PROGRAMACIÓN II

TP 8: Interfaces y Excepciones en Java

OBJETIVO GENERAL

Desarrollar habilidades en el uso de interfaces y manejo de excepciones en Java para fomentar la modularidad, flexibilidad y robustez del código. Comprender la definición e implementación de interfaces como contratos de comportamiento y su aplicación en el diseño orientado a objetos. Aplicar jerarquías de excepciones para controlar y comunicar errores de forma segura. Diferenciar entre excepciones comprobadas y no comprobadas, y utilizar bloques `try`, `catch`, `finally` y `throw` para garantizar la integridad del programa. Integrar interfaces y manejo de excepciones en el desarrollo de aplicaciones escalables y mantenibles.

Concepto	Aplicación en el proyecto
Interfaces	Definición de contratos de comportamiento común entre distintas clases
Herencia múltiple con interfaces	Permite que una clase implementa múltiples comportamientos sin herencia de estado
Implementación de interfaces	Uso de **implements** para que una clase cumpla con los métodos definidos en una interfaz
Excepciones	Manejo de errores en tiempo de ejecución mediante estructuras **try-catch**
Excepciones checked y unchecked	Diferencias y usos según la naturaleza del error

Excepciones personalizadas	Creación de nuevas clases que extienden **Exception**
finally y try-with-resources	Buenas prácticas para liberar recursos correctamente
Uso de throw y throws	Declaración y lanzamiento de excepciones
Interfaces	Definición de contratos de comportamiento común entre distintas clases
Herencia múltiple con interfaces	Permite que una clase implementa múltiples comportamientos sin herencia de estado

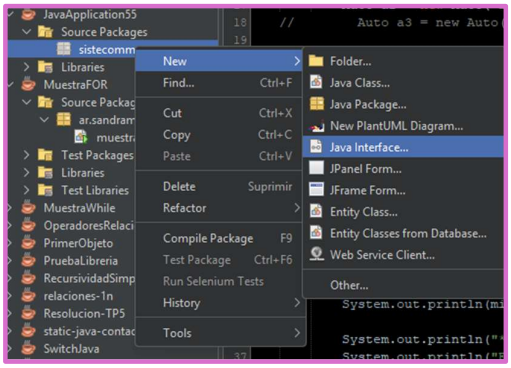
MARCO TEÓRICO
Caso Practico

Parte 1: Interfaces en un sistema de E-commerce

1. Crear una interfaz **Pagable** con el método **calcularTotal().**

Esta **interfaz** sirve para que todas las clases que tengan un “precio” o “total a pagar” compartan el mismo **método calcularTotal().**

- Como primer paso cree una clase Interface:



```java
/*
 * Click nbfs://nbhost/SystemFileSystem/Template
 * Click nbfs://nbhost/SystemFileSystem/Template
 */
package sistecomm;

/**
 *
 * @author Sandra Martinez
 */
public interface Pagable {
    double calcularTotal();

}
```

Explicacion Codigo:

public interface Pagable

- Declara una **interfaz pública** llamada Pagable.
- Una interfaz no tiene código ejecutable, solo define **qué métodos** deben tener las clases que la implementen.

double calcularTotal();

- Declara un **método abstracto** (sin cuerpo, sin llaves {}) llamado **calcularTotal.**
- Devuelve un número decimal (double), que representará el monto total.
- Cualquier clase que implemente esta interfaz deberá escribir su propia versión del método.

2. Clase **Producto**: tiene nombre y precio, implementa **Pagable.**

La **clase Producto** representa un objeto con un nombre y un precio.
Implementa la interfaz Pagable, por lo que define el método **calcularTotal(),** que devuelve su propio precio.

```java
package sistecomm;

/**
 *
 * @author Sandra Martinez
 */
    public class Producto implements Pagable {
    private String nombre;
    private double precio;

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

     Override
    public double calcularTotal() {
        return precio;
    }

    public String getNombre() {
        return nombre;
    }

    public double getPrecio() {
        return precio;
    }
```

Explicación Código:

public class Producto implements Pagable

- Crea una **clase pública** llamada Producto.
  - Usa la palabra **implements** para indicar que la clase cumple con el contrato de la interfaz Pagable.
  - Esto significa que Producto está obligada a tener el método calcularTotal().

private String nombre; y private double precio;

- Se definen dos **atributos privados**.
- nombre guarda el nombre del producto.
- precio guarda su valor numérico

public Producto(String nombre, double precio)

- Es el **constructor**.
- Permite crear un objeto de tipo **Producto** indicando el nombre y el precio.

this.nombre = nombre;
this.precio = precio;

- Asignan los valores que se reciben al crear el producto a los atributos **internos del objeto.**
- La palabra **this** sirve para diferenciar entre el nombre del atributo y el nombre del parámetro.

@Override

- Indica que el método que sigue **pertenece** a la **interfaz Pagable**.
- Es una forma de confirmar que la clase está cumpliendo con el **contrato de la interfaz.**

public double calcularTotal()

- Es el método exigido por la interfaz Pagable.
- En este caso, devuelve el precio del producto.

getNombre()
getPrecio()

- Son **métodos de acceso**
- Sirven para obtener los valores de los atributos privados nombre y precio desde fuera de la clase.

3. Clase **Pedido**: tiene una lista de productos, implementa **Pagable** y calcula el total del pedido.

```java
package sistecomm;

import java.util.ArrayList;
import java.util.List;

/**
 *
 * @author Sandra Martinez
 */

public class Pedido implements Pagable {
    private List<Producto> productos;

    public Pedido() {
        productos = new ArrayList<>();
    }

    public void agregarProducto(Producto producto) {
        productos.add(producto);
    }

     Override
    public double calcularTotal() {
        double total = 0;
        for (Producto p : productos) {
            total += p.calcularTotal();
        }
        return total;
    }
```

Explicación Código:

public class Pedido implements Pagable

- Crea una clase pública llamada Pedido.
- Usa la palabra **implements** para indicar que la clase cumple con el contrato de la interfaz **Pagable.**
- Esto significa que Pedido está **obligada** a tener el método calcularTotal().

private List<Producto> productos

- Declara un atributo privado llamado productos.
- El tipo **List<Producto>** indica que se trata de una lista que va a guardar varios objetos de tipo Producto.

public Pedido()

- Es el constructor de la clase.
- Se ejecuta cuando se crea un nuevo pedido. Dentro del constructor se inicializa la lista **con productos = new ArrayList<>();** para que empiece vacía y lista para usar.

public void agregarProducto(Producto producto)

- Es un método público que permite agregar productos al pedido.
- Recibe como parámetro un objeto de tipo Producto.
- Agrega ese producto a la lista **con productos.add(producto).**

@Override

- Indica que el método que sigue proviene de la interfaz Pagable.

- Se usa para asegurarse de que el método se está implementando correctamente.

public double calcularTotal()

- Es el método exigido por la interfaz Pagable. Crea una variable **total** que comienza en 0.
- Recorre todos los productos dentro de la lista y suma los precios obtenidos con p.calcularTotal().
- Finalmente devuelve el valor total de todos los productos del pedido.

4. Ampliar con interfaces **Pago** y **PagoConDescuento** para distintos medios de pago (**TarjetaCredito, PayPal**), con métodos **procesarPago(double)** y **aplicarDescuento(double)**.

```
 */
package sistecomm;

/**
 *
 * @author Sandra Martinez
 */
public interface Pago {
    void procesarPago(double monto);

}
```

Explicacion código:

public interface Pago

- Crea una interfaz pública llamada **Pago**.
- En este caso, la interfaz define la estructura básica para procesar un pago.

void procesarPago(double monto);

- Declara un método llamado **procesarPago** que recibe un valor numérico (monto).
- Las clases que implementen **Pago** deberán escribir su propia versión de **procesarPago(),** explicando cómo manejan el pago en cada caso.

```
package sistecomm;

/**
 *
 * @author Sandra Martinez
 */
public interface PagoConDescuento extends Pago {
    double aplicarDescuento(double monto);
}
```

Explicación código:

public interface PagoConDescuento extends Pago

- Crea una interfaz pública llamada **PagoConDescuento.**
- Usa la palabra **extends** para indicar que hereda de la interfaz Pago.
- Esto significa que **PagoConDescuento** también incluye el método **procesarPago,** pero agrega uno nuevo relacionado con descuentos.
- La idea es que las clases que implementen esta interfaz puedan procesar un pago y además aplicar un descuento.

double aplicarDescuento(double monto);

- Declara un método llamado **aplicarDescuento** que recibe un valor numérico (monto) y devuelve otro valor numérico.
- Este método servirá para calcular el monto final después de aplicar un descuento.
- Las clases que implementen PagoConDescuento deberán escribir su propia lógica para definir cuánto descuento se aplica y cómo se calcula.

```
package sistecomm;

/**
 *
 * @author Sandra Martinez
 */
public class TarjetaCredito implements PagoConDescuento {
    private String titular;
    private String numeroTarjeta;

    public TarjetaCredito(String titular, String numeroTarjeta) {
        this.titular = titular;
        this.numeroTarjeta = numeroTarjeta;
    }

     Override
    public void procesarPago(double monto) {
        System.out.println("Proceso de pago con tarjeta de crédito de " + titular + " por $" + monto);
    }

     Override
    public double aplicarDescuento(double monto) {
        double descuento = monto * 0.10;
        double montoFinal = monto - descuento;
        System.out.println("Se aplicó 10% de descuento. Monto final: $" + montoFinal);
        return montoFinal;
    }
```

Expliacacion del código:

public class TarjetaCredito implements PagoConDescuento

- Crea una clase pública llamada **TarjetaCredito.**
- Usa la palabra **implements** para indicar que cumple con el contrato de la interfaz **PagoConDescuento.**
- Esto significa que debe tener los métodos **procesarPago() y aplicarDescuento().**

private String titular;

private String numeroTarjeta;

- Declara 2 atributos privados: uno para guardar el nombre del titular de la tarjeta y otro para su número.
- Estos datos identifican la tarjeta con la que se realiza el pago.

public TarjetaCredito(String titular, String numeroTarjeta)

- Es el constructor de la clase.
- Permite crear un objeto **TarjetaCredito** indicando el nombre del titular y el número de la tarjeta.
- Asigna los valores recibidos a los atributos internos del objeto.

@Override public void procesarPago(double monto)

- Implementa el método exigido por la **interfaz Pago.**
- Muestra por pantalla un mensaje que simula el proceso del pago con tarjeta de crédito e informa el monto.

@Override public double aplicarDescuento(double monto)

- Implementa el método exigido por la **interfaz PagoConDescuento.**
- Calcula un 10% de descuento sobre el monto recibido.
- Luego resta ese descuento para obtener el monto final y muestra un mensaje informando el resultado.
- Devuelve el valor del monto final después de aplicar el descuento.

```
package sistecomm;

/**
 *
 * @author Sandra Martinez
 */
public class PayPal implements PagoConDescuento {
    private String correo;

    public PayPal(String correo) {
        this.correo = correo;
    }

     Override
    public void procesarPago(double monto) {
        System.out.println("Proceso de pago con PayPal de la cuenta " + correo + " por $" + monto);
    }

     Override
    public double aplicarDescuento(double monto) {
        double descuento = monto * 0.05;
        double montoFinal = monto - descuento;
        System.out.println("Se aplicó 5% de descuento. Monto final: $" + montoFinal);
        return montoFinal;
    }
```

Explicación del código:

public class PayPal implements PagoConDescuento

- Se crea una clase pública llamada PayPal.
- Usa implements para indicar que cumple con la interfaz PagoConDescuento.
- Esto obliga a definir los métodos **procesarPago()** y **aplicarDescuento()**.

private String correoUsuario;

- Se declara un atributo privado llamado correoUsuario.

- Guarda el mail asociado a la cuenta PayPal.

public PayPal(String correoUsuario)

- Es el **constructor** de la clase.
- Permite crear un objeto PayPal indicando el correo del usuario.

this.correoUsuario = correoUsuario;

- Asigna el valor recibido al atributo interno del objeto.
- this sirve para diferenciar entre el parámetro del método y el atributo.

@Override

- Indica que el método que sigue está implementando uno de la interfaz PagoConDescuento.

public void procesarPago(double monto)

- Este método muestra un mensaje indicando que se está realizando un pago con PayPal.
- Informa el correo del usuario y el monto del pago.

public double aplicarDescuento(double monto)

- Aplica un **5% de descuento** sobre el monto recibido.
- Calcula el monto final restando ese descuento.
- Muestra un mensaje con el nuevo total.
- Devuelve el monto final con el descuento aplicado.

5. Crear una interfaz **Notificable** para notificar cambios de estado. La clase **Cliente** implementa dicha interfaz y **Pedido** debe notificarlo al cambiar de estado.

```
/**
 *
 * @author Sandra Martinez
 */
public interface Notificable {
    void notificarCambioEstado(String mensaje);

}
```

Explicación código:

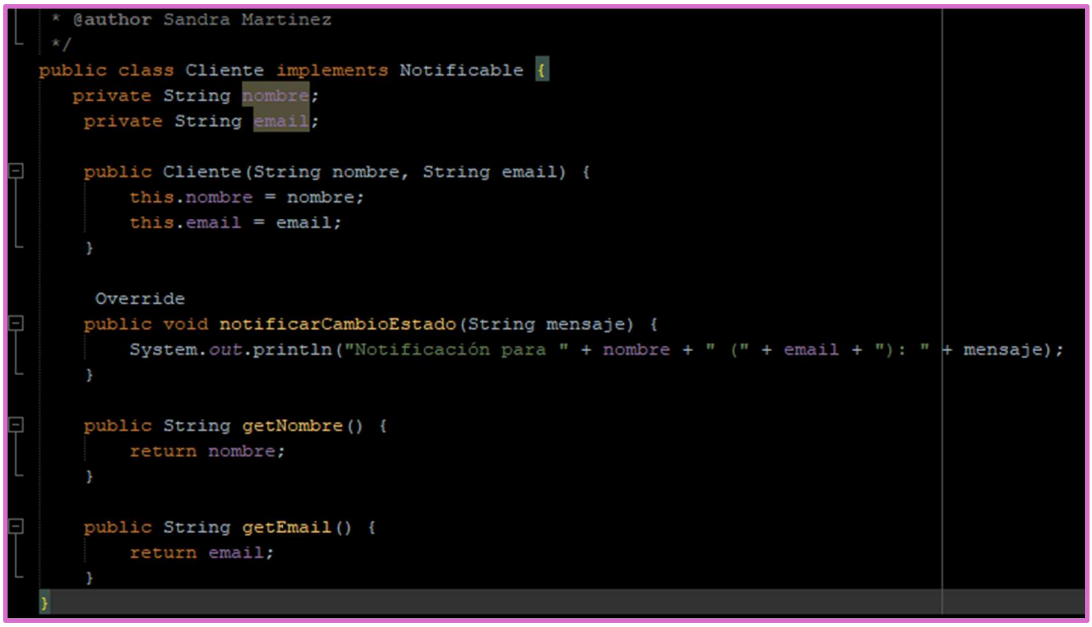
public interface Notificable

- **Se declara una interfaz pública llamada Notificable.**
- En este caso, el comportamiento está relacionado con enviar notificaciones cuando ocurre un **cambio de estado.**

void notificarCambioEstado(String mensaje);

- **Es el único método que la interfaz exige.**
- void significa que el método no devuelve ningún valor.
- El método recibe un **parámetro tipo String** llamado mensaje, que contendrá el texto de la notificación.

- Cualquier clase que implemente esta interfa estará **obligada a definir cómo se envía o muestra esa notificación**.



```
 * @author Sandra Martinez
 */
public class Cliente implements Notificable {
   private String nombre;
    private String email;

    public Cliente(String nombre, String email) {
        this.nombre = nombre;
        this.email = email;
    }

     Override
    public void notificarCambioEstado(String mensaje) {
        System.out.println("Notificación para " + nombre + " (" + email + "): " + mensaje);
    }

    public String getNombre() {
        return nombre;
    }

    public String getEmail() {
        return email;
    }
}
```

Explicación del código:

public class Cliente implements Notificable

- Se declara una clase pública llamada Cliente que implementa la interfaz Notificable, lo que significa que debe definir el método notificarCambioEstado.
  private String nombre;
  private String email;
- Se declaran dos atributos privados: nombre, que almacena el nombre del cliente, y email, que guarda su dirección de correo electrónic

public Cliente(String nombre, String email)

- Es el constructor de la clase. Permite crear un nuevo objeto Cliente recibiendo como parámetros el nombre y el correo electrónico.

this.nombre = nombre;
this.email = email;

- Asignan los valores recibidos a los atributos internos del objeto. La palabra this se usa para diferenciar el atributo del parámetro.

@Override

- Indica que el método que sigue proviene de la interfaz **Notificable**

public void notificarCambioEstado(String mensaje)

- Es el método obligatorio de la interfaz. Muestra un mensaje por consola informando al cliente sobre el cambio de estado.
  System.out.println("Notificación para " + nombre + " (" + email + "): " + mensaje);

- Imprime el mensaje con los datos del cliente y el contenido de la notificación. **public String getNombre() y public String getEmail()**
- Son métodos de acceso **(getters)** que permiten obtener el nombre y el email del cliente desde fuera de la clase, manteniendo los atributos encapsulados.

```
 *
 * @author Sandra Martinez
 */

public class Principal {

    public static void main(String[] args) {

        Producto p1 = new Producto("Camiseta", 15000);
        Producto p2 = new Producto("Zapatillas", 35000);
        Producto p3 = new Producto("Pantalón", 20000);

        Pedido pedido = new Pedido();
        pedido.agregarProducto(p1);
        pedido.agregarProducto(p2);
        pedido.agregarProducto(p3);

        double total = pedido.calcularTotal();
        System.out.println("Total del pedido: $" + total);

        TarjetaCredito tarjeta = new TarjetaCredito("Sandra Martinez", "1234-5678-9999");
        double totalConDescuento = tarjeta.aplicarDescuento(total);
        tarjeta.procesarPago(totalConDescuento);

        System.out.println("Tu pedido fue procesado y está en camino.");
    }
}
```

public class Principal

- Se declara una clase pública llamada **Principal**.
- Es la clase que contiene el método **main**, punto de entrada del programa.

public static void main(String[] args)

- Es el método principal que se ejecuta cuando corre el programa.
- Dentro de él se crean los objetos y se realizan las operaciones necesarias.

Producto p1 = new Producto("Camiseta", 15000);

Producto p2 = new Producto("Zapatillas", 35000);

Producto p3 = new Producto("Pantalón", 20000);

- Se crean tres objetos de tipo **Producto**, cada uno con su nombre y precio.
- Cada producto representa un artículo que luego formará parte del pedido.

Pedido pedido = new Pedido();

- Se crea un objeto de tipo **Pedido**.

- Este objeto va a contener la lista de productos que el cliente selecciona.

pedido.agregarProducto(p1);
pedido.agregarProducto(p2);
pedido.agregarProducto(p3);

- Se agregan los tres productos creados al pedido.
- El método **agregarProducto()** incorpora cada producto dentro de la lista interna del pedido.

double total = pedido.calcularTotal();

- Se calcula el total del pedido sumando los precios de todos los productos.
- El resultado se guarda en la variable **total**.

System.out.println("Total del pedido: $" + total);

- Muestra por pantalla el total a pagar por el pedido.

TarjetaCredito tarjeta = new TarjetaCredito("Sandra Martinez", "1234-5678-9999");

- Se crea un objeto **TarjetaCredito**, indicando el nombre del titular y el número de la tarjeta.

double totalConDescuento = tarjeta.aplicarDescuento(total);

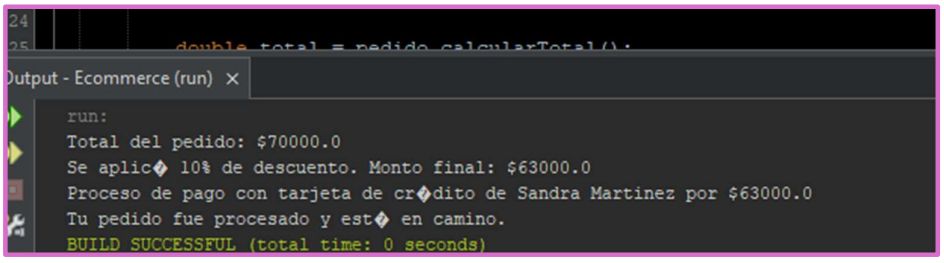
- Se aplica un descuento al total del pedido.
- El método **aplicarDescuento()** devuelve el monto final con descuento incluido.

tarjeta.procesarPago(totalConDescuento);

- Simula el procesamiento del pago con la tarjeta.
- Muestra un mensaje con el monto final pagado.

System.out.println("Tu pedido fue procesado y está en camino.");

- Finalmente, se imprime un mensaje informando que el pedido ya fue procesado.



Parte 2: Ejercicios sobre Excepciones

1. **División segura**
   - Solicitar dos números y dividirlos. Manejar **ArithmeticException** si el divisor es cero.

```java
import java.util.Scanner;

/**
 *
 * @author Sandra Martinez
 */
public class Division {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Ingrese el primer numero: ");
        int dividendo = scanner.nextInt();

        System.out.print("Ingrese el segundo numero: ");
        int divisor = scanner.nextInt();

        try {
            int resultado = dividendo / divisor;
            System.out.println("El resultado es: " + resultado);
        } catch (ArithmeticException e) {
            System.out.println("Error: no se puede dividir por cero.");
        } finally {
            System.out.println("Fin.");
            scanner.close();
        }
    }
}
```

Explicación del código:

public static void main(String[] args)

- Método principal que se ejecuta al iniciar el programa

Scanner scanner = new Scanner(System.in);

- Crea un objeto Scanner para leer números del usuario.

System.out.print("Ingrese el dividendo: ");

- Muestra mensaje para que el usuario ingrese el dividendo.

int dividendo = scanner.nextInt();

- Lee el número ingresado y lo guarda en la variable dividendo.

System.out.print("Ingrese el divisor: ");

- Muestra mensaje para que el usuario ingrese el divisor.

int divisor = scanner.nextInt();

- Lee el número ingresado y lo guarda en la variable divisor.

try { int resultado = dividendo / divisor; ... }

- Intenta ejecutar la división.
- Si el divisor es cero, genera una **ArithmeticException. (protege la división y muestra un mensaje si el divisor es cero.)**

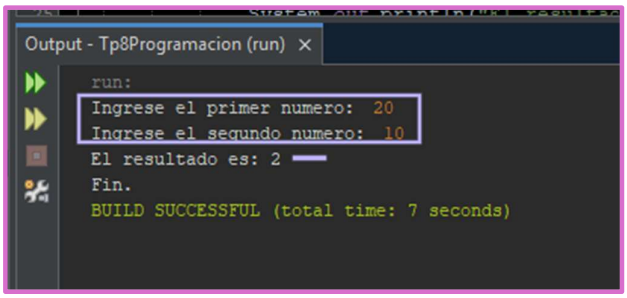
catch (ArithmeticException e) { ... }

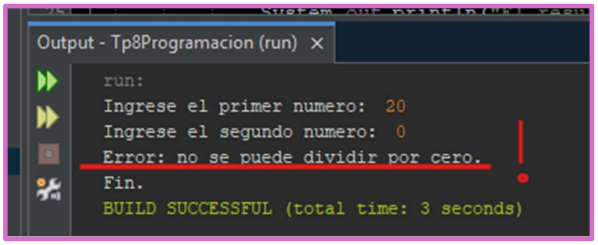
- Captura la excepción y evita que el programa se caiga.
- Muestra un mensaje de error indicando que no se puede dividir por cero.

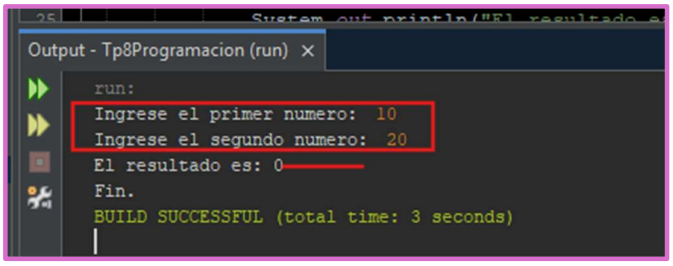
finally { ... scanner.close(); }

- Se ejecuta siempre, haya ocurrido o no la excepción.
- Cierra el Scanner para liberar recursos.

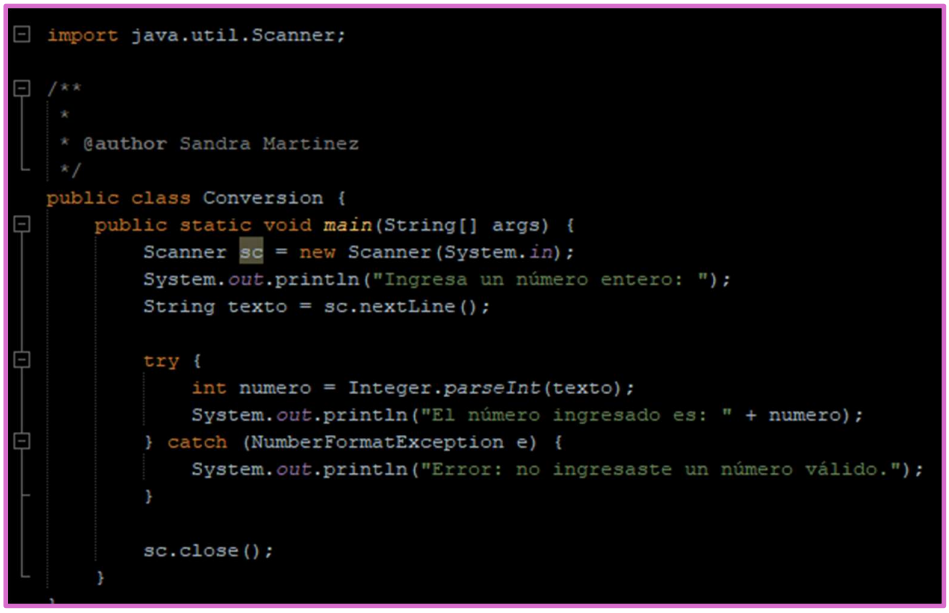
Pruebas:







2. **Conversión de cadena a número**
   - Leer texto del usuario e intentar convertirlo a int. Manejar **NumberFormatException** si no es válido.



```java
import java.util.Scanner;

/**
 *
 * @author Sandra Martinez
 */
public class Conversion {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Ingresa un número entero: ");
        String texto = sc.nextLine();

        try {
            int numero = Integer.parseInt(texto);
            System.out.println("El número ingresado es: " + numero);
        } catch (NumberFormatException e) {
            System.out.println("Error: no ingresaste un número válido.");
        }

        sc.close();
    }
}
```

public static void main(String[] args)

- Es el método principal donde empieza a ejecutarse el programa.

Scanner sc = new Scanner(System.in);

- Crea un objeto Scanner llamado sc para capturar la entrada del usuario desde el teclado.

System.out.println("Ingresa un número entero: ");

- Muestra en la consola un mensaje para que el usuario escriba un número.

String texto = sc.nextLine();

- Guarda en la variable texto lo que el usuario escribe como cadena de texto.

try

- Inicia la sección donde se intenta hacer algo que puede dar error.

int numero = Integer.parseInt(texto);

- Intenta convertir la cadena de texto a un número entero.

System.out.println("El número ingresado es: " + numero);

- Muestra en la consola el número convertido.

catch (NumberFormatException e)

- Captura el error si el texto no se puede convertir a número.

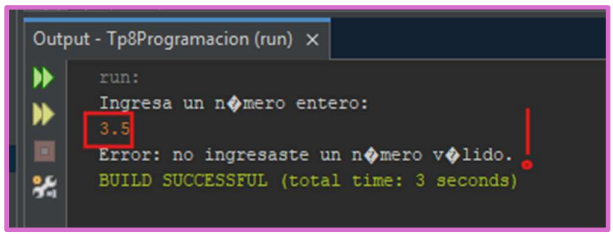
System.out.println("Error: no ingresaste un número válido.");

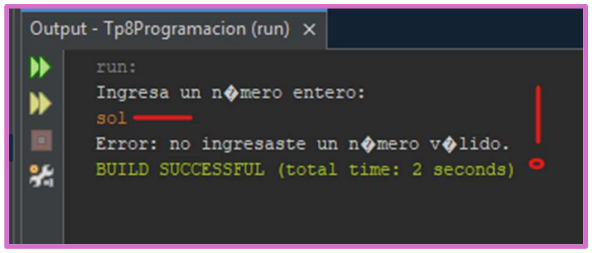
- Muestra un mensaje de error indicando que el usuario no ingresó un número válido.

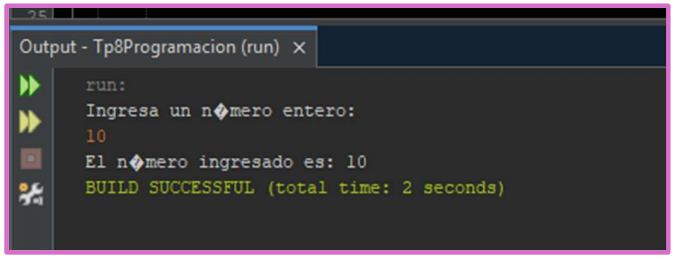
sc.close();

- Cierra el Scanner para liberar el recurso de entrada.

Pruebas:

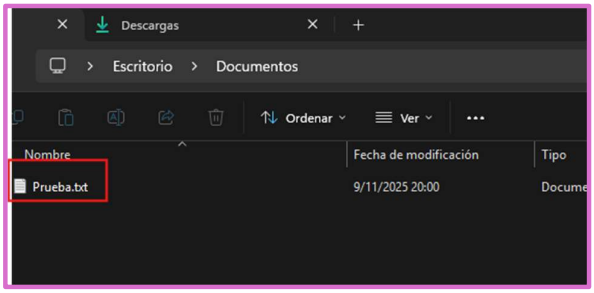


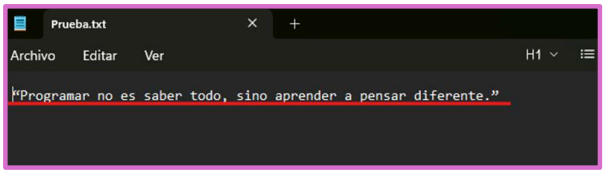




3. **Lectura de archivo**
   - Leer un archivo de texto y mostrarlo. Manejar **FileNotFoundException** si el archivo no existe.

Primero, para est punto cree un archivo txt.





Codigo:

```
package excepciones;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
/**
 *
 * @author Sandra Martinez
 */
public class EscribirArchivo {
    public static void main(String[] args) {
        try {
            File archivo = new File("C:\\Users\\sandra\\Desktop\\Documentos\\Prueba.txt");
            Scanner lector = new Scanner(archivo);

            while (lector.hasNextLine()) {
                String linea = lector.nextLine();
                System.out.println(linea);
            }

            lector.close();
        } catch (FileNotFoundException e) {
            System.out.println("Error: el archivo no fue encontrado.");
        }
    }
}
```

Explicación del código:

public static void main(String[] args)
→ Indica el punto de inicio del programa. Todo lo que está dentro del método main se ejecuta cuando el programa comienza.
try {
→ Inicia un bloque donde se coloca el código que podría generar una excepción.
File archivo = new File("C:\Users\sandra\Desktop\Documentos\Prueba.txt");

- Crea un objeto File llamado archivo, indicando la ruta exacta del archivo que se quiere leer.

Scanner lector = new Scanner(archivo);

- Crea un objeto Scanner llamado lector, que abrirá el archivo para leer su contenido.

while (lector.hasNextLine()) {

- Inicia un bucle que se repite mientras haya una nueva línea de texto **disponible en el archivo.**

String linea = lector.nextLine();

- Lee una línea completa del archivo y la guarda en la variable linea.

System.out.println(linea);

- Muestra por pantalla la línea leída, para que el usuario vea el contenido del archivo.

lector.close();

- Cierra el objeto Scanner para liberar el recurso del archivo después de leerlo. Es una buena práctica.

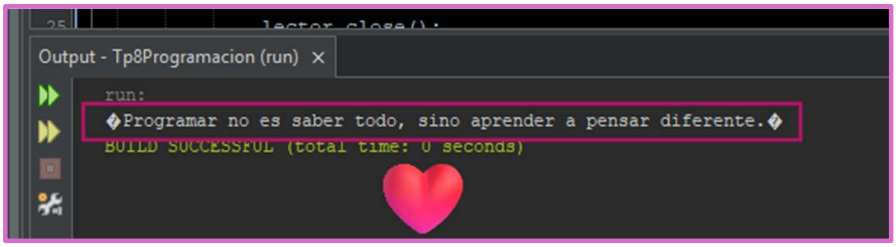
} catch (FileNotFoundException e) {

- Si el archivo no existe o la ruta es incorrecta, el programa no se detiene: entra en este bloque.

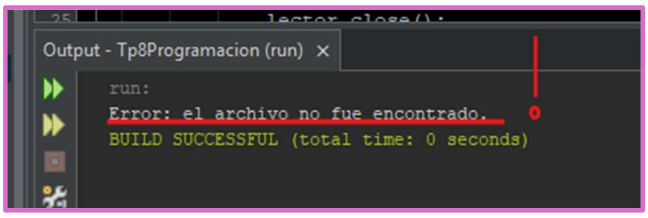
System.out.println("Error: el archivo no fue encontrado.");

- Muestra un mensaje de error claro para el usuario en caso de que el archivo no se haya podido abrir.

Prueba:



En el código edité el nombre del archivo y lke agregué una “s”. El archivo no existe.



4. **Excepción personalizada**
    - Crear **EdadInvalidaException**. Lanzarla si la edad es menor a 0 o mayor a 120. Capturarla y mostrar mensaje.

```java
/**
 *
 * @author Sandra Martinez
 */
public class EdadInvalidaException extends Exception {
    public EdadInvalidaException(String mensaje) {
        super(mensaje);
    }
}
```

La clase **EdadInvalidaException** es una excepción personalizada que hereda de la clase **Exception.** Su propósito es representar un error específico cuando la edad ingresada no cumple con las condiciones establecidas.

El **constructor** recibe un mensaje que se envía a la clase base mediante **super(mensaje),** lo que permite mostrar un texto descriptivo al capturar la excepción.

```java
package excepciones;

import java.util.Scanner;

/**
 *
 * @author Sandra Martinez
 */
public class VerificarEdad {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        try {
            System.out.print("Ingrese su edad: ");
            int edad = scanner.nextInt();

            if (edad < 0 || edad > 120) {
                throw new EdadInvalidaException("Error: la edad debe estar entre 0 y 120 años.");
            }

            System.out.println("Edad válida: " + edad);

        } catch (EdadInvalidaException e) {
            System.out.println(e.getMessage());
        } finally {
            System.out.println("Fin.");
            scanner.close();
        }
    }
```

Explicación del código:

Scanner scanner = new Scanner(System.in);

- Crea un objeto Scanner que permitirá leer los datos ingresados por teclado.

try

- Inicia un bloque que controlará posibles errores durante la ejecución del código.

System.out.print("Ingrese su edad: ");

- Muestra un mensaje en pantalla para solicitar al usuario que ingrese su edad.

int edad = scanner.nextInt();

- Lee el número entero ingresado por el usuario y lo guarda en la variable edad.

if (edad < 0 || edad > 120)

- Verifica si la edad está fuera del rango válido (menor que 0 o mayor que 120).

throw new EdadInvalidaException("Error: la edad debe estar entre 0 y 120 años.");

- Genera una excepción personalizada con un mensaje de error si la edad no cumple con el rango.

System.out.println("Edad válida: " + edad);

- Muestra por pantalla la edad ingresada si es correcta.

catch (EdadInvalidaException e)

- Captura la excepción lanzada en caso de que la edad sea inválida.

System.out.println(e.getMessage());

- Muestra el mensaje de error definido en la excepción.

finally

- Bloque que siempre se ejecuta, ocurra o no una excepción.

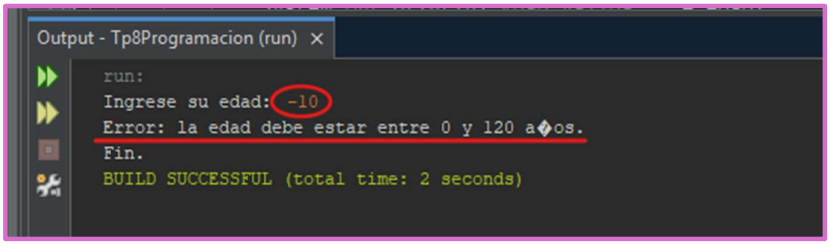
System.out.println("Fin.");

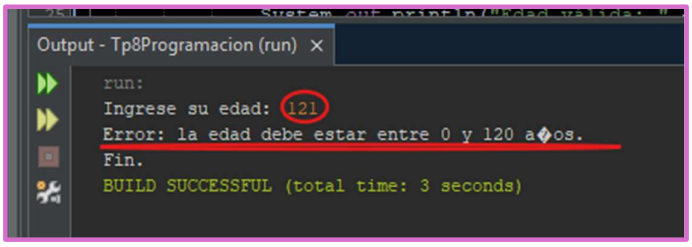
- Imprime un mensaje final indicando que el programa terminó.

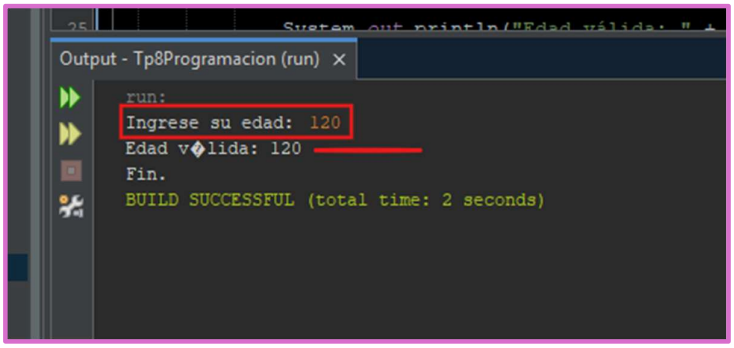
scanner.close();

- Cierra el objeto Scanner para liberar los recursos utilizados.

Pruebas:







5. **Uso de try-with-resources**
   - Leer un archivo con **BufferedReader** usando **try-with-resources**. Manejar **IOException** correctamente.

✓ **BufferedReader**}: sirve para leer texto línea por línea.
✓ **FileReader** : abre el archivo.
✓ **IOException** :maneja errores si el archivo no existe o no se puede leer.

import java.io.BufferedReader;
❖ Importa la clase BufferedReader, que permite leer texto línea por línea desde un archivo.

import java.io.FileReader;
❖ Importa la clase FileReader, que abre un archivo de texto para su lectura.

import java.io.IOException;
❖ Importa la clase IOException, usada para manejar errores de entrada o salida.

public class LeerArchivo

❖ Declara la clase pública LeerArchivo que contiene el método principal.

public static void main(String[] args)
❖ Método principal que se ejecuta al iniciar el programa.

try (BufferedReader lector = new BufferedReader(new FileReader("C:\\Users\\sandra\\Desktop\\Documentos\\Prueba.txt"))
❖ Inicia un bloque try-with-resources que crea un BufferedReader para leer el archivo “Prueba.txt”. El recurso se cierra automáticamente al finalizar el bloque.

String linea;
❖ Declara la variable linea, donde se guardará cada línea del archivo.

while ((linea = lector.readLine()) != null
❖ Inicia un bucle que se repite mientras haya líneas en el archivo. El método readLine() lee una línea y devuelve null al llegar al final.

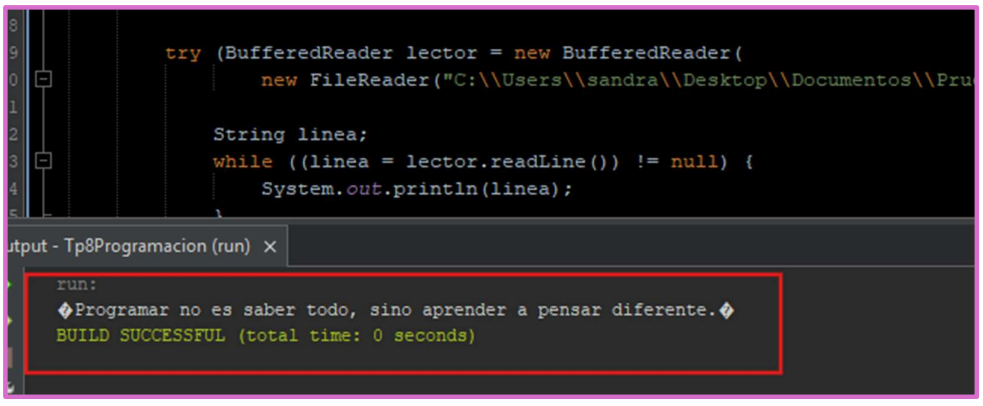
System.out.println(linea);
❖ Muestra por pantalla la línea leída del archivo.

catch (IOException e
❖ Captura los errores al abrir o leer el archivo (por ejemplo, si no existe).

System.out.println("Error al leer el archivo: " + e.getMessage());
❖ Muestra un mensaje de error con el detalle del problema.

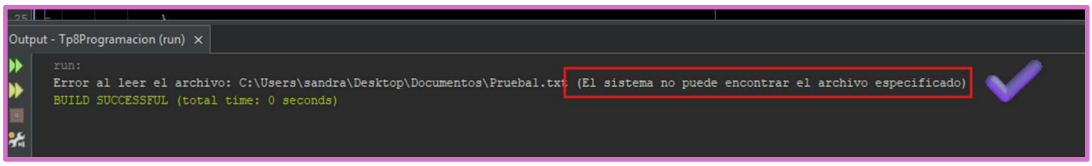
Prueba:

Utilizando el mismo archivo creado en el ejercicio anterior, estas fueronlas pruebas



Aquí cambiando el código:

```
try (BufferedReader lector = new BufferedReader(
        new FileReader("C:\\Users\\sandra\\Desktop\\Documentos\\Pruebal.txt"))) {
```



CONCLUSIONES ESPERADAS

- Comprender la utilidad de las interfaces para lograr diseños desacoplados y reutilizables.
- Aplicar herencia múltiple a través de interfaces para combinar comportamientos.
- Utilizar correctamente estructuras de control de excepciones para evitar caídas del programa.
- Crear excepciones personalizadas para validar reglas de negocio.
- Aplicar buenas prácticas como **try-with-resources** y uso del bloque **finally** para manejar recursos y errores.
- Reforzar el diseño robusto y mantenible mediante la integración de interfaces y manejo de errores en Java.

GitHub:
https://github.com/Sandra86Martinez/Tp8Programacion