PROGRAMACIÓN II

TP 8: Interfaces y Excepciones en Java

Alumno: Victor Barroeta

DNI: 95805903

Comisión: M2025 – 2

Matricula: 100393

Link de Github:
https://github.com/victormbar/UTN_Prog2_VictorBarroeta/tree/main/javaUTN_TPs/src/TPS_JAVA_TP8

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().**

```
public interface Pagable {
    public double calcularTotal();
}
```

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

```java
public class Producto implements Pagable{
    private String nombre;
    private double precio;

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

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public double getPrecio() {
        return precio;
    }

    public void setPrecio(double precio) {
        this.precio = precio;
    }

    @Override
    public double calcularTotal() {
        return this.precio;
    }

}
```

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

```java
public class Pedido implements Pagable { // Extendemos de pagable
    // Declaramos atributos
    ArrayList<Producto> productos;
    private String estado;
    private Cliente cliente;

    // Creamos el constructor inicializando el array
    public Pedido(String estado, Cliente cliente) {
        this.estado = estado;
        this.cliente = cliente;
        this.productos = new ArrayList();
    }

    // Metodo para agregar un producto al array
    public void agregarProducto(Producto producto) {
        productos.add(producto);
    }

    // Sobreescribimos el metodo calcular total
    @Override
    public double calcularTotal() {
        double total = 0;
        for (Producto p : productos) {
            total += p.getPrecio();
        }

        return total;
    }

    // Metodo para notificar el estado del pedido al cliente
    public void notificarEstado(String nuevoestado) {
        this.estado = nuevoestado;
        cliente.notificarCamboDeEstado(nuevoestado);
    }

}
```

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

```java
public interface Pago {
    public void procesarPago(double monto);
}
```

```java
package TPS_JAVA_TP8.E_Commerce;

public interface PagoConDescuento extends Pago{
    public double aplicarDescuento(double monto);
}
```

```java
public class PagoConTarjeta implements Pago{

    @Override
    public void procesarPago(double monto) {
        System.out.println("El total es: " + monto);
        System.out.println("El pago se realizo con éxito!");
    }

}
```

```java
public class PagoConPayPal implements PagoConDescuento{

    @Override
    public double aplicarDescuento(double monto) {
        return monto - (monto * 0.15);
    }

    @Override
    public void procesarPago(double monto) {
        double total = aplicarDescuento(monto);
        System.out.println("El total con descuento es: " + total);
        System.out.println("El pago se realizo correctamente");
    }

}
```

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

```java
public interface Notificable{
    public void notificarCamboDeEstado(String nuevoEstado);
}
```

```java
public class Cliente implements Notificable{

    private String nombre;

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

    @Override
    public void notificarCamboDeEstado(String nuevoEstado) {
        System.out.println(nombre + " tu pedido cambio de estado a " + nuevoEstado);
    }

}
```

```java
public class Main {

    public static void main(String[] args) {
        // Creamos dos clientes
        Cliente c1 = new Cliente("Cliente 1");
        Cliente c2 = new Cliente("Cliente 2");

        // Creamos un pedido para cada cliente
        Pedido pedido = new Pedido("Pendiente", c1);
        Pedido pedido2 = new Pedido("Pendiente", c2);

        // Creamos 3 productos
        Producto p1 = new Producto("Play", 10000000);
        Producto p2 = new Producto("Microondas", 500000);
        Producto p3 = new Producto("Computadora", 750000);

        // Agregamos 2 productos al pedido del cliente 1
        pedido.agregarProducto(p1);
        pedido.agregarProducto(p3);

        // Notificamos el estado en proceso del pedido del cliente 1
        pedido.notificarEstado("EN_PROCESO");

        // Calculamos el total del pedido del cliente 1
        double total = pedido.calcularTotal();

        // Creamos un nuevo pago con pay pal
        PagoConPayPal pago = new PagoConPayPal();

        // Procesamos el pago del cliente 1
        pago.procesarPago(total);

        // Notificamos el estado del pedido pagado
        pedido.notificarEstado("PAGADO");
```

```
    pedido.notificarEstado("EN_PROCESO");

    // Calculamos el total del pedido del cliente 1
    double total = pedido.calcularTotal();

    // Creamos un nuevo pago con pay pal
    PagoConPayPal pago = new PagoConPayPal();

    // Procesamos el pago del cliente 1
    pago.procesarPago(total);

    // Notificamos el estado del pedido pagado
    pedido.notificarEstado("PAGADO");

    // Agregamos un producto al pedido del cliente 2
    pedido2.agregarProducto(p2);
    // Notificamos su estado en proceso
    pedido2.notificarEstado("EN_PROCESO");

    // Calculamos el total del pedido del cliente 2
    double total2 = pedido2.calcularTotal();

    // Creamos un pago con tarjeta
    PagoConTarjeta pago2 = new PagoConTarjeta();

    // Procesamos el pago con el monto del pedido del cliente 2
    pago2.procesarPago(total2);

    // Notificamos al cliente 2 su estado de pedido pagado
    pedido2.notificarEstado("PAGADO");
}
```

```
Cliente 1 tu pedido cambio de estado a EN_PROCESO
El total con descuento es: 9137500.0
El pago se realizo correctamente
Cliente 1 tu pedido cambio de estado a PAGADO
Cliente 2 tu pedido cambio de estado a EN_PROCESO
El total es: 500000.0
El pago se realizo con xito!
Cliente 2 tu pedido cambio de estado a PAGADO
BUILD SUCCESSFUL (total time: 0 seconds)
```

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;

public class DivisionSegura {

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        double num1, num2;

        System.out.println("Ingrese dos numeros");
        System.out.println("Numero uno: ");
        num1 = scan.nextDouble();

        System.out.println("Numero dos:");
        num2 = scan.nextDouble();

        try {
            if (num2 == 0) {
                throw new ArithmeticException("No se puede dividir por 0");
            }
            double resultado = num1 / num2;
            System.out.println("El resultado de la división es: " + resultado);
        } catch (ArithmeticException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }

}
```

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;

public class ConversionNumeroACadena {

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.println("Ingrese un texto que pueda ser convertido a entero");
        String textoAConvertir = scan.nextLine();

        try {
            int textoConvertido = Integer.parseInt(textoAConvertir);
            System.out.println("El número convertido es: " + textoConvertido);
        } catch (NumberFormatException ex) {
            System.out.println("Error: El texto ingresado no es un número entero válido.");
        }
    }

}
```

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

```java
package TP3_JAVA_TP3.Excepciones;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.Scanner;

public class LecturaDeArchivo {

    public static void main(String[] args) throws IOException {
        Scanner scan = new Scanner(System.in);

        System.out.println("Ingrese el nombre del archivo txt: ");
        String nombre = scan.nextLine();

        try {
            File archivo = new File(nombre);
            BufferedReader br = new BufferedReader(new FileReader(archivo));
            System.out.println(br.readLine());
        } catch (FileNotFoundException e) {
            System.out.println("Error: El archivo no fue encontrado.");
        } catch(IOException ex){
            System.out.println("Error de E/S");
            System.out.println(ex.getMessage());
        }
    }

}
```

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

```java
import java.util.Scanner;

public class ExcepcionPersonalizada {

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

        System.out.println("Ingrese una edad");
        int edad = Integer.parseInt(scan.nextLine());
        if (edad <= 0 || edad >= 120) {
            throw new EdadInvalidaException("Ingreso una edad invalida");
        }

    }

}
```

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

```java
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class TryWithResources {

    public static void main(String[] args) {
        File archivo = new File("C:\\Users\\vcman\\OneDrive\\Documentos\\UTN Programacion\\2do Cuatrimestre\\Programación II\\TP
        try(BufferedReader br = new BufferedReader(new FileReader(archivo))){
            System.out.println(br.readLine());
        } catch(IOException ex) {
            System.out.println("Error de E/S: "+ ex.getMessage());
        }
    }

}
```

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.