UNIVERSIDAD TECNOLOGICA NACIONAL

Trabajo Práctico N° 8

Interfaces y Excepciones en Java

Comision 16

Macarena Marinoni <*mainonimacarena@gmail.com*>

Tecnicatura Universitaria en Programación - Universidad Tecnológica Nacional

Programación II

Profesor: Cortez, Alberto

Tutor: Bianchi, Neyén

16/11/2025

ÍNDICE

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.

Marco teórico

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

Caso práctico

Parte 1: Interfaces en un sistema de E-commerce

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

```java
...4 lines
package Tp8_InterfacesExcepciones;

/**...4 lines */

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

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

```java
package Tp8_InterfacesExcepciones;

/**...4 lines */
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;
    }
}
```

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

```java
...4 lines
package Tp8_InterfacesExcepciones;

/**...4 lines */
public enum EstadoPedido {
    CREADO, PROCESANDO, ENVIADO, ENTREGADO
}
```

```java
...4 lines
package Tp8_InterfacesExcepciones;

/**...4 lines */
public interface Notificable {
    void notificar(String mensaje);
}
```

```java
...4 lines

package Tp8_InterfacesExcepciones;

/**...4 lines */
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 notificar(String mensaje) {
        System.out.println("Notificación para " + nombre + " (" + email + "): " + mensaje);
    }
}
```

```java
package Tp8_InterfacesExcepciones;

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

/**...4 lines */
public class Pedido implements Pagable {

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

    public Pedido(Cliente cliente) {
        this.productos = new ArrayList<>();
        this.estado = EstadoPedido.CREADO;
        this.cliente = cliente;
        notificarEstado();
    }

    public void agregarProducto(Producto p) {
        productos.add(p);
        cliente.notificar("Se agregó el producto: " + p);
    }

    public void cambiarEstado(EstadoPedido nuevoEstado) {
        this.estado = nuevoEstado;
        notificarEstado();
    }

    private void notificarEstado() {
        cliente.notificar("El pedido cambió al estado: " + estado);
    }

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

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

```java
...4 lines
package Tp8_InterfacesExcepciones;

/**...4 lines */
public interface Pago {
    void procesarPago(double monto);
}
```

```java
...4 lines
package Tp8_InterfacesExcepciones;

/**...4 lines */

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

```java
...4 lines

package Tp8_InterfacesExcepciones;

/**...4 lines */
public class TarjetaCredito implements Pago, PagoConDescuento {

    @Override
    public double aplicarDescuento(double monto) {
        return monto * 0.90; // 10% descuento
    }

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

```java
...4 lines

package Tp8_InterfacesExcepciones;

/**...4 lines */
public class PayPal implements Pago {

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

```java
...4 lines

package Tp8_InterfacesExcepciones;

/**...4 lines */
public class KataInterfacesMain {

    public static void main(String[] args) {

        Cliente cliente = new Cliente("Marianela", "maru@example.com");

        Pedido pedido = new Pedido(cliente);

        pedido.agregarProducto(new Producto("Notebook", 500000));
        pedido.agregarProducto(new Producto("Mouse", 15000));

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

        pedido.cambiarEstado(EstadoPedido.PROCESANDO);
        pedido.cambiarEstado(EstadoPedido.ENVIADO);

        // Pago con tarjeta
        Pago tarjeta = new TarjetaCredito();
        double montoConDescuento = ((TarjetaCredito) tarjeta).aplicarDescuento(pedido.calcularTotal());
        tarjeta.procesarPago(montoConDescuento);

        // Pago con PayPal
        Pago paypal = new PayPal();
        paypal.procesarPago(pedido.calcularTotal());
    }
}
```

```
run:
Notificaci�n para Marianela (maru@example.com): El pedido cambi� al estado: CREADO
Notificaci�n para Marianela (maru@example.com): Se agreg� el producto: Notebook - $500000.0
Notificaci�n para Marianela (maru@example.com): Se agreg� el producto: Mouse - $15000.0
Total del pedido: $515000.0
Notificaci�n para Marianela (maru@example.com): El pedido cambi� al estado: PROCESANDO
Notificaci�n para Marianela (maru@example.com): El pedido cambi� al estado: ENVIADO
Pago procesado con Tarjeta de Cr�dito por $463500.0
Pago procesado v�a PayPal por $515000.0
BUILD SUCCESSFUL (total time: 0 seconds)
```

La primera parte del trabajo permitió aplicar los conceptos fundamentales de **interfaces** en Java mediante la simulación de un sistema simple de e-commerce.

A través de la creación de las **interfaces Pagable, Pago, PagoConDescuento y Notificable**, se logró definir contratos de comportamiento que luego fueron implementados por distintas clases, permitiendo diseñar un modelo flexible y extensible.

Las **clases Producto y Pedido** implementan la **interfaz Pagable**, demostrando cómo una misma operación puede aplicarse de forma uniforme a objetos diferentes. La **interfaz Notificable** permitió incorporar un mecanismo de comunicación directa con el cliente cada vez que el estado del pedido cambiaba, lo cual refuerza el uso de interfaces para lograr una arquitectura desacoplada.

Finalmente, las **clases TarjetaCredito y PayPal** mostraron cómo diferentes métodos de pago pueden coexistir y aplicarse indistintamente gracias al uso de interfaces y polimorfismo.

En conjunto, esta parte del TP evidencia cómo las interfaces permiten diseñar sistemas más robustos, escalables y fáciles de mantener, ya que el comportamiento se define de manera abstracta y se implementa según las necesidades de cada clase concreta.

El resultado es una solución organizada, modular y alineada con las buenas prácticas de la Programación Orientada a Objetos.

Parte 2: Ejercicios sobre Excepciones

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

En este ejercicio se implementa un mecanismo de división segura utilizando bloques try-catch. El programa solicita al usuario dos valores enteros y realiza la división, capturando la excepción ArithmeticException en caso de que el divisor sea cero, y InputMismatchException si se ingresan datos no numéricos. Esto permite evitar errores en tiempo de ejecución y mejorar la robustez del programa.

```java
package Tp8_InterfacesExcepciones;

import java.util.InputMismatchException;
import java.util.Scanner;


/**...4 lines */
public class Ej1DivisionSegura {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.println("===== EJERCICIO 1 - División segura =====");

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

            System.out.print("Ingresá el denominador: ");
            int denominador = scanner.nextInt();

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

        } catch (ArithmeticException e) {
            System.out.println("Error: no se puede dividir por cero.");
        } catch (InputMismatchException e) {
            System.out.println("Error: debés ingresar sólo números enteros.");
        } finally {
            System.out.println("Finalizando el ejercicio 1.\n");
            scanner.close();
        }
    }
}
```

```
run:
===== EJERCICIO 1 - Divisi□n segura =====
Ingres□ el numerador: 16
Ingres□ el denominador: 2
Resultado de la divisi□n: 8
Finalizando el ejercicio 1.

BUILD SUCCESSFUL (total time: 18 seconds)
```

2. Conversión de cadena a número

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

Este ejercicio solicita un texto al usuario e intenta convertirlo a un número entero mediante el método Integer.parseInt(). Si el contenido ingresado no es convertible a número, se captura la excepción NumberFormatException, mostrando un mensaje adecuado al usuario. El objetivo es validar entradas y garantizar que solo se procesen datos numéricos válidos.

```java
...4 lines

package Tp8_InterfacesExcepciones;

import java.util.Scanner;

/** ...4 lines */
public class Ej2ConversionNumero {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.println("===== EJERCICIO 2 - Conversión de cadena a número =====");

        System.out.print("Ingresá un número (como texto): ");
        String numeroTexto = scanner.nextLine();

        try {
            int numero = Integer.parseInt(numeroTexto);
            System.out.println("Conversión exitosa. El número es: " + numero);
        } catch (NumberFormatException e) {
            System.out.println("Error: el texto ingresado no es un número entero válido.");
        } finally {
            System.out.println("Finalizando el ejercicio 2.\n");
            scanner.close();
        }
    }
}
```

```
run:
===== EJERCICIO 2 - Conversi□n de cadena a n□mero =====
Ingres□ un n□mero (como texto): 45
Conversi□n exitosa. El n□mero es: 45
Finalizando el ejercicio 2.

BUILD SUCCESSFUL (total time: 2 seconds)
```

3. Lectura de archivo

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

Aquí se trabaja con operaciones de lectura de archivos, utilizando FileReader y BufferedReader. El programa solicita el nombre de un archivo y maneja posibles errores como FileNotFoundException si el archivo no existe, e IOException ante cualquier problema de lectura. Además, se emplea el bloque finally para asegurar el cierre del recurso, reforzando buenas prácticas en manejo de archivos.

```java
...4 lines

package Tp8_InterfacesExcepciones;

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

/**...4 lines */
public class Ej3LecturaArchivo {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.println("===== EJERCICIO 3 - Lectura de archivo =====");
        System.out.print("Ingresá el nombre del archivo (con extensión): ");
        String nombreArchivo = scanner.nextLine();

        BufferedReader br = null;

        try {
            br = new BufferedReader(new FileReader(nombreArchivo));
            String linea;
            System.out.println("\nContenido del archivo:");
            while ((linea = br.readLine()) != null) {
                System.out.println(linea);
            }

        } catch (FileNotFoundException e) {
            System.out.println("Error: el archivo '" + nombreArchivo + "' no fue encontrado.");
        } catch (IOException e) {
            System.out.println("Error de lectura del archivo: " + e.getMessage());
        } finally {
            System.out.println("\nFinalizando el ejercicio 3.");
            try {
                if (br != null) {
                    br.close();
                }
            } catch (IOException e) {
```

```
===== EJERCICIO 3 - Lectura de archivo =====
Ingres  el nombre del archivo (con extensi n): "Animales.xls"
Error: el archivo '"Animales.xls"' no fue encontrado.

Finalizando el ejercicio 3.
BUILD SUCCESSFUL (total time: 14 seconds)
```

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

En este ejercicio se define una excepción personalizada llamada EdadInvalidaException para validar valores de edad. El programa solicita al usuario ingresar una edad y lanza la excepción si el número está fuera de un rango razonable. Se demuestra cómo crear excepciones propias y cómo utilizarlas para reforzar las reglas de negocio dentro de una aplicación.

```java
...4 lines

package Tp8_InterfacesExcepciones;

/**...4 lines */
public class EdadInvalidaException extends Exception {

    public EdadInvalidaException(String mensaje) {
        super(mensaje);
    }
}
```

```java
...4 lines

package Tp8_InterfacesExcepciones;
import java.util.InputMismatchException;
import java.util.Scanner;


/**...4 lines */
public class Ej4EdadInvalida {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.println("===== EJERCICIO 4 - Excepción personalizada EdadInvalidaException =====");

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

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

        } catch (InputMismatchException e) {
            System.out.println("Error: debés ingresar un número entero para la edad.");
        } catch (EdadInvalidaException e) {
            System.out.println("Error de edad: " + e.getMessage());
        } finally {
            System.out.println("Finalizando el ejercicio 4.\n");
            scanner.close();
        }
    }

    // Método que lanza la excepción personalizada
    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.");
        }
    }
}
```

```
run:
===== EJERCICIO 4 - Excepci□n personalizada EdadInvalidaException =====
Ingres□ tu edad: 35
Edad v□lida: 35
Finalizando el ejercicio 4.

BUILD SUCCESSFUL (total time: 3 seconds)
```

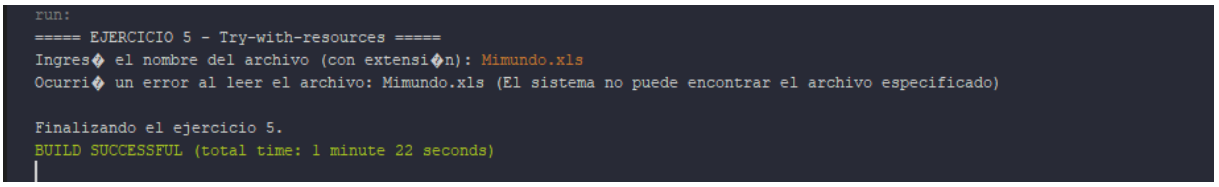
```
run:
===== EJERCICIO 4 - Excepci□n personalizada EdadInvalidaException =====
Ingres□ tu edad: 250
Error de edad: La edad debe estar entre 0 y 120 a□os.
Finalizando el ejercicio 4.

BUILD SUCCESSFUL (total time: 4 seconds)
```

5. Uso de try-with-resources

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

Este ejercicio retoma la lectura de archivos, pero utilizando el mecanismo moderno de try-with-resources, que permite abrir recursos (como archivos) y cerrarlos automáticamente sin necesidad de un bloque finally. Se capturan posibles IOException y se demuestra una forma más limpia y segura de trabajar con recursos externos en Java.

```java
package Tp8_InterfacesExcepciones;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.Scanner;

/**...4 lines */
public class Ej5TryWithResources {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.println("===== EJERCICIO 5 - Try-with-resources =====");
        System.out.print("Ingresá el nombre del archivo (con extensión): ");
        String nombreArchivo = scanner.nextLine();

        // try-with-resources: el BufferedReader se cierra solo
        try (BufferedReader br = new BufferedReader(new FileReader(nombreArchivo))) {

            String linea;
            System.out.println("\nContenido del archivo:");
            while ((linea = br.readLine()) != null) {
                System.out.println(linea);
            }

        } catch (IOException e) {
            System.out.println("Ocurrió un error al leer el archivo: " + e.getMessage());
        } finally {
            System.out.println("\nFinalizando el ejercicio 5.");
            scanner.close();
        }
```



En esta segunda parte del trabajo se exploraron distintos tipos de excepciones en Java, tanto propias del lenguaje como personalizadas. A través de los ejercicios desarrollados, se logró comprender cómo las excepciones permiten manejar errores de manera controlada, evitando interrupciones inesperadas en el programa y asegurando un flujo de ejecución seguro.

Se aplicaron diferentes técnicas, como el uso de múltiples bloques **catch**, validaciones de tipo de dato, manejo de archivos con **try-catch-finally** y el mecanismo **try-with-resources**. Además, se incorporó la creación de una excepción personalizada, demostrando cómo adaptar este concepto a las reglas específicas de una aplicación.

En conjunto, esta parte del TP permitió reforzar el uso de excepciones para mejorar la robustez, fiabilidad y mantenibilidad del código, cumpliendo con los objetivos planteados para la unidad.

Link a Repositorio Github

https://github.com/MaquiMarinoni/TUPaD_P2-C22025.git