# 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.

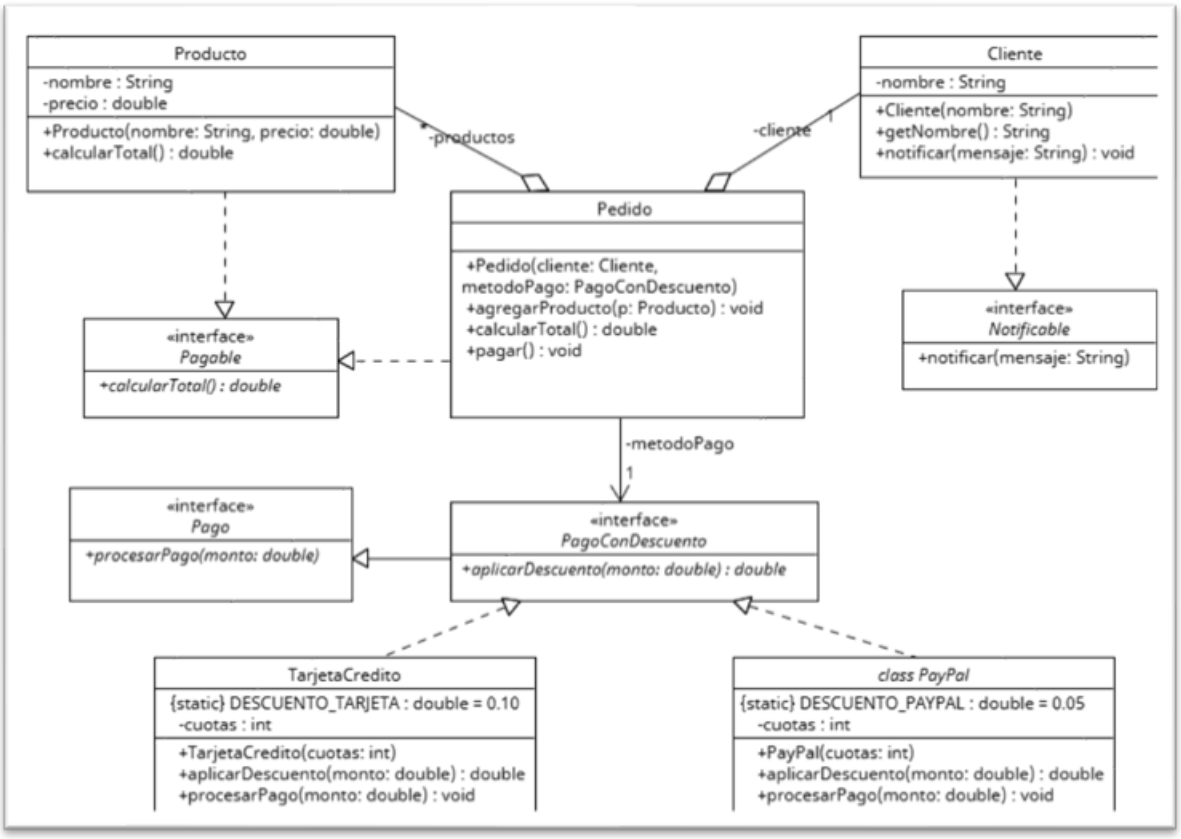
**Enlace Git Hub:**

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

```
 * @author marin
 */
public interface Pagable {
    public abstract double calcularTotal();

}
```

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.

## 2) Clase Producto

```java
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;
    }

    @Override
    public String toString() {
        return nombre + " - $" + precio;
    }
}
```

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.

## 3) Interface Notificable

```java
public interface Notificable {
    public abstract void notificar(String mensaje);
}
```

Define el comportamiento que deben cumplir los objetos capaces de recibir notificaciones.

Contiene el método notificar(String mensaje), utilizado para enviar avisos al cliente.

## 4) Clase Cliente

```java
public class Cliente implements Notificable {
    private String nombre;

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

    public String getNombre() {
        return nombre;
    }

    @Override
    public void notificar(String mensaje) {
        System.out.println(nombre + " recibió notificación: " + mensaje);
    }
}
```

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.

### 5) Interface Pago

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

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.

### 6) Interface PagoConDescuento

```java
// Extiende Pago y agrega comportamiento adicional.
public interface PagoConDescuento extends Pago {
    double aplicarDescuento(double monto);
}
```

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.

### 7) Clase TarjetaCredito

```java
public class TarjetaCredito implements PagoConDescuento {
    private static final double DESCUENTO_TARJETA = 0.10; // 10%
    private int cuotas;

    public TarjetaCredito(int cuotas) {
        this.cuotas = cuotas;
    }

    @Override
    public double aplicarDescuento(double monto) {
        if (cuotas == 1) {
            System.out.println(x: "Pago en 1 cuota: se aplica un descuento del 10%.");
            return monto * (1 - DESCUENTO_TARJETA);
        } else {
            System.out.println("Pago en " + cuotas + " cuotas: sin descuento.");
            return monto;
        }
    }

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

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.

**8) Clase PayPal**

```java
public class PayPal implements PagoConDescuento {
    private static final double DESCUENTO_PAYPAL = 0.05; // 5%
    private int cuotas;

    public PayPal(int cuotas) {
        this.cuotas = cuotas;
    }

    @Override
    public double aplicarDescuento(double monto) {
        if (cuotas == 1) {
            System.out.println(x: "Pago en 1 cuota: se aplica un descuento del 5%.");
            return monto * (1 - DESCUENTO_PAYPAL);
        } else {
            System.out.println("Pago en " + cuotas + " cuotas: sin descuento.");
            return monto;
        }
    }

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

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.

**9) Clase Pedido**

```java
public class Pedido implements Pagable {
    private List<Producto> productos = new ArrayList<>();
    private Cliente cliente;
    private PagoConDescuento metodoPago; // Usa la interfaz, no una clase concreta


    public Pedido(Cliente cliente, PagoConDescuento metodoPago) {
        this.cliente = cliente;
        this.metodoPago = metodoPago;
    }

    public void agregarProducto(Producto p) {
        productos.add(e: p);
        cliente.notificar("Producto agregado: " + p);
    }

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

    public void pagar() {
        double total = calcularTotal();
        double conDescuento = metodoPago.aplicarDescuento(monto: total);
        metodoPago.procesarPago(monto: conDescuento);
    }
```

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.

**10) Clase MainEcommerce**

```java
public static void main(String[] args) {
    // --- CLIENTE PRINCIPAL ---
    Cliente cliente = new Cliente(nombre: "Marina");

    // --- PEDIDO 1: Tarjeta de crédito en 3 cuotas (sin descuento) ---
    System.out.println(x: "========== PEDIDO 1 ==========");
    PagoConDescuento pagoTarjeta = new TarjetaCredito(cuotas: 3);
    Pedido pedido1 = new Pedido(cliente, metodoPago: pagoTarjeta);

    pedido1.agregarProducto(new Producto(nombre: "Libro", precio: 5000));
    pedido1.agregarProducto(new Producto(nombre: "Mouse", precio: 3500));

    pedido1.pagar(); // Se espera sin descuento

    // --- PEDIDO 2: PayPal en 1 pago (con descuento) ---
    System.out.println(x: "\n========== PEDIDO 2 ==========");
    PagoConDescuento pagoPayPal = new PayPal(cuotas: 1);
    Pedido pedido2 = new Pedido(cliente, metodoPago: pagoPayPal);

    pedido2.agregarProducto(new Producto(nombre: "Auriculares", precio: 6000));
    pedido2.agregarProducto(new Producto(nombre: "Teclado", precio: 4000));

    pedido2.pagar(); // Se espera con descuento
}
```

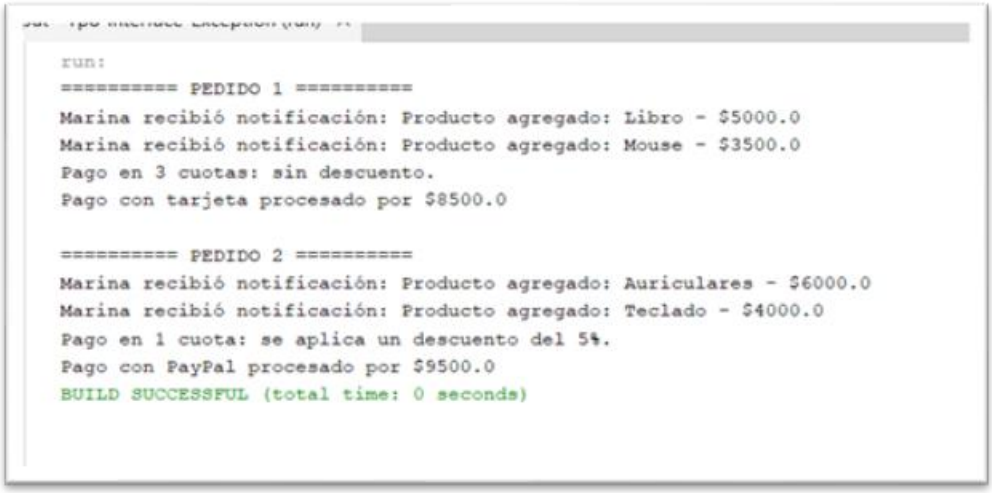
Clase principal del sistema.

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

```java
public static void divisionSegura() {
    Scanner sc = new Scanner(source: System.in);
    try {
        System.out.print(s: "Ingrese numerador: ");
        int numerador = Integer.parseInt(s: sc.nextLine().trim()); // Conversión de

        System.out.print(s: "Ingrese denominador: ");
        int denominador = Integer.parseInt(s: sc.nextLine().trim());

        // Esta línea puede lanzar ArithmeticException si el denominador es 0
        int resultado = numerador / denominador;

        System.out.println("Resultado: " + numerador + " / " + denominador
                + " = " + resultado);

    } catch (ArithmeticException e) {
        // Se ejecuta si ocurre una división por cero
        System.out.println(x: "⚠ Error: No se puede dividir por cero.");
    } catch (NumberFormatException e) {
        // Se ejecuta si el usuario escribe texto en lugar de números
        System.out.println(x: "⚠ Error: Debe ingresar números enteros válidos.");
    } finally {
        // Bloque que siempre se ejecuta, ocurra o no una excepción
        System.out.println(x: "Ejecución finalizada.\n");
    }
```

```
Ingrese numerador: 12
Ingrese denominador: 6
Resultado: 12 / 6 = 2
Ejecución finalizada.
```

```
Ingrese numerador: 12
Ingrese denominador: 0
?? Error: No se puede dividir por cero.
Ejecución finalizada.
```

- 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.

```java
public static void conversionCadenaNumero() {
    Scanner sc = new Scanner(source: System.in);

    try {
        System.out.print(s: "Ingrese un número entero: ");
        String texto = sc.nextLine().trim(); // Lee el texto ingresado por el usuario

        // Intenta convertir el texto a un número entero
        int numero = Integer.parseInt(s: texto);

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

    } catch (NumberFormatException e) {
        // Se ejecuta si el texto no es numérico
        System.out.println(x: "⚠ Error: El valor ingresado no es un número válido.");
    }

    System.out.println(x: "Programa finalizado.\n");
}
```

```
Ingrese un número entero: 1452
? El número ingresado es: 1452
Programa finalizado.
```

```
Seleccione una opción: 2
Ingrese un número entero: hola
?? Error: El valor ingresado no es un número válido.
Programa finalizado.
```

### 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.

```java
// ================================================================
public static void lecturaArchivo() {
    Scanner sc = new Scanner(source: System.in);
    System.out.print(s: "Ingrese el nombre del archivo: ");
    String nombreArchivo = sc.nextLine().trim();

    try {
        File archivo = new File(pathname: nombreArchivo);        // Crea un objet
        Scanner lector = new Scanner(source: archivo);          // Intenta abrir e

        System.out.println(x: "\n Contenido del archivo:");
        while (lector.hasNextLine()) {
            System.out.println(x: lector.nextLine());     // Muestra línea por l
        }

        lector.close();                                  // Cierra el archivo

    } catch (FileNotFoundException e) {
        // Se ejecuta si el archivo no existe o no se puede abrir
        System.out.println("⚠ Error: El archivo '" + nombreArchivo
                + "' no existe o no se puede abrir.");
    }

    System.out.println(x: "Lectura finalizada.\n");
}
```

```
? Contenido del archivo:
Este es un archivo de prueba para el ejercicio 3
Lectura finalizada.
```

```
Seleccione una opción: 3
Ingrese el nombre del archivo: archivo
?? Error: El archivo 'archivo' no existe o no se puede abrir.
Lectura finalizada.
```

### 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.

```
 * Se considera invalida si es menor a 0 o mayor a 120.
 */
public class EdadInvalidaException extends Exception {

    // Constructor que recibe un mensaje descriptivo
    public EdadInvalidaException(String mensaje) {
        super(message: mensaje);
    }
}
```

```
    // ============================================================
     public static void excepcionPersonalizada() {
    Scanner sc = new Scanner(source: System.in);

    System.out.println(x: "\n=== EJERCICIO 4: Excepción personalizada ===");

    try {
        // 1️⃣ Se solicita la edad al usuario
        System.out.print(s: "Ingrese su edad: ");
        int edad = Integer.parseInt(s: sc.nextLine().trim());

        // 2️⃣ Se valida la edad mediante un método que puede lanzar la excepción
        validarEdad(edad);

        // 3️⃣ Si no se lanzó excepción, la edad es válida
        System.out.println("✅ Edad válida: " + edad + " años.");

    }
    // Captura de la excepción personalizada (definida por nosotros)
    catch (EdadInvalidaException e) {
        System.out.println("⚠️ Error: " + e.getMessage());
    }
    // Captura de error por ingreso de texto en lugar de número
    catch (NumberFormatException e) {
        System.out.println(x: "⚠️ Error: debe ingresar un número entero.");
    }

    System.out.println(x: "Lectura finalizada.\n");
}
```

```java
/**
 * Método que valida una edad.
 * Si no cumple las condiciones, lanza una EdadInvalidaException.
 *
 * @param edad valor ingresado por el usuario
 * @throws EdadInvalidaException si la edad es negativa o superior a 120
 */
public static void validarEdad(int edad) throws EdadInvalidaException {
    if (edad < 0) {
        // Se lanza la excepción con un mensaje personalizado
        throw new EdadInvalidaException(mensaje: "La edad no puede ser negativa.");
    }
    if (edad > 120) {
        // Se lanza la excepción con otro mensaje personalizado
        throw new EdadInvalidaException(mensaje: "La edad no puede superar los 120 años.");
    }
```

```
Seleccione una opción: 4
Ingrese su edad: 25
? Edad válida: 25 años.
```

```
Seleccione una opcion: 4
Ingrese su edad: -5
?? Error: La edad no puede ser negativa.
```

```
Seleccione una opcion: 4
Ingrese su edad: 150
?? Error: La edad no puede superar los 120 años.
```

### 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.

```java
public static void lecturaTryWithResources() {
    Scanner sc = new Scanner(source: System.in);
    System.out.println(x: "\n=== EJERCICIO 5: Try-with-resources ===");

    // Se pide al usuario el nombre del archivo
    System.out.print(s: "Ingrese el nombre del archivo a leer: ");
    String nombreArchivo = sc.nextLine().trim();

    // El bloque try-with-resources garantiza el cierre automático del BufferedReader
    try (BufferedReader lector = new BufferedReader(new FileReader(fileName: nombreArchivo))) {

        System.out.println(x: "\n Contenido del archivo:");
        System.out.println(x: "-------------------------");

        // Se lee línea por línea hasta que readLine() devuelve null
        String linea;
        while ((linea = lector.readLine()) != null) {
            System.out.println(x: linea);
        }

        System.out.println(x: "\nLectura finalizada correctamente.");

    } catch (IOException e) {
        // Se captura cualquier error de entrada/salida:
        // - archivo inexistente
        // - permisos insuficientes
        // - error al leer
        System.out.println("⚠ Error al leer el archivo: " + e.getMessage());
```

```
Ingrese el nombre del archivo a leer: Ejemplo.txt.txt

? Contenido del archivo:
-------------------------
Este es un archivo de prueba para el ejercicio 3

Lectura finalizada correctamente.
```

```
Ingrese el nombre del archivo a leer: Ejemplo.txt
?? Error al leer el archivo: Ejemplo.txt (El sistema no puede encontrar el archivo especificado)
```

### 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.