/TP8-Interfaces-Excepciones/tree/main/Tp8-Interface-Exception/src/tp8/pkginterface/exception>

## Desarrollo del ejercicio

El programa simula un sistema de comercio electrónico donde un cliente realiza pedidos que contienen productos y los abona utilizando distintos medios de pago. Cada medio de pago aplica o no un descuento según la cantidad de cuotas. Se utilizaron constantes para los valores de descuento, mejorando la legibilidad y el mantenimiento del código.

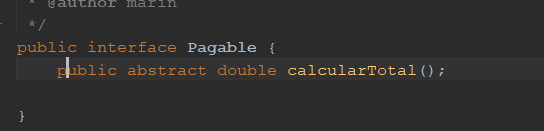
## Diagramas UML



## Código fuente

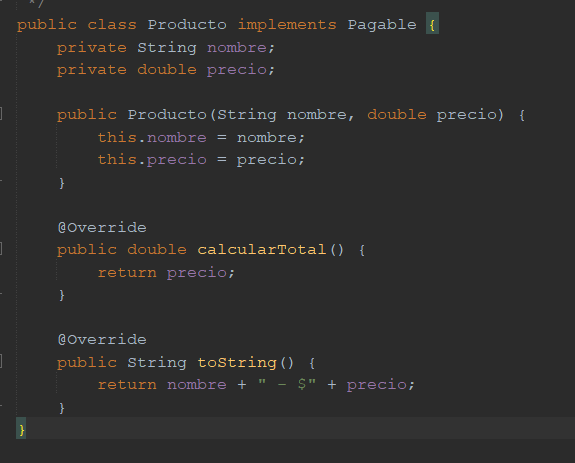
[Espacio para insertar capturas del código en NetBeans de las clases e interfaces]

1. **Interface Pagable**



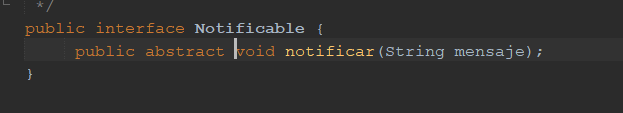
Define un contrato para los objetos que pueden calcular un monto total.  
Es implementada por las clases Producto y Pedido.  
Permite aplicar polimorfismo en el cálculo del total, independientemente del tipo de objeto.

1. **Clase Producto**



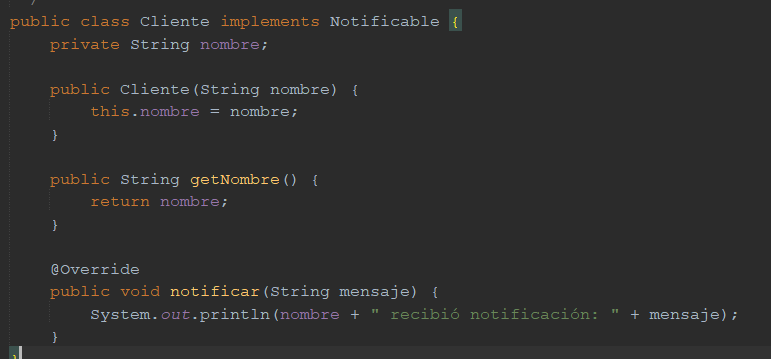
Representa un producto dentro del sistema con su nombre y precio.  
Implementa la interfaz Pagable, sobrescribiendo el método calcularTotal().  
Su responsabilidad es devolver el precio total del producto.

1. **Interface Notificable**

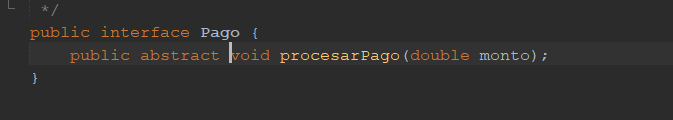


Define el comportamiento que deben cumplir los objetos capaces de recibir notificaciones.  
Contiene el método notificar(String mensaje), utilizado para enviar avisos al cliente.

1. **Clase Cliente**

Representa a un cliente del sistema.  
Implementa la interfaz Notificable, ya que puede recibir mensajes sobre su pedido.  
Guarda el nombre del cliente y define cómo se muestra la notificación.

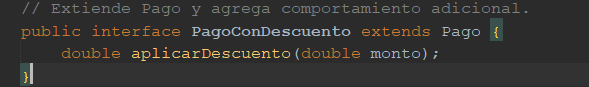
1. **Interface Pago**



Establece la estructura básica de cualquier tipo de pago dentro del sistema.

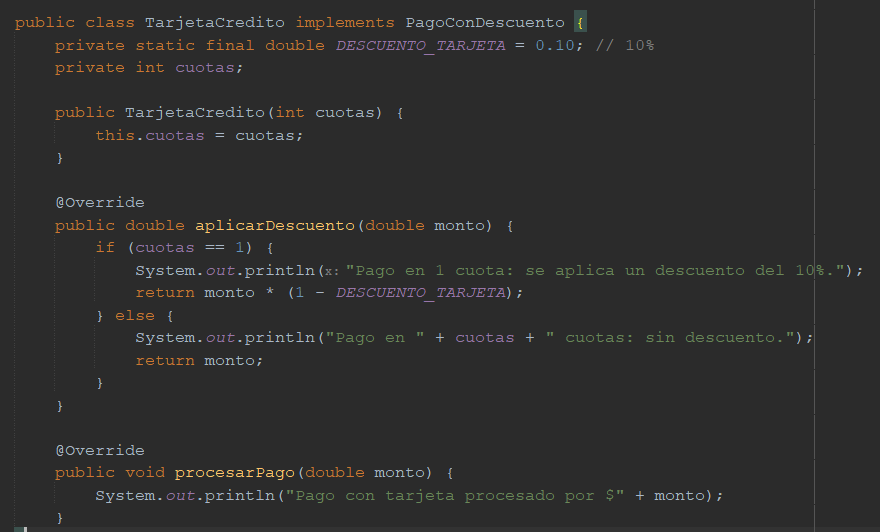
Contiene el método procesarPago(double monto), que cada medio de pago debe implementar.  
Es la interfaz base de toda la jerarquía de pagos.

1. **Interface PagoConDescuento**

  
Extiende la interfaz Pago y agrega el método aplicarDescuento(double monto).

Permite definir medios de pago que pueden ofrecer descuentos según condiciones específicas.  
Es implementada por las clases TarjetaCredito y PayPal.

1. **Clase TarjetaCredito**

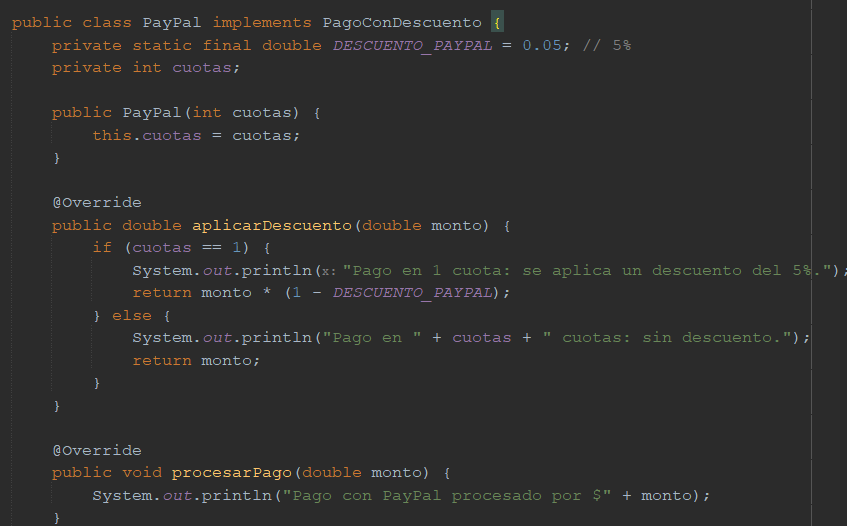
Implementa la interfaz PagoConDescuento.

Contiene un atributo cuotas y una constante DESCUENTO\_TARJETA del 10%.

Aplica descuento solo si el pago es en una sola cuota.

Implementa el método procesarPago() mostrando el monto procesado con tarjeta.

1. **Clase PayPal**

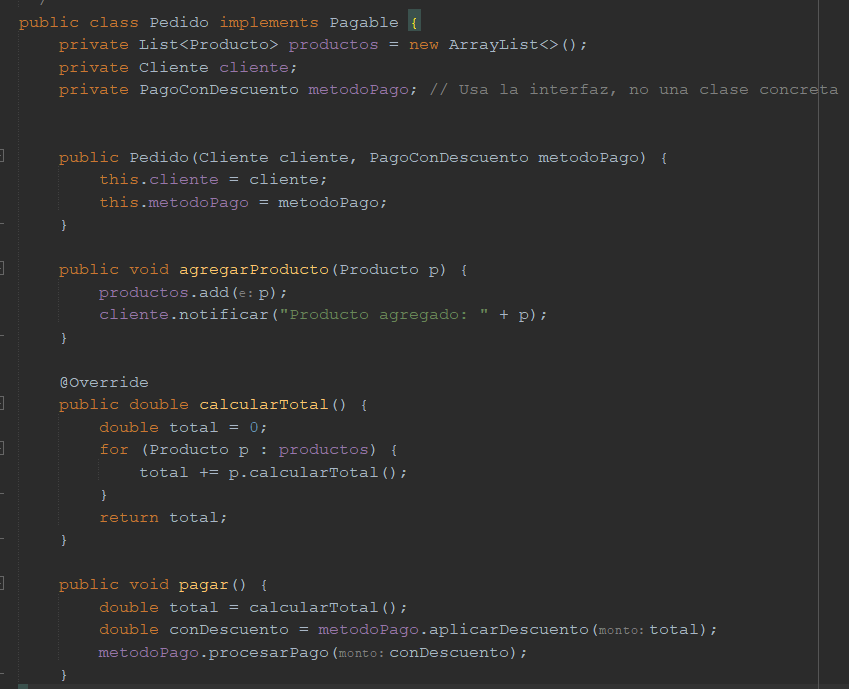
  
También implementa la interfaz PagoConDescuento.

Aplica un descuento del 5% si el pago se realiza en una sola cuota.

Implementa procesarPago() para mostrar el monto procesado mediante PayPal.

Demuestra el uso del polimorfismo en medios de pago distintos.

1. **Clase Pedido**

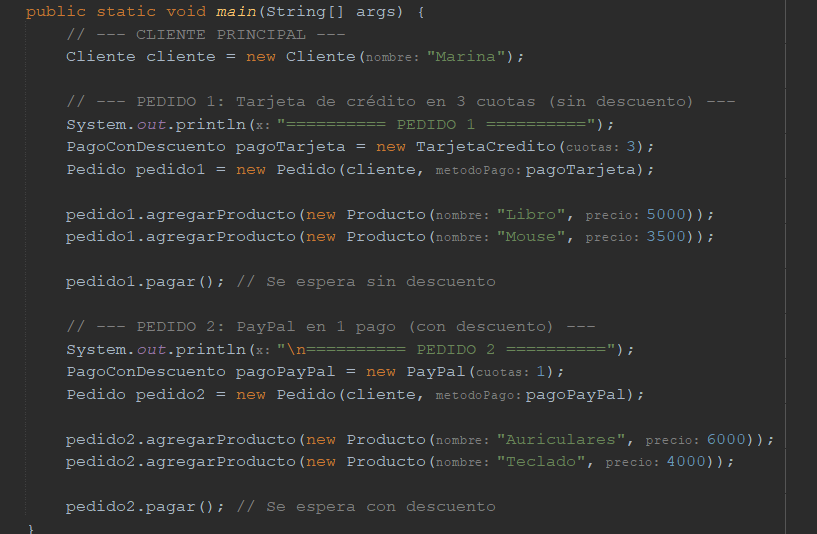
   
Representa un pedido realizado por un cliente.

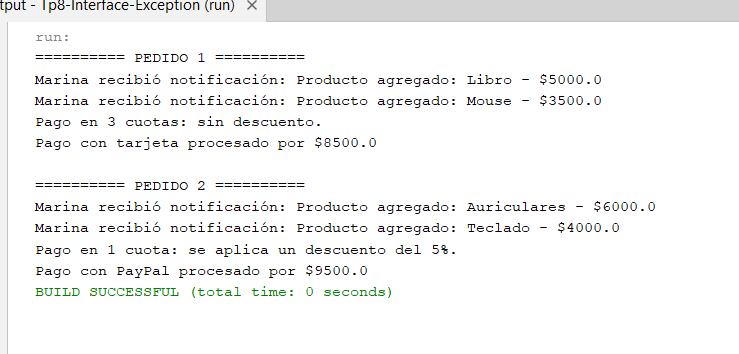
Contiene (agregación) una lista de productos (List<Producto>), un cliente (Cliente) y un medio de pago (PagoConDescuento) (asociación unilateral).

Implementa Pagable y define los métodos calcularTotal() y pagar(), delegando en el medio de pago la lógica de descuento y procesamiento.

Demuestra composición y polimorfismo en acción.

1. **Clase MainEcommerce**

Clase principal del sistema.  
Crea objetos Cliente, Producto, Pedido, TarjetaCredito y PayPal.  
Muestra la ejecución completa: carga de productos, cálculo del total, aplicación del descuento y pago final.  
Permite verificar la correcta interacción entre todas las interfaces y clases.  
  
**Ejecución del programa**



## Conclusión

A través de este ejercicio se comprendió el rol de las interfaces como mecanismos de abstracción y reutilización de código, permitiendo lograr un diseño más flexible y escalable. Además, se aplicaron buenas prácticas de programación utilizando constantes y controlando la lógica de descuentos de forma coherente y modular.

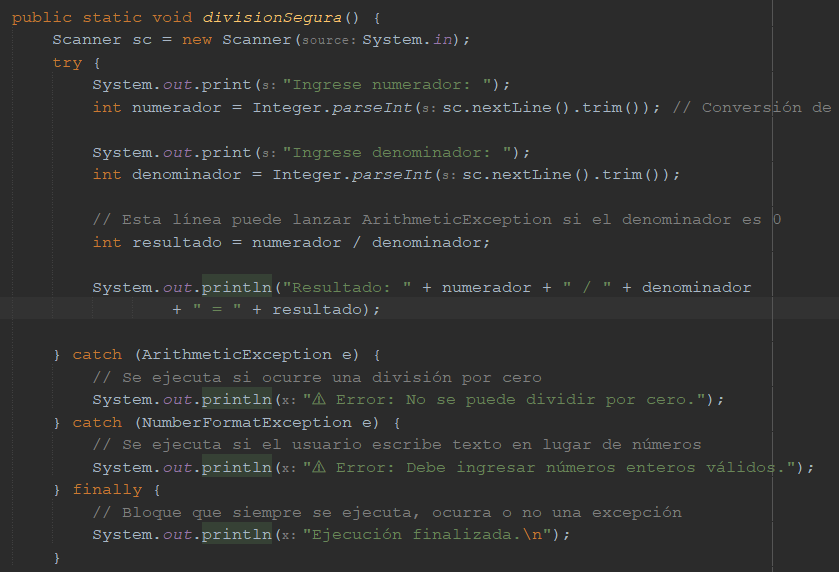
# Parte 2: Excepciones en Java

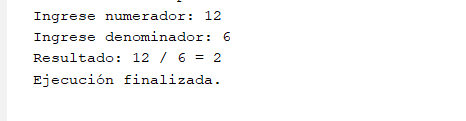
## Ejercicio 1 – División segura

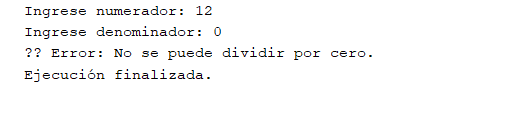
El programa solicita dos números enteros y realiza la división entre ellos. Si el divisor es cero, se lanza una ArithmeticException que se captura para evitar el error en tiempo de ejecución.

**Enlace Git Hub:**

<https://github.com/Marigi84/TP8-Interfaces-Excepciones/tree/main/Tp8.Excepcion/src/tp8/excepcion>



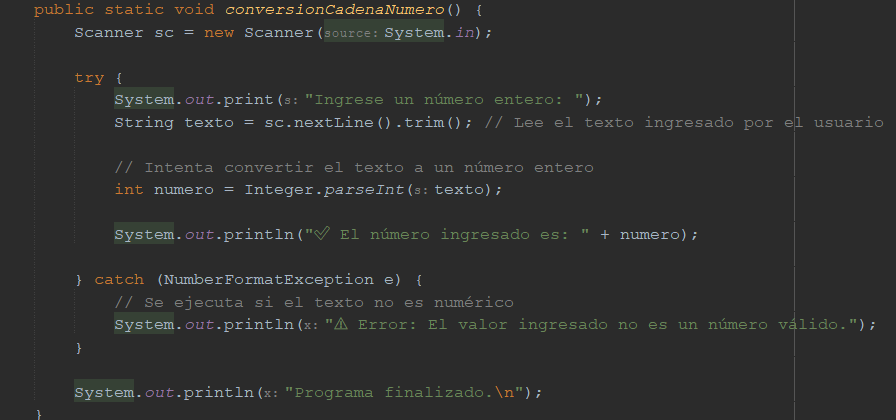


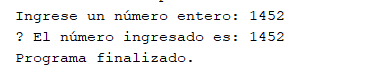


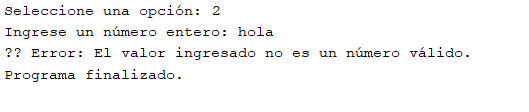
• La división puede lanzar ArithmeticException si el divisor es cero.  
• El bloque try-catch captura y maneja el error mostrando un mensaje amigable.  
• El bloque finally asegura que se muestre un mensaje de finalización.

## Ejercicio 2 – Conversión de cadena a número

El programa solicita un texto al usuario y trata de convertirlo a número entero usando Integer.parseInt(). Si el texto contiene caracteres no numéricos, se lanza y captura NumberFormatException.





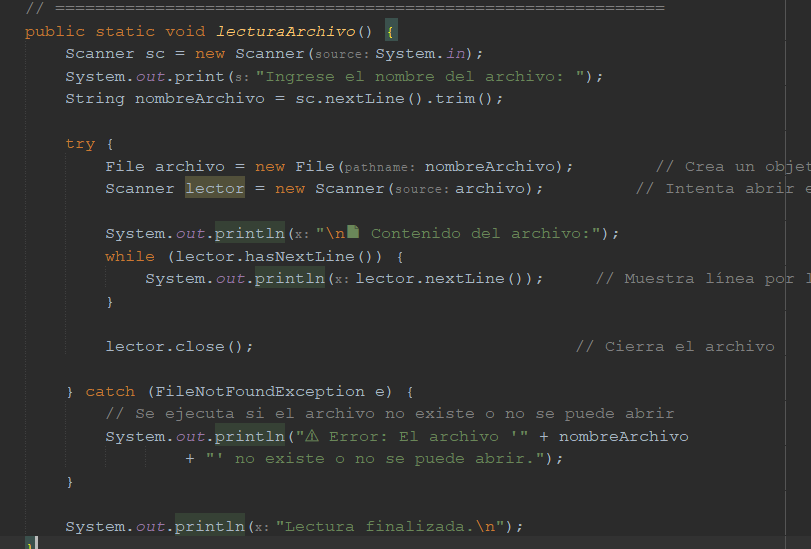


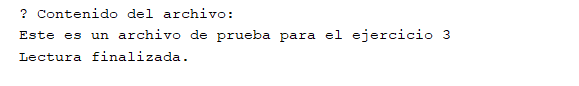
### Explicación del código

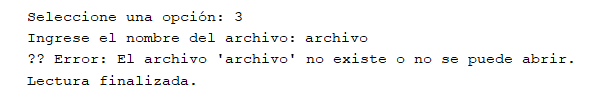
• El método parseInt() intenta convertir la cadena a int.  
• Si el texto no es numérico, se lanza NumberFormatException.  
• El bloque catch maneja la excepción sin detener el programa.

## Ejercicio 3 – Lectura de archivo

Este programa solicita el nombre de un archivo de texto, intenta abrirlo con Scanner y muestra su contenido. Si el archivo no existe, se lanza FileNotFoundException y se muestra un mensaje de error controlado.





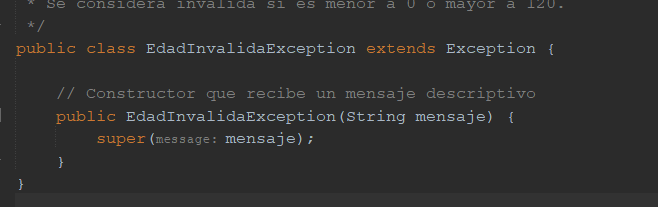


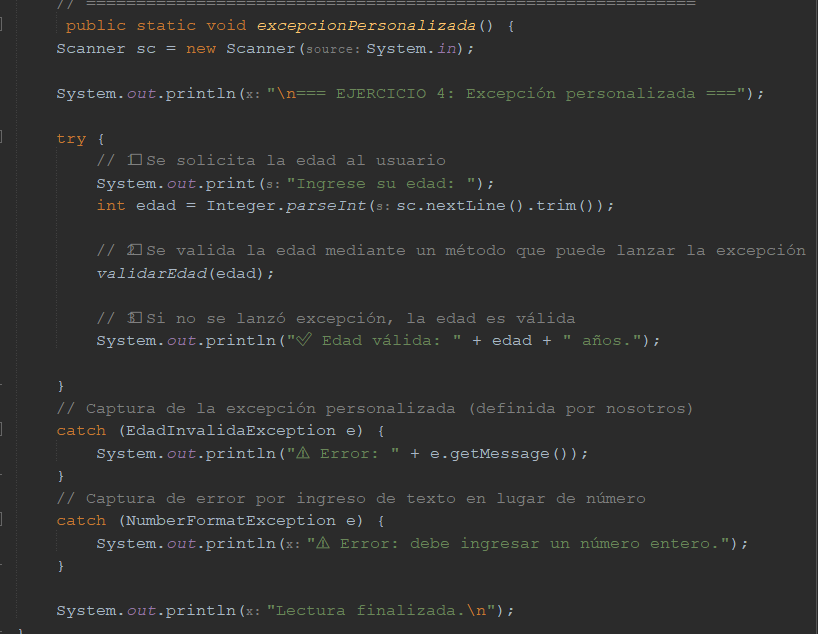
### Explicación del código

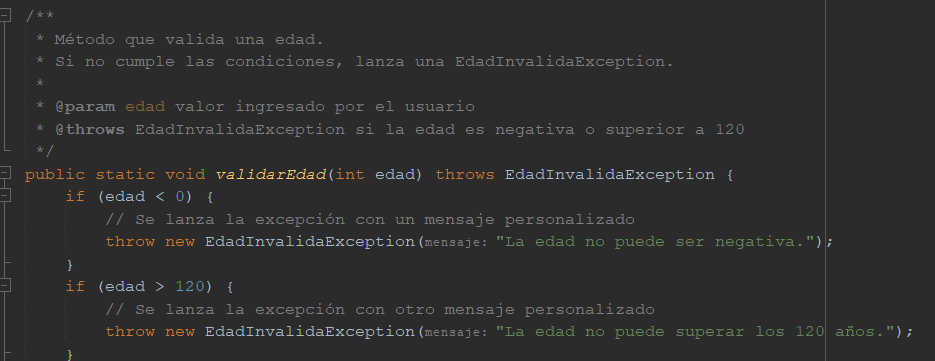
• La clase File representa al archivo físico.  
• El objeto Scanner asociado al archivo puede lanzar FileNotFoundException.  
• El catch muestra un mensaje si el archivo no existe o no puede abrirse.  
• Se recorre el archivo línea por línea con hasNextLine().

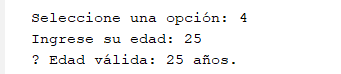
## Ejercicio 4 – Excepción personalizada: EdadInvalidaException

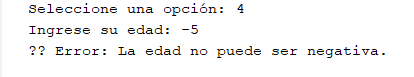
Se define una excepción propia llamada EdadInvalidaException. El programa solicita una edad y lanza esta excepción si la edad es menor a 0 o mayor a 120. También maneja NumberFormatException si el usuario no ingresa un número.

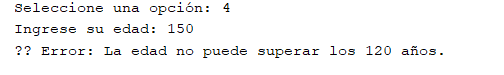










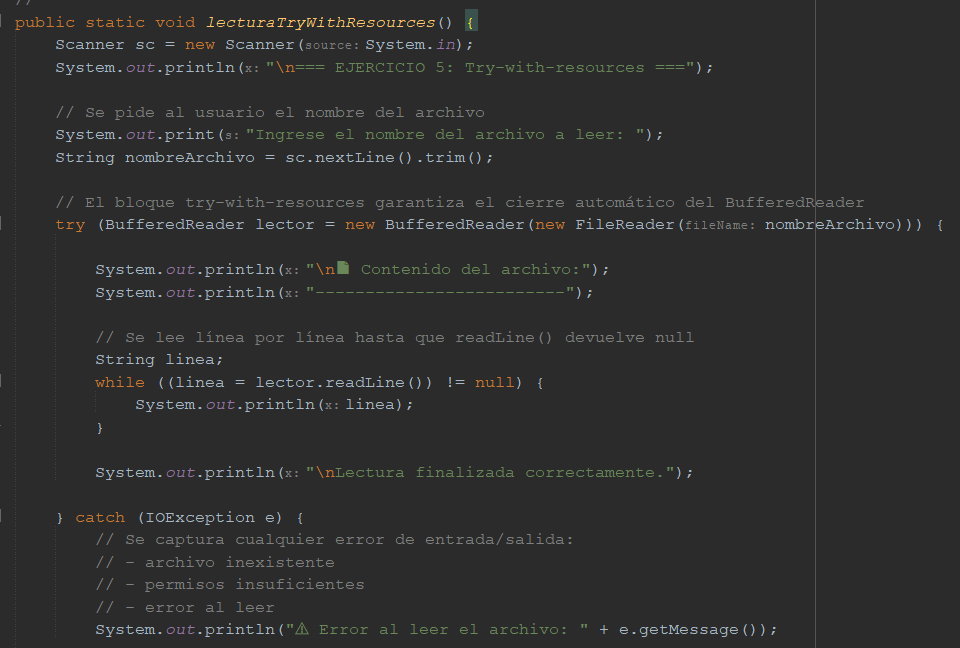


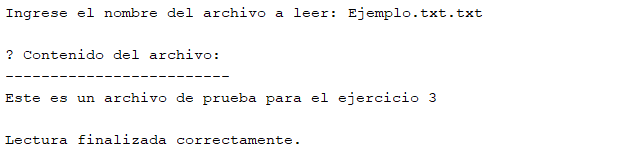
### Explicación del código

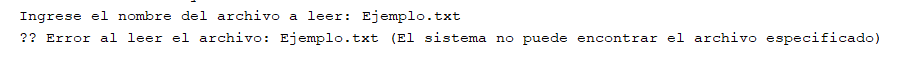
• La clase EdadInvalidaException hereda de Exception y recibe un mensaje personalizado.  
• El método validarEdad lanza esta excepción si la edad no cumple con el rango permitido.  
• El bloque try-catch maneja tanto la excepción personalizada como errores de formato numérico.

## Ejercicio 5 – Uso de try-with-resources con BufferedReader

Este programa lee un archivo usando BufferedReader dentro de un bloque try-with-resources, lo que garantiza el cierre automático del recurso. Maneja IOException correctamente.







### Explicación del código

• try-with-resources cierra automáticamente el BufferedReader al finalizar el bloque.  
• readLine() lee línea por línea hasta que retorna null.  
• IOException captura tanto errores de lectura como archivo no encontrado.

**Conclusión:**

El manejo de excepciones en Java permite desarrollar programas más seguros, robustos y predecibles.  
A través de los distintos ejercicios se demostró cómo prevenir errores comunes que pueden ocurrir durante la ejecución, como divisiones por cero, conversiones inválidas, archivos inexistentes o datos fuera de rango.

El uso de bloques try-catch garantiza que el programa no se detenga ante una situación inesperada, sino que pueda capturar el error y responder de manera controlada, informando al usuario lo sucedido.

Además, la implementación de **excepciones personalizadas** (como EdadInvalidaException) permite crear mensajes de error específicos y adaptados al contexto de la aplicación, mejorando la legibilidad y el mantenimiento del código.

Finalmente, el uso de **try-with-resources** mostró una práctica moderna para manejar archivos y otros recursos, asegurando su cierre automático y evitando fugas de memoria.

En conjunto, esta parte del trabajo permitió afianzar conceptos esenciales de programación defensiva, buenas prácticas de control de errores y diseño orientado a objetos, aportando una base sólida para el desarrollo de aplicaciones más confiables y profesionales.