




Trabajo Práctico 8: Interfaces y Excepciones en Java

Alumno: Calcatelli Renzo - rcalcatelli@gmail.com

Comisión: M2025-1

Materia: Programación II

Profesor: Ariel Enferrel

 [Enlace](#) al repositorio de GitHub

Trabajo Práctico 8: Interfaces y Excepciones en Java

ÍNDICE

1. Objetivos del Trabajo Práctico
2. Marco Teórico
3. Desarrollo del Caso Práctico
 - Parte 1: Interfaces en Sistema E-commerce
 - Parte 2: Ejercicios sobre Excepciones
4. Conclusiones
5. Bibliografía y Referencias

1. OBJETIVOS DEL TRABAJO PRÁCTICO

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.

Objetivos Específicos

- 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.

2. MARCO TEÓRICO

2.1 Interfaces en Java

Las interfaces son contratos que definen el comportamiento que una clase debe implementar. Permiten:

Concepto	Descripción
Definición de interfaces	Especificación de métodos sin implementación que las clases deben cumplir
Herencia múltiple	Una clase puede implementar múltiples interfaces, combinando comportamientos sin herencia de estado
Implementación	Uso de la palabra clave implements para que una clase cumpla con los métodos definidos
Polimorfismo	Permite tratar objetos de diferentes clases de manera uniforme a través de una interfaz común

Ventajas de usar Interfaces:

- Desacoplamiento:** El código depende de abstracciones, no de implementaciones concretas.
- Flexibilidad:** Facilita el cambio de implementaciones sin afectar el código cliente.
- Testabilidad:** Permite crear mocks y stubs para pruebas unitarias.
- Diseño por contrato:** Define claramente las responsabilidades de cada componente.

2.2 Excepciones en Java

Las excepciones son mecanismos para manejar errores y situaciones excepcionales en tiempo de ejecución:

Concepto	Descripción
Try-Catch-Finally	Estructura para capturar y manejar errores de forma controlada
Excepciones Checked	Deben ser declaradas o capturadas (ej: IOException, FileNotFoundException)
Excepciones Unchecked	No requieren declaración explícita (ej: NullPointerException, ArithmeticException)

Excepciones Personalizadas	Clases que extienden Exception para modelar errores específicos del dominio
Try-with-resources	Cierre automático de recursos que implementan AutoCloseable
Throw y Throws	Lanzamiento y declaración de excepciones respectivamente

3. DESARROLLO DEL CASO PRÁCTICO

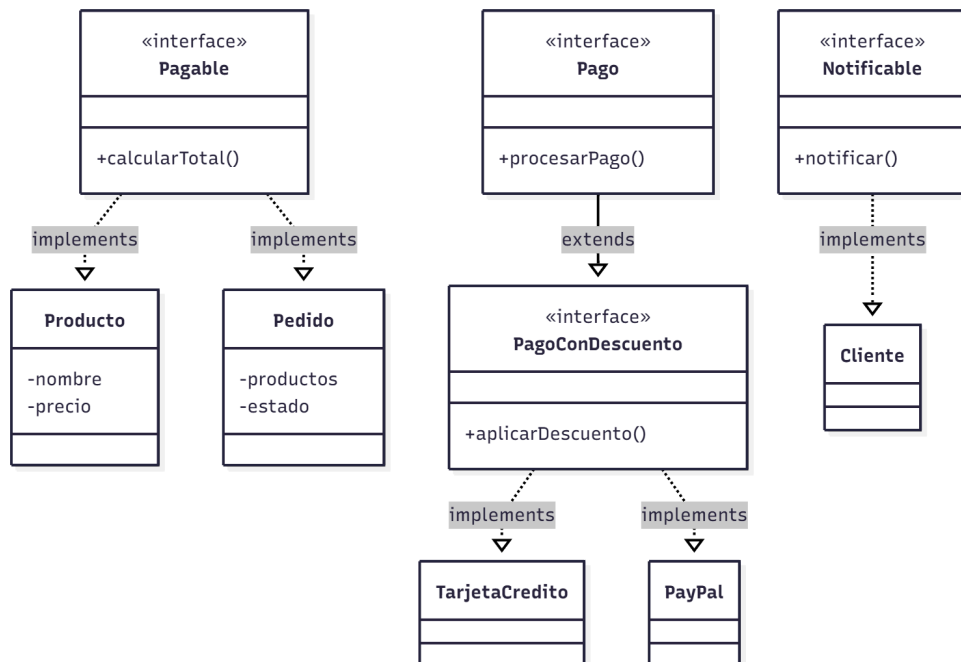
PARTE 1: INTERFACES EN UN SISTEMA DE E-COMMERCE

3.1 Análisis del Problema

Se requiere desarrollar un sistema de comercio electrónico que permita:

- Gestionar productos con precios
- Crear pedidos con múltiples productos
- Calcular totales de manera polimórfica
- Procesar pagos con diferentes medios (Tarjeta de Crédito, PayPal)
- Aplicar descuentos según el medio de pago
- Notificar a los clientes sobre cambios en sus pedidos

3.2 Diseño de Interfaces



3.3 Implementación de Interfaces

3.3.1 Interfaz Pagable

Esta interfaz define el contrato para todos los elementos que pueden calcular un total:

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

Justificación del diseño:

- Permite que tanto productos individuales como pedidos completos puedan calcular su total.
- Facilita el polimorfismo: podemos tratar productos y pedidos de manera uniforme.
- Cumple con el Principio de Segregación de Interfaces (ISP).

3.3.2 Clase Producto

Implementa **Pagable** representando un producto individual:

```
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 this.precio;  
    }  
  
    // Getters, setters y toString()...  
}
```

Decisiones de diseño:

- **Encapsulamiento:** atributos privados con acceso controlado.
- **Implementación simple de `calcularTotal()`:** retorna el precio.
- Método **`toString()`** para representación textual clara.

3.3.3 Clase Pedido

Implementa **Pagable** representando un pedido con múltiples productos:

```
public class Pedido implements Pagable {
    private int numeroPedido;
    private List<Producto> productos;
    private String estado;
    private Notificable cliente;

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

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

    public void cambiarEstado(String nuevoEstado) {
        String estadoAnterior = this.estado;
        this.estado = nuevoEstado;

        if (cliente != null) {
            String mensaje = String.format(
                "Pedido #%d cambió de estado: %s → %s",
                numeroPedido, estadoAnterior, nuevoEstado
            );
            cliente.notificar(mensaje);
        }
    }

    // Métodos adicionales...
}
```

Patrones aplicados:

- **Composite:** Un pedido contiene múltiples productos y calcula el total sumando los totales individuales.
- **Observer:** Notifica al cliente cuando cambia el estado del pedido.

3.4 Herencia Múltiple con Interfaces**3.4.1 Interfaces de Pago****Demostración de herencia de interfaces:**

```
interface Pago {
    boolean procesarPago(double monto);
}

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

Ventaja clave: **PagoConDescuento** hereda el método **procesarPago()** de **Pago** y añade **aplicarDescuento()**. Las clases que implementen **PagoConDescuento** deben implementar ambos métodos.

3.4.2 Implementación: TarjetaCredito

```
public class TarjetaCredito implements PagoConDescuento {
    private String numeroTarjeta;
    private String titular;
    private double porcentajeDescuento;

    @Override
    public boolean procesarPago(double monto) {
        System.out.println("Procesando pago con tarjeta...");
        // Lógica de procesamiento
        return true;
    }

    @Override
    public double aplicarDescuento(double monto) {
        double descuento = monto * porcentajeDescuento;
        return monto - descuento;
    }
}
```

Análisis:

- Implementa los dos métodos requeridos por **PagoConDescuento**.
- Simula el procesamiento real de un pago con tarjeta.
- Calcula descuentos basados en un porcentaje configurable.

3.4.3 Implementación: PayPal

Similar a TarjetaCredito pero con lógica específica de PayPal:

```
public class PayPal implements PagoConDescuento {
    private String email;
    private double porcentajeDescuento;

    @Override
    public boolean procesarPago(double monto) {
        System.out.println("Procesando pago con PayPal...");
        System.out.println("Cuenta: " + email);
        return true;
    }

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

3.5 Sistema de Notificaciones (Patrón Observer)**3.5.1 Interfaz Notificable**

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


3.5.2 Clase Cliente

```
public class Cliente implements Notificable {
    private String nombre;
    private String email;
    private String telefono;

    @Override
    public void notificar(String mensaje) {
        System.out.println("\n===== NOTIFICACIÓN =====");
        System.out.println("Cliente: " + nombre);
        System.out.println("Email: " + email);
        System.out.println("Mensaje: " + mensaje);
        // En producción: enviar email/SMS real
    }
}
```

Implementación del patrón Observer:

- El **Pedido** (Observable) mantiene referencia a un **Cliente** (Observer).
- Cuando el pedido cambia de estado, notifica automáticamente al cliente.
- Bajo acoplamiento: el pedido no necesita conocer detalles del cliente.

PARTE 2: EJERCICIOS SOBRE EXCEPCIONES

4.1 División Segura (ArithmeticException)

Objetivo: Manejar la división por cero de forma controlada.

```
public static void divisionSegura() {
    Scanner scanner = new Scanner(System.in);

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

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

        int resultado = dividendo / divisor;
        System.out.println("Resultado: " + resultado);
    }
}
```

```
} catch (ArithmeticException e) {  
    System.err.println("ERROR: No se puede dividir por cero");  
} catch (Exception e) {  
    System.err.println("ERROR: Entrada inválida");  
} finally {  
    System.out.println("Operación finalizada");  
}  
}
```

Análisis:

- **Try:** Código que puede lanzar excepciones.
- **Catch específico:** Captura **ArithmeticException** para división por cero.
- **Catch genérico:** Captura otras excepciones (ej: entrada no numérica).
- **Finally:** Se ejecuta siempre, útil para limpieza de recursos.

4.2 Conversión de Cadena a Número (NumberFormatException)

Objetivo: Validar entrada del usuario antes de convertir a número.

```
public static void conversionCadenaNumero() {  
    Scanner scanner = new Scanner(System.in);  
  
    System.out.print("Ingrese un número: ");  
    String texto = scanner.nextLine();  
  
    try {  
        int numero = Integer.parseInt(texto);  
        System.out.println("Conversión exitosa: " + numero);  
        System.out.println("El doble es: " + (numero * 2));  
    } catch (NumberFormatException e) {  
        System.err.println("ERROR: '" + texto + "' no es un número  
válido");  
        System.err.println("Debe ingresar solo dígitos");  
    }  
}
```

Conceptos aplicados:

- Captura de **NumberFormatException** lanzada por **parseInt()**.
- Mensaje de error informativo que incluye el valor ingresado.
- Continúa la ejecución del programa tras el error.

4.3 Lectura de Archivo (FileNotFoundException)

Objetivo: Manejar archivos inexistentes de forma elegante.

```
public static void lecturaArchivo(String nombreArchivo) {
    BufferedReader reader = null;

    try {
        reader = new BufferedReader(new FileReader(nombreArchivo));

        String linea;
        int numeroLinea = 1;

        while ((linea = reader.readLine()) != null) {
            System.out.println(numeroLinea + ": " + linea);
            numeroLinea++;
        }

    } catch (FileNotFoundException e) {
        System.err.println("ERROR: Archivo no encontrado");
    } catch (IOException e) {
        System.err.println("ERROR al leer el archivo");
    } finally {
        if (reader != null) {
            try {
                reader.close();
            } catch (IOException e) {
                System.err.println("Error al cerrar archivo");
            }
        }
    }
}
```

Buenas prácticas demostradas:

- Múltiples catch para diferentes tipos de excepciones.
- Finally para cerrar recursos (evita memory leaks).
- Manejo de excepciones incluso al cerrar el archivo.

4.4 Excepción Personalizada (EdadInvalidaException)

Objetivo: Crear excepciones específicas del dominio de negocio.

4.4.1 Definición de la Excepción

```
public class EdadInvalidaException extends Exception {  
  
    public EdadInvalidaException() {  
        super("Edad inválida: debe estar entre 0 y 120 años");  
    }  
  
    public EdadInvalidaException(String mensaje) {  
        super(mensaje);  
    }  
  
    public EdadInvalidaException(String mensaje, Throwable causa) {  
        super(mensaje, causa);  
    }  
}
```

Decisiones de diseño:

- Extiende **Exception** (checked) para forzar su manejo.
- Tres constructores para diferentes escenarios.
- Permite encadenamiento de excepciones con **Throwable causa**.

4.4.2 Uso de la Excepción

```
public static void validarEdad(int edad) throws EdadInvalidaException  
{  
    if (edad < 0) {  
        throw new EdadInvalidaException(  
            "La edad no puede ser negativa: " + edad  
        );  
    }  
  
    if (edad > 120) {  
        throw new EdadInvalidaException(  
            "La edad no puede ser mayor a 120 años: " + edad  
        );  
    }  
  
    System.out.println("Edad válida: " + edad + " años");  
}
```

```
// Uso:
try {
    validarEdad(-5);
} catch (EdadInvalidaException e) {
    System.err.println(e.getMessage());
}
```

Ventajas:

- Expresa reglas de negocio de forma clara.
- Más semántico que usar excepciones genéricas.
- Facilita el manejo diferenciado de errores.

4.5 Try-with-Resources (Buenas Prácticas)

Objetivo: Demostrar el cierre automático de recursos.

Antes (Java 6):

```
BufferedReader reader = null;
try {
    reader = new BufferedReader(new FileReader("archivo.txt"));
    String linea = reader.readLine();
} finally {
    if (reader != null) {
        try {
            reader.close();
        } catch (IOException e) {
            // Manejar error de cierre
        }
    }
}
```

Después (Java 7+):

```
try (BufferedReader reader = new BufferedReader(
    new FileReader("archivo.txt"))) {

    String linea;
    while ((linea = reader.readLine()) != null) {
        System.out.println(linea);
    }
    // reader.close() se llama AUTOMÁTICAMENTE

} catch (FileNotFoundException e) {
    System.err.println("Archivo no encontrado");
} catch (IOException e) {
    System.err.println("Error de lectura");
}
```

Ventajas de Try-with-Resources:

- Código más limpio y legible.
- Previene memory leaks.
- Cierre automático incluso si hay excepción.
- Puede manejar múltiples recursos: `try (R1 r1 = ...; R2 r2 = ...).`
- Forma recomendada desde Java 7.

4. CONCLUSIONES

4.1 Sobre Interfaces

Cumplimiento de objetivos:

1. **Diseño desacoplado:** Las interfaces permitieron crear un sistema donde las implementaciones pueden cambiar sin afectar el código cliente. Por ejemplo, podemos agregar nuevos medios de pago sin modificar la clase **Pedido**.
2. **Herencia múltiple:** Se demostró que Java permite heredar comportamiento de múltiples interfaces. La interfaz **PagoConDescuento** extiende **Pago**, y las clases **TarjetaCredito** y **PayPal** implementan ambos comportamientos.
3. **Polimorfismo efectivo:** A través de la interfaz **Pagable**, pudimos tratar productos y pedidos de manera uniforme, simplificando el código que calcula totales.
4. **Patrón Observer:** La interfaz **Notificable** permitió implementar el patrón Observer de forma elegante, facilitando la comunicación entre pedidos y clientes.

4.2 Sobre Excepciones

Cumplimiento de objetivos:

1. **Robustez del programa:** El manejo correcto de excepciones evitó que el programa termine abruptamente ante errores. Todos los errores fueron capturados y manejados apropiadamente.
2. **Excepciones personalizadas:** La creación de `EdadInvalidaException` demostró cómo expresar reglas de negocio de forma clara y específica, mejorando la legibilidad del código.
3. **Try-with-resources:** Se aplicó la mejor práctica de Java 7+ para manejo de recursos, garantizando su cierre automático y previniendo memory leaks.
4. **Distinción de tipos:** Se implementaron correctamente tanto excepciones checked (`FileNotFoundException`, `EdadInvalidaException`) como unchecked (`ArithmeticException`, `NumberFormatException`).

4.3 Integración de Conceptos

El trabajo práctico logró integrar exitosamente interfaces y excepciones en un sistema coherente:

- Las interfaces proporcionaron la estructura y contratos del sistema.
- Las excepciones garantizaron la robustez y manejo de errores.
- Ambos conceptos trabajaron juntos para crear un sistema profesional y mantenible.

4.4 Aplicabilidad Práctica

Los conceptos aprendidos son fundamentales en el desarrollo profesional:

- **Interfaces:** Base del diseño orientado a objetos, patrones de diseño y arquitecturas modernas.
- **Excepciones:** Esenciales para aplicaciones robustas en producción.
- **Combinación:** Permiten crear sistemas escalables y mantenibles.

4.5 Aprendizajes Clave

1. Las interfaces permiten separar "qué hace" del "cómo lo hace".
2. La herencia múltiple de interfaces es más flexible que la herencia de clases.
3. El manejo de excepciones es crucial para la experiencia del usuario.
4. Try-with-resources simplifica significativamente el código.
5. Las excepciones personalizadas hacen el código más expresivo.

5. BIBLIOGRAFÍA Y REFERENCIAS

-
- Oracle Java Documentation - <https://docs.oracle.com/javase/tutorial/>
 - Effective Java - Joshua Bloch (3rd Edition)
 - Head First Design Patterns - Freeman & Freeman
 - Java: The Complete Reference - Herbert Schildt
 - Material de Cátedra - Programación II, UTN
-