

<https://github.com/Coman-Che/Programacion-II/tree/main/Unidad%208>

# PROGRAMACIÓN II

## Trabajo Práctico 8: Interfaces y Excepciones en Java

### OBJETIVO GENERAL

Desarrollar habilidades en el uso de Genéricos en Java para mejorar la seguridad, reutilización y escalabilidad del código. Comprender la implementación de clases, métodos e interfaces genéricas en estructuras de datos dinámicas. Aplicar comodines (?, extends, super) para gestionar diferentes tipos de datos en colecciones. Utilizar Comparable y Comparator para ordenar y buscar elementos de manera flexible. Integrar Genéricos en el diseño modular del software.

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 <b>implements</b> para que una clase cumpla con los métodos definidos en una interfaz
Excepciones	Manejo de errores en tiempo de ejecución mediante estructuras <b>try-catch</b>
Excepciones checked y unchecked	Diferencias y usos según la naturaleza del error
Excepciones personalizadas	Creación de nuevas clases que extienden <b>Exception</b>
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 Práctico

#### Parte 1: Interfaces en un sistema de E-commerce

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

```
package Parte1_Interfaces;  
  
public interface Pagable {  
  
    double calcularTotal();  
  
}
```

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

```
package Parte1_Interfaces;

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;
    }

    // Getters
    public String getNombre() {
        return nombre;
    }

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

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

```
package Parte1_Interfaces;

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

public class Pedido implements Pagable, Notificable {

    private List<Producto> productos;
    private String estado;
    private Cliente cliente;

    public Pedido(Cliente cliente) {
        this.cliente = cliente;
        this.productos = new ArrayList<>();
        this.estado = "Pendiente";
    }

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

```

@Override
public double calcularTotal() {
    return productos.stream().mapToDouble(Producto::calcularTotal).sum();
}

public void cambiarEstado(String nuevoEstado) {
    this.estado = nuevoEstado;
    notificar(); // Notifica al cliente cuando cambia el estado
}

@Override
public void notificar() {
    cliente.recibirNotificacion("Tu pedido ahora está: " + estado);
}
}

```

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

```

package Parte1_Interfaces;

public interface Pago {

    void procesarPago(double monto);

}

```

---

```

package Parte1_Interfaces;

public interface PagoConDescuento extends Pago {

    void aplicarDescuento(double porcentaje);

}

```

---

```

package Parte1_Interfaces;

public class TarjetaDeCredito implements PagoConDescuento {

    @Override
    public void procesarPago(double monto) {
        System.out.println("Pago con tarjeta de crédito procesado: $" + monto);
    }

    @Override

```

```
        public void aplicarDescuento(double porcentaje) {
            System.out.println("Descuento aplicado: " + porcentaje + "%");
        }
    }
}
```

---

```
package Parte1_Interfaces;
```

```
public class PayPal implements Pago {
```

```
    @Override
    public void procesarPago(double monto) {
        System.out.println("Pago con PayPal procesado: $" + monto);
    }
}
```

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

```
package Parte1_Interfaces;
```

```
public interface Notificable {
```

```
    void notificar();
}
```

---

```
package Parte1_Interfaces;
```

```
public class Main_Parte1 {
```

```
    public static void main(String[] args) {
```

```
        // Crear cliente
        Cliente cliente = new Cliente("Ana López");
```

```
        // Crear productos
        Producto laptop = new Producto("Laptop Gamer", 1200.0);
        Producto mouse = new Producto("Mouse Inalámbrico", 25.5);
```

```
        // Crear pedido
        Pedido pedido = new Pedido(cliente);
        pedido.agregarProducto(laptop);
        pedido.agregarProducto(mouse);
```

```
        System.out.println("Total del pedido: $" + String.format("%.2f",
            pedido.calcularTotal()));
    }
```

```

// Cambiar estado del pedido → notifica al cliente
pedido.cambiarEstado("Enviado");

// Procesar pago con tarjeta (con descuento)
TarjetaDeCredito tarjeta = new TarjetaDeCredito();
tarjeta.aplicarDescuento(10.0);
tarjeta.procesarPago(pedido.calcularTotal());

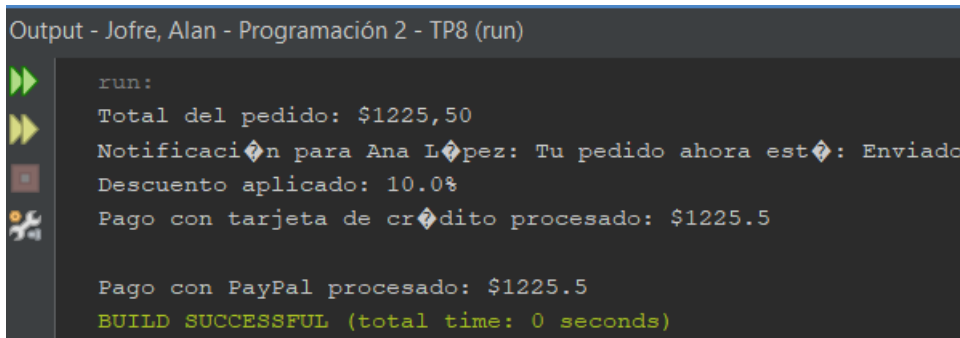
System.out.println();

// Procesar pago con PayPal (sin descuento)
PayPal paypal = new PayPal();
paypal.procesarPago(pedido.calcularTotal());

}

}

```



```

Output - Jofre, Alan - Programación 2 - TP8 (run)

run:
Total del pedido: $1225,50
Notificación para Ana Lopez: Tu pedido ahora está: Enviado
Descuento aplicado: 10.0%
Pago con tarjeta de crédito procesado: $1225.5

Pago con PayPal procesado: $1225.5
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.

```

package Parte2_Excepciones;

import java.util.Scanner;

public class DivisionSegura {

    public static void ejecutar() {
        Scanner sc = new Scanner(System.in);
        System.out.print("Ingresa el dividendo: ");
        int dividendo = sc.nextInt();
        System.out.print("Ingresa el divisor: ");
        int divisor = sc.nextInt();

        try {
            int resultado = dividendo / divisor;
            System.out.println("Resultado: " + dividendo + " / " + divisor + " = " + resultado);
        } catch (ArithmeticException e) {
            System.err.println("Error: No se puede dividir entre cero.");
        }
    }
}

```

```
}  
  
}
```

## 2. Conversión de cadena a número

- Leer texto del usuario e intentar convertirlo a `int`. Manejar `NumberFormatException` si no es válido.

```
package Parte2_Excepciones;  
  
import java.util.Scanner;  
  
public class ConversionCadenaANumero {  
  
    public static void ejecutar() {  
        Scanner sc = new Scanner(System.in);  
        System.out.print("Ingresa un número entero: ");  
        String entrada = sc.nextLine();  
  
        try {  
            int numero = Integer.parseInt(entrada);  
            System.out.println("Número válido: " + numero);  
        } catch (NumberFormatException e) {  
            System.err.println("Error: " + entrada + " no es un número entero válido.");  
        }  
    }  
}
```

## 3. Lectura de archivo

- Leer un archivo de texto y mostrarlo. Manejar `FileNotFoundException` si el archivo no existe.

```
package Parte2_Excepciones;  
  
import java.io.BufferedReader;  
import java.io.FileNotFoundException;  
import java.io.FileReader;  
import java.io.IOException;  
  
public class LecturaArchivo {  
  
    public static void ejecutar() {  
        try {  
            FileReader fr = new FileReader("datos.txt");  
            BufferedReader br = new BufferedReader(fr);  
            String linea;  
            System.out.println("Contenido del archivo:");  
            while ((linea = br.readLine()) != null) {  
                System.out.println(linea);  
            }  
            br.close();  
        }  
    }  
}
```



```
    } catch (FileNotFoundException e) {  
        System.err.println("Error: El archivo 'datos.txt' no fue encontrado.");  
    } catch (IOException e) {  
        System.err.println("Error al leer el archivo: " + e.getMessage());  
    }  
}  
}
```

#### 4. Excepción personalizada

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

```
package Parte2_Excepciones;
```

```
public class EdadInvalidaException extends Exception {  
  
    public EdadInvalidaException(String mensaje) {  
        super(mensaje);  
    }  
}
```

```
package Parte2_Excepciones;
```

```
public class ValidadorEdad {  
  
    public static void validarEdad(int edad) throws EdadInvalidaException {  
        if (edad < 0 || edad > 120) {  
            throw new EdadInvalidaException("La edad debe estar entre 0 y 120 años.");  
        }  
        System.out.println("Edad válida: " + edad + " años.");  
    }  
  
    public static void ejecutar() {  
        java.util.Scanner sc = new java.util.Scanner(System.in);  
        System.out.print("Ingresa tu edad: ");  
        int edad = sc.nextInt();  
  
        try {  
            validarEdad(edad);  
        } catch (EdadInvalidaException e) {  
            System.err.println("Error de validación: " + e.getMessage());  
        }  
    }  
}
```

#### 5. Uso de try-with-resources

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

```

package Parte2_Excepciones;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public class LecturaConTryWithResources {

    public static void ejecutar() {
        // try-with-resources: BufferedReader se cierra automáticamente
        try (BufferedReader br = new BufferedReader(new FileReader("datos.txt"))) {
            String linea;
            System.out.println("Leyendo con try-with-resources:");
            while ((linea = br.readLine()) != null) {
                System.out.println(linea);
            }
        } catch (FileNotFoundException e) {
            System.err.println("Archivo no encontrado: " + e.getMessage());
        } catch (IOException e) {
            System.err.println("Error de E/S: " + e.getMessage());
        }
    }
}

```

```

Output - Jofre, Alan - Programación 2 - TP8 (run)

run:
=== Ejercicio 1: División Segura ===
Ingresa el dividendo: 8
Ingresa el divisor: 4
Resultado: 8 / 4 = 2

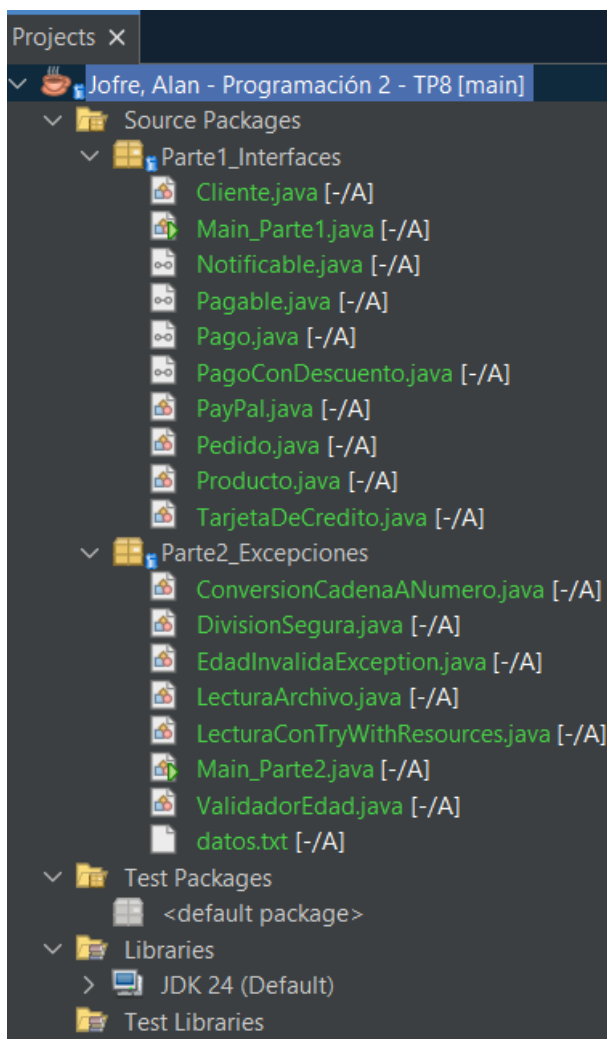
=== Ejercicio 2: Conversión de Cadena ===
Ingresa un número entero: 4
Número válido: 4

=== Ejercicio 3: Lectura de Archivo (método tradicional) ===
Error: El archivo 'datos.txt' no fue encontrado.

=== Ejercicio 4: Excepción Personalizada ===
Ingresa tu edad: 18
Edad válida: 18 años.

=== Ejercicio 5: Lectura con try-with-resources ===
Archivo no encontrado: datos.txt (El sistema no puede encontrar el archivo especificado)
BUILD SUCCESSFUL (total time: 9 seconds)

```



## 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.