Informe del Trabajo Practico 7

Ejercicio 1:

# **Informe del Sistema de Validación de Contraseñas**

## **Introducción**

El presente informe describe el desarrollo de un programa en Java que implementa un sistema de **validación de contraseñas** utilizando **excepciones personalizadas**. El objetivo principal es garantizar que las contraseñas definidas por el usuario cumplan con ciertos requisitos de seguridad, y en caso de no cumplirse, notificar al usuario de manera clara mediante mensajes personalizados.

## **Objetivos del Sistema**

* Solicitar al usuario una contraseña desde la entrada estándar.
* Validar que la contraseña:
  + **No sea nula ni vacía**.
  + **Tenga más de 8 caracteres**.
  + **Incluya al menos un número** dentro de la cadena.
* Lanzar excepciones descriptivas en caso de que la contraseña no cumpla con los criterios establecidos.
* Informar al usuario el resultado de la validación y finalizar el proceso de forma ordenada.

## **Diseño e Implementación**

### **1. Clase ValidacionContraseña**

Esta clase contiene el método estático ValidarContraseña(String contraseña), el cual concentra toda la lógica de validación:

* Si la contraseña es null o vacía, se lanza una **IllegalArgumentException**.
* Si la longitud es menor o igual a 8 caracteres, se lanza una **ExcepcionPersonalizada** con un mensaje orientativo.
* Si no contiene al menos un dígito, se lanza otra **ExcepcionPersonalizada**.
* Si supera todas las validaciones, se confirma que la contraseña es válida.

### **2. Clase ExcepcionPersonalizada**

Se definió una excepción propia que extiende de RuntimeException, permitiendo encapsular mensajes específicos relacionados con las reglas de negocio del sistema.

### **3. Clase Main**

El punto de entrada de la aplicación:

* Utiliza un objeto Scanner para solicitar la contraseña al usuario.
* Llama al método de validación dentro de un bloque try-catch-finally.
* Maneja los distintos errores con mensajes claros para el usuario:
  + IllegalArgumentException para errores de nulidad o vacío.
  + ExcepcionPersonalizada para errores de longitud o falta de números.
* Incluye un bloque finally para indicar la finalización del proceso sin importar el resultado.

## **Resultados Obtenidos**

* El sistema valida correctamente las contraseñas ingresadas por el usuario.
* Los mensajes de error son **claros y específicos**, lo que mejora la experiencia de usuario.
* El programa es robusto frente a entradas inválidas, ya que evita contraseñas inseguras.

## **Posibles Errores Detectados**

1. **Errores lógicos (posibles mejoras):**
   1. Actualmente, el sistema **no exige caracteres especiales** ni letras mayúsculas, lo que podría considerarse una debilidad de seguridad.
   2. El mensaje "¡Error!" se concatena dos veces en algunos casos, lo que puede generar confusión.
   3. El bloque finally siempre imprime “El proceso finalizó…”, incluso cuando hubo un error grave. Esto puede interpretarse como una terminación correcta cuando en realidad no lo fue.

## **Conclusiones**

El ejercicio permitió:

* Aplicar conceptos de **validación de datos de entrada**.
* Implementar **excepciones personalizadas**, lo que otorga mayor flexibilidad al manejo de errores.
* Construir un programa modular y escalable, ya que las reglas de validación pueden ampliarse fácilmente (por ejemplo: exigir caracteres especiales o mayúsculas).

Ejercicio 2:

# **Informe del Sistema de Validación de Calificaciones de Estudiantes**

## **Introducción**

El presente informe describe la implementación en Java de un programa que valida las calificaciones de un estudiante, garantizando que se encuentren dentro de un rango aceptado (0 a 10). Para lograrlo, se utilizaron **excepciones personalizadas** que permiten controlar y notificar al usuario cuando se ingresan datos incorrectos.

## **Objetivos**

* Solicitar al usuario una calificación por consola.
* Verificar que la calificación:
  + Sea un valor **numérico**.
  + Se encuentre en el rango **[0, 10]**.
* Lanzar mensajes de error claros en caso de que los valores sean inválidos.
* Mostrar confirmación cuando la calificación sea válida.

## **Diseño del Sistema**

### **1. Clase calificacionestudiante**

* Contiene el método estático NotaEstudiante(Double nota).
* Valida que el valor ingresado esté en el rango permitido:
  + Si es menor que 0 o mayor que 10, lanza una excepción Calificacion.
* Se utilizó un condicional con **operador OR (||)** para simplificar la verificación de los límites.

### **2. Clase Calificacion**

* Hereda de RuntimeException.
* Permite definir excepciones personalizadas con mensajes específicos para cuando una nota está fuera de rango.

### **3. Clase Main**

* Pide al usuario que ingrese una nota mediante un Scanner.
* Contiene un bloque try-catch que captura tres posibles escenarios:
  + **Nota fuera de rango (0–10):** se lanza Calificacion.
  + **Error de tipo de dato:** si el usuario ingresa un valor no numérico (ejemplo: letras), se captura con RuntimeException.
  + **Nota válida:** se imprime un mensaje de confirmación.
* Cierra el Scanner al finalizar el proceso.

## **Posibles Errores Detectados**

1. **Errores lógicos detectados durante el desarrollo:**
   1. Al concatenar + nota al mensaje de la excepción, el número ingresado se imprimía junto al texto, lo que generaba confusión en la salida. Esto fue corregido eliminando la concatenación innecesaria.

## **Conclusiones**

El sistema de validación de calificaciones cumple con los objetivos planteados, asegurando que las notas ingresadas sean **válidas y numéricas**. La implementación de excepciones personalizadas (Calificacion) mejora la claridad de los mensajes de error y refuerza la robustez del programa.

Ejercicio 3:

# **Informe del Sistema de Reservas de Asientos de Cine**

## **Introducción**

El presente informe describe el desarrollo de un sistema de reservas de asientos para una sala de cine denominado **CineSunstarSolar**. El objetivo del sistema es simular la gestión de asientos de manera interactiva, validando diferentes condiciones mediante el uso de **excepciones personalizadas**.

## **Objetivos del Sistema**

* Simular una sala de cine con una capacidad definida por el usuario.
* Inicializar los asientos de modo que los números **impares estén ocupados** y los **pares libres**.
* Permitir que los usuarios seleccionen un asiento para reservar.
* Validar diferentes reglas de negocio a través de excepciones:
  + **AsientoInvalido**: cuando el número de asiento no existe.
  + **AsientosOcupados**: cuando se intenta reservar un asiento ya ocupado.
  + **Cinelleno**: cuando todos los asientos se encuentran ocupados.

## **Diseño e Implementación**

### **1. Entidad Asiento**

Cada asiento posee un número y un estado (ocupado o libre). La clase incluye el método guardarLugar() que controla la lógica de reserva y lanza una excepción en caso de estar ya ocupado.

### **2. Clase CineSunstarSolar**

Esta clase representa la sala de cine.

* Al instanciarse, inicializa los asientos según la capacidad ingresada.
* Dispone de métodos para:
  + **mostrarEstado()**: despliega en consola los asientos, indicando con una X los ocupados.
  + **asientosinvalidos()**: valida si un número de asiento existe y muestra si está ocupado o libre.
  + **reservarAsiento()**: permite reservar un asiento y lanza excepción si ya está ocupado.
  + **cineLLeno()**: verifica si la sala alcanzó su máxima capacidad y en ese caso impide más reservas.

### **3. Excepciones Personalizadas**

* AsientoInvalido: asegura que solo se seleccionen asientos dentro del rango permitido.
* AsientosOcupados: evita reservar un asiento previamente ocupado.
* Cinelleno: interrumpe la ejecución cuando todos los asientos están reservados.

### **4. Interfaz Main**

Se implementó un menú interactivo en consola donde el usuario:

1. Visualiza el estado actual de los asientos.
2. Ingresa el número del asiento que desea reservar.
3. Recibe retroalimentación inmediata si el asiento está libre, ocupado, o inválido.
4. El sistema finaliza cuando el usuario ingresa 0 o cuando la sala se encuentra llena.

### **Errores Lógicos del Sistema**

1. **Inicialización de Asientos**
   1. Actualmente todos los **asientos impares inician como ocupados** y los pares como libres.
   2. Esto puede no coincidir con el objetivo real del sistema (normalmente todos los asientos deberían comenzar libres).
   3. Consecuencia: desde el inicio el usuario encuentra la mitad de la sala ocupada sin haber reservado nada.

## **Conclusiones**

El desarrollo de este sistema permitió:

* Implementar un modelo de **orientación a objetos** para representar entidades del mundo real (asientos y sala de cine).
* Aplicar el concepto de **excepciones personalizadas** para validar condiciones específicas y garantizar el correcto funcionamiento del sistema.
* Construir una aplicación interactiva, clara y con un flujo lógico para el usuario.