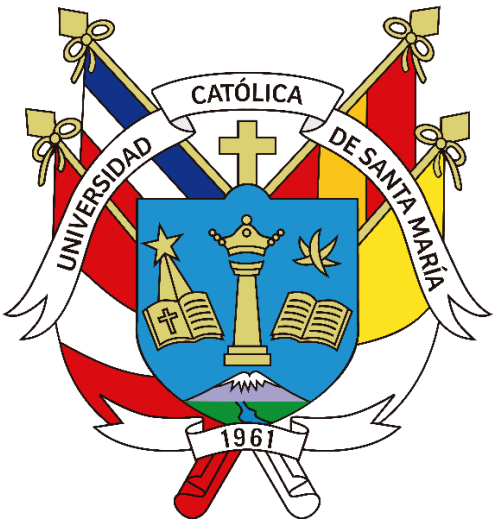
UNIVERSIDAD CATÓLICA DE SANTA MARÍA

FACULTAD DE CIENCIAS FÍSICAS Y FORMALES

ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMAS

LENGUAJES DE PROGRAMACIÓN III



Manejo de excepciones

Autor

Cusirramos Chiri Santiago Jesus

Flores Gutierrez Mateo Jesus

Loaiza Cruz Joaquin Armando

Arequipa, 2024

Resumen ejecutivo

El presente informe tiene como objetivo abordar el manejo de excepciones en Java, una herramienta esencial para garantizar la estabilidad y robustez de las aplicaciones, permitiendo que los errores sean gestionados de manera controlada en lugar de interrumpir abruptamente el flujo del programa. A lo largo del documento, se desarrollará una introducción teórica sobre las excepciones, su jerarquía y clasificación, diferenciando entre excepciones comprobadas y no comprobadas, así como el uso de los bloques try-catch-finally para capturar y manejar estos eventos anómalos. Se analizará el impacto de las excepciones en la programación orientada a objetos y cómo contribuyen a un código más seguro y eficiente. En cuanto a la parte práctica, se realizarán diversas actividades centradas en la implementación de excepciones en un sistema bancario, en el cual se simularán operaciones comunes como la creación de cuentas, depósitos, retiros, transferencias entre cuentas y generación de reportes de transacciones. Se hará especial énfasis en la creación y uso de excepciones personalizadas para escenarios específicos, tales como SaldoInsuficienteException, CuentaNoEncontradaException, y LimiteCreditoExcedidoException, las cuales permitirán un control más preciso sobre los posibles errores que puedan surgir durante la ejecución del sistema. Además, se utilizarán estructuras de control como try-with-resources para garantizar la correcta liberación de recursos como archivos y conexiones, incluso en caso de que se produzcan excepciones. El informe también incluye un conjunto de ejercicios propuestos, como la creación de calculadoras con validación de divisiones por cero, sistemas de registro de estudiantes con manejo de excepciones, y simulaciones de transacciones inmobiliarias, que pondrán a prueba la comprensión y aplicación del manejo de excepciones por parte del lector. Finalmente, se presentará un cuestionario que busca afianzar los conceptos aprendidos, abordando preguntas sobre las diferencias entre lanzar y capturar excepciones, los distintos tipos de excepciones en Java, y cómo manejar adecuadamente los recursos cuando ocurren errores. Todo esto tiene el fin de proporcionar al estudiante una comprensión profunda de cómo gestionar las excepciones en Java para crear aplicaciones más seguras y fiables.

Introducción

En el desarrollo de aplicaciones informáticas, el manejo de errores es una parte fundamental para asegurar que los programas funcionen de manera correcta y eficiente, incluso en situaciones imprevistas. En este contexto, el manejo de excepciones juega un rol clave, ya que permite detectar y gestionar de forma controlada los errores o situaciones anómalas que pueden surgir durante la ejecución de un programa. Las excepciones en Java, al igual que en muchos otros lenguajes de programación, son mecanismos que permiten a los desarrolladores capturar y manejar condiciones de error sin que esto implique la interrupción abrupta del programa, lo cual es crucial en sistemas donde la continuidad y estabilidad son esenciales.

Una excepción es, en términos simples, un evento que ocurre durante la ejecución de un programa y que altera el flujo normal de instrucciones. En lugar de detenerse o fallar por completo, el programa puede "lanzar" (throw) una excepción, que luego será "capturada" (catch) por un bloque de código especializado en gestionarla. Esta captura permite que el programa pueda recuperarse del error o, al menos, tratarlo de manera adecuada, lo que proporciona una mayor robustez y fiabilidad al sistema. Este enfoque es particularmente útil en aplicaciones de misión crítica, como sistemas bancarios, redes de telecomunicaciones, y cualquier entorno en el que la mínima interrupción pueda tener consecuencias graves.

En Java, las excepciones siguen una jerarquía bien definida, donde todas las excepciones son descendientes de la clase Throwable. Dentro de esta jerarquía, podemos distinguir entre dos tipos principales de excepciones: las excepciones verificadas o comprobadas (checked exceptions), que son aquellas que deben ser declaradas o manejadas por el programador, y las excepciones no verificadas (unchecked exceptions), que derivan de errores en la lógica del programa y no requieren ser explicitamente gestionadas. Por ejemplo, un error común como el acceso fuera de los límites de un arreglo es una excepción no verificada, mientras que errores relacionados con la entrada/salida, como un archivo inexistente, son excepciones verificadas.

El manejo adecuado de estas excepciones no solo permite identificar y solucionar errores, sino que también mejora la experiencia del usuario al garantizar que la aplicación siga funcionando sin interrupciones, proporcionando mensajes informativos sobre lo que salió mal. Esto se logra a través de bloques de código try-catch-finally, donde el bloque try contiene el código que podría generar la excepción, el bloque catch define cómo manejar la excepción si ocurre, y el bloque finally asegura que ciertas tareas críticas, como el cierre de recursos, se realicen sin importar si ocurre o no una excepción.

Adicionalmente, en Java es posible crear excepciones personalizadas, diseñadas específicamente para reflejar errores únicos de la aplicación en desarrollo. Esto permite a los programadores definir tipos de errores más precisos que los predefinidos en Java, ofreciendo una mejor descripción de lo que ocurrió y facilitando su manejo. Por ejemplo, en un sistema bancario, podría crearse una excepción como SaldoInsuficienteException para señalar que un cliente intenta retirar más dinero del que tiene disponible, o una CuentaNoEncontradaException para indicar que una cuenta bancaria solicitada no existe en el sistema. Estas excepciones específicas permiten que el código sea más claro y que los errores se manejen de manera más precisa.

En resumen, el manejo de excepciones en Java no solo es una técnica para gestionar errores, sino también una práctica que mejora la calidad y robustez de las aplicaciones. A lo largo de este informe, se profundizará en los aspectos teóricos y prácticos del manejo de excepciones, proporcionando una guía detallada sobre cómo implementar sistemas que sean resilientes ante fallos y capaces de recuperarse de manera eficiente ante situaciones adversas. El objetivo final es capacitar al lector en la creación de programas más estables, asegurando que los errores se traten de manera adecuada sin afectar la experiencia del usuario ni comprometer la integridad del sistema.

Actividades

CONTEXTO DEL PROBLEMA: SISTEMA DE GESTIÓN DE CUENTAS BANCARIAS CON MANEJO DE EXCEPCIONES

Este proyecto consiste en desarrollar un Sistema de Gestión de Cuentas Bancarias en Java, aplicando los principios de Programación Orientada a Objetos (POO) y el manejo de excepciones. El sistema permitirá crear cuentas bancarias, realizar operaciones como depósitos, retiros, transferencias entre cuentas, consultas de saldo, cierre de cuentas, y ofrecerá funcionalidades adicionales como la generación de reportes de transacciones y la gestión de límites de crédito

Requisitos funcionales:

- Creación de cuentas bancarias.
- Depósito de dinero.
- Retiro de dinero.
- Transferencias entre cuentas.
- Cierre de cuentas si el saldo es cero.
- Consulta del saldo de una cuenta.
- Gestión de límites de crédito para cuentas especiales.
- Generación de reportes de transacciones.

Requisitos Técnicos:

- Manejo de excepciones proporcionadas por Java como IllegalArgumentException.
- Creación de excepciones personalizadas como SaldoInsuficienteException, CuentaNoEncontradaException, SaldoNoCeroException, y HistorialVacioException.
- Manejo de recursos utilizando bloques try-with-resources o finally para garantizar la correcta liberación de recursos, como archivos y Scanner.
- Aplicación de POO para modelar las operaciones y relaciones entre las clases.

EXPERIENCIA DE PRÁCTICA N° 01: MANEJO DE EXCEPCIONES INTERNAS Y DE VALIDACION DE DATOS

El objetivo de esta experiencia es que los estudiantes implementen el manejo de excepciones proporcionadas por Java y validen las entradas de usuario.

A) DISENO DE LA CLASE CUENTA BANCARIA:
   - Crear la clase CuentaBancaria con atributos numeroCuenta, titular y saldo.
   - Validar los datos al crear una cuenta: IllegalArgumentException: Lanzar cuando se intente crear una cuenta con saldo inicial negativo

B) OPERACIONES BÁSICAS CON MANEJO DE EXCEPCIONES
   - Implementar el método depositar(double monto), que valide que el monto sea positivo. Si no lo es, lanzar una IllegalArgumentException.
   - Implementar el método retirar(double monto), que verifique que el monto a retirar sea positivo y menor o igual al saldo disponible. Si no es así, lanzar una SaldoInsuficienteException (excepcion personalizada)

C) PRUEBAS EN ESTA EXPERIENCIA:
   - Crear cuentas con datos válidos e inválidos.
   - Probar depósitos y retiros con datos incorrectos para validar el manejo de excepciones.

EXPERIENCIA DE PRÁCTICA N° 02: CREACIÓN Y MANEJO DE

EXCEPCIONES PERSONALIZADAS

Esta experiencia tiene como objetivo que los estudiantes creen excepciones personalizadas y manejen casos específicos del sistema bancario.

A) CREACIÓN DE EXCEPCIONES PERSONALIZADAS

- SaldoInsuficienteException: Se lanza cuando no hay suficiente saldo para un retiro o transferencia.
- CuentaNoEncontradaException: Se lanza cuando una cuenta no existe en el sistema.
- SaldoNoCeroException: Se lanza cuando se intenta cerrar una cuenta con saldo positivo.

B) OPERACIONES AVANZADAS CON MANEJO DE EXCEPCIONES
- Implementar el método transferir(CuentaBancaria destino, double monto). Lanzar CuentaNoEncontradaException si la cuenta destino no existe, y SaldoInsuficienteException si el saldo de la cuenta origen es insuficiente.
- Implementar el método cerrarCuenta(), que permita cerrar una cuenta si el saldo es cero. Si no, lanzar SaldoNoCeroException.

C) PRUEBAS EN ESTA EXPERIENCIA:
- Probar transferencias entre cuentas válidas e inválidas.
- Intentar cerrar cuentas con y sin saldo y verificar el manejo correcto de excepciones.

EXPERIENCIA DE PRÁCTICA N° 03: GESTIÓN DE LÍMITES DE

CRÉDITO Y PROPAGACIÓN DE EXCEPCIONES RELACIONADAS

En esta experiencia, los estudiantes agregarán funcionalidad para gestionar límites de crédito en cuentas especiales y manejarán las excepciones asociadas las cuales deberán propagarse.

A) IMPLEMENTACIÓN DE LÍMITES DE CRÉDITO
- Crear una clase hija de CuentaBancaria llamada CuentaCredito, con un atributo adicional limiteCredito
- Modificar los métodos de retiro y transferencia para permitir retiros o transferencias que excedan el saldo, pero solo hasta el límite de crédito.
- Lanzar LimiteCreditoExcedidoException (excepción personalizada) si la operación supera el límite de crédito disponible. Esta debe ser propagada para que sea manejada en el método que invoca al método retiro.

B) PRUEBAS EN ESTA EXPERIENCIA
- Probar retiros y transferencias en cuentas con y sin límite de crédito
- Verificar que se lance correctamente la excepción LimiteCreditoExcedidoException cuando se supera el límite de crédito.

EXPERIENCIA DE PRÁCTICA N° 04: GENERACIÓN DE REPORTES DE TRANSACCIONES Y MANEJO DE RECURSOS

En esta experiencia, los estudiantes trabajarán en la generación de reportes de transacciones y el manejo de recursos como archivos y Scanner, garantizando la correcta liberación de estos recursos incluso cuando ocurran excepciones

A) GENERACIÓN DE REPORTES Y MANEJO DE ARCHIVOS:
- Implementar una clase ReporteTransacciones, que permita generar reportes de las transacciones de una cuenta. Los datos a ser almacenados deben ser: numeroCuenta, titular y saldo. Este reporte debe escribirse en un archivo de texto.

- Utilizar bloques try-with-resources para manejar la escritura en archivos y garantizar que el archivo se cierre correctamente, incluso si ocurre una excepción durante el proceso.

B) USO DE SCANNER PARA LEER DATOS:
- Implementar una funcionalidad para que el sistema pueda leer datos desde un archivo de texto donde se encuentren almacenadas las transacciones de una cuenta, utilizando la clase Scanner.
- Utilizar try-with-resources para garantizar que el Scanner se cierre correctamente, incluso si ocurre una excepción durante la lectura del archivo.

C) GENERACIÓN DE EXCEPCIONES EN REPORTES:
- HistorialVacioException: Se lanza cuando se intenta generar un reporte de una cuenta sin transacciones.
- Manejo de otras excepciones relacionadas con errores al escribir o leer archivos (ej. IOException, FileNotFoundException).

D) PRUEBAS EN ESTA EXPERIENCIA:
- Probar la generación de reportes en cuentas sin transacciones y verificar que se lance la excepción HistorialVacioException.
- Probar la lectura de datos de un archivo inexistente y verificar que se maneje la excepción FileNotFoundException.
- Verificar que, independientemente de si ocurre o no una excepción, los archivos y Scanner se cierran correctamente utilizando try-with-resources.

```java
package Trabajo;

public class Main {
    public static void main(String[] args) {

        CuentaBancaria cuenta1 = new CuentaBancaria(123456, "Juan Pérez",
1000.0);
        CuentaBancaria cuenta2 = new CuentaBancaria(654321, "Ana Gómez",
500.0);

        CuentaCredito cuentaCredito = new CuentaCredito(789012, "Carlos
Fernández", 200.0, 500.0);

        try {
            System.out.println("Saldo inicial de cuenta1: " +
cuenta1.getSaldo());
            cuenta1.depositar(200.0);
            System.out.println("Saldo después de depósito: " +
cuenta1.getSaldo());

            cuenta1.retirar(100.0);
            System.out.println("Saldo después de retiro: " +
cuenta1.getSaldo());

            try {
                cuenta1.retirar(1200.0);
            } catch (SaldoInsuficienteException e) {
                System.out.println("Error: " + e.getMessage());
            }

        } catch (IllegalArgumentException e) {
            System.out.println("Error: " + e.getMessage());
        }

        try {
            cuenta1.transferir(cuenta2, 300.0);
            System.out.println("Saldo de cuenta1 después de transferencia:
" + cuenta1.getSaldo());
            System.out.println("Saldo de cuenta2 después de transferencia:
" + cuenta2.getSaldo());
        } catch (CuentaNoEncontradaException | SaldoInsuficienteException
e) {
            System.out.println("Error: " + e.getMessage());
        }

        try {
            cuenta1.cerrarCuenta();
        } catch (SaldoNoCeroException e) {
            System.out.println("Error: " + e.getMessage());
        }

        try {
            cuentaCredito.retirar(600.0);
        } catch (LimiteCreditoExcedidoException e) {
            System.out.println("Error: " + e.getMessage());
        } catch (SaldoInsuficienteException e) {
             System.out.println("Error: " + e.getMessage());
        }
```

```
        ReporteTransacciones reporte = new ReporteTransacciones();
        try {
            reporte.generarReporte(cuenta1);
            System.out.println("logrado");
        } catch (HistorialVacioException e) {
            System.out.println("Error: " + e.getMessage());
        }

        LectorTransacciones lector = new LectorTransacciones();
        try {
            lector.leerTransacciones("transacciones.txt");
        } catch (Exception e) {
            System.out.println("Error al leer el archivo: " +
e.getMessage());
        }
    }
}
```

```
package Trabajo;
public class CuentaBancaria {
      private  int numeroCuenta;
      private String titular;
      private double saldo;

      public CuentaBancaria(int numeroCuenta, String titular, double saldo)
{
            if (saldo < 0) {
            throw new IllegalArgumentException("El saldo inicial no puede
ser negativo.");
        }
            this.numeroCuenta = numeroCuenta;
            this.titular = titular;
            this.saldo = saldo;
      }

      public void depositar(double monto) {
            if (monto <= 0) {
                  throw new IllegalArgumentException("El monto a
depositar debe ser positivo.");
            }
            saldo += monto;
      }

      public void retirar(double monto) {
            if (monto <= 0) {
                throw new IllegalArgumentException("El monto a retirar
debe ser positivo.");
            }
            if (monto > saldo) {
                throw new SaldoInsuficienteException("Saldo insuficiente
para realizar el retiro.");
            }
            saldo -= monto;
      }
```

```
        public int getNumeroCuenta() {
                return numeroCuenta;
        }

        public String getTitular() {
                return titular;
        }

        public double getSaldo() {
                return saldo;
        }

        public void transferir(CuentaBancaria destino, double monto) {
          if (destino == null) {
              throw new CuentaNoEncontradaException("Cuenta de destino no
encontrada.");
          }
          if (monto > saldo) {
              throw new SaldoInsuficienteException("Saldo insuficiente para
realizar la transferencia.");
          }
          retirar(monto);
          destino.depositar(monto);
      }

        public void cerrarCuenta() {
          if (saldo > 0) {
              throw new SaldoNoCeroException("No se puede cerrar una cuenta
con saldo positivo.");
          }
      }
}
```

```java
package Trabajo;
public class CuentaBancaria {
	private  int numeroCuenta;
	private String titular;
	private double saldo;

	public CuentaBancaria(int numeroCuenta, String titular, double saldo)
{
		if (saldo < 0) {
		throw new IllegalArgumentException("El saldo inicial no puede
ser negativo.");
	}
		this.numeroCuenta = numeroCuenta;
		this.titular = titular;
		this.saldo = saldo;
	}

	public void depositar(double monto) {
		 if (monto <= 0) {
			throw new IllegalArgumentException("El monto a
depositar debe ser positivo.");
		}
		 saldo += monto;
	}

	public void retirar(double monto) {
		 if (monto <= 0) {
			throw new IllegalArgumentException("El monto a retirar
debe ser positivo.");
		}
		if (monto > saldo) {
			throw new SaldoInsuficienteException("Saldo insuficiente
para realizar el retiro.");
		}
		 saldo -= monto;
	}

	public int getNumeroCuenta() {
		return numeroCuenta;
	}

	public String getTitular() {
		return titular;
	}

	public double getSaldo() {
		return saldo;
	}

	public void transferir(CuentaBancaria destino, double monto) {
	  if (destino == null) {
		throw new CuentaNoEncontradaException("Cuenta de destino no
encontrada.");
	}
	  if (monto > saldo) {
		throw new SaldoInsuficienteException("Saldo insuficiente para
realizar la transferencia.");
	}
	  retirar(monto);
	  destino.depositar(monto);
    }
```

```
      public void cerrarCuenta() {
        if (saldo > 0) {
            throw new SaldoNoCeroException("No se puede cerrar una cuenta
con saldo positivo.");
        }
    }
}
```

```
package Trabajo;

public class Excpeciones {
}

  class SaldoInsuficienteException extends RuntimeException {

      public SaldoInsuficienteException(String mensaje) {
         super(mensaje);
     }
 }

 class CuentaNoEncontradaException extends RuntimeException {
     public CuentaNoEncontradaException(String mensaje) {
         super(mensaje);
     }
 }

 class SaldoNoCeroException extends RuntimeException {
     public SaldoNoCeroException(String mensaje) {
         super(mensaje);
     }
 }

  class LimiteCreditoExcedidoException extends RuntimeException {
     public LimiteCreditoExcedidoException(String mensaje) {
         super(mensaje);
     }
 }

  class HistorialVacioException extends RuntimeException {
     public HistorialVacioException(String mensaje) {
         super(mensaje);
     }
 }

  class IllegalArgumentException extends java.lang.IllegalArgumentException
{
     public IllegalArgumentException(String mensaje) {
         super(mensaje);
     }
 }

  class IOException extends java.io.IOException {
     public IOException(String mensaje) {
         super(mensaje);
     }
 }
```

```
class FileNotFoundException extends java.io.FileNotFoundException {
     public FileNotFoundException(String mensaje) {
         super(mensaje);
     }
 }
```

```
package Trabajo;
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class LectorTransacciones {
      public void leerTransacciones(String archivo) {
        try (Scanner scanner = new Scanner(new File(archivo))) {
            while (scanner.hasNextLine()) {
                String linea = scanner.nextLine();
                System.out.println(linea);
            }
        } catch (FileNotFoundException e) {
             System.out.println("Error: " + e.getMessage());
        }
    }
}
```

```
package Trabajo;

import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

public class ReporteTransacciones {
    public void generarReporte(CuentaBancaria cuenta) throws
HistorialVacioException {
        if (cuenta.getSaldo() == 0) {
            throw new HistorialVacioException("No hay transacciones para
reportar.");
        }
        try (FileWriter fileWriter = new FileWriter("reporte.txt");
             PrintWriter printWriter = new PrintWriter(fileWriter)) {
            printWriter.println("Número de Cuenta: " +
cuenta.getNumeroCuenta());
            printWriter.println("Titular: " + cuenta.getTitular());
            printWriter.println("Saldo: " + cuenta.getSaldo());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Ejercicio

Diagrama de clases UML

Explicación del programa

1. Objetivo del Sistema

El sistema de gestión de reservas de hotel está pensado para facilitar la administración de habitaciones, reservas, clientes, y personal de limpieza. Además, ofrece flexibilidad para gestionar diferentes tipos de habitaciones, aplicar políticas de cancelación, y promociones especiales. Es importante destacar que este sistema está diseñado pensando en la escalabilidad, permitiendo que se adapte a las necesidades del hotel a medida que crezca.

El diseño sigue los principios **SOLID**, lo que ayuda a que el código sea fácil de mantener, flexible, y escalable. La idea principal es que cada parte del sistema tiene una responsabilidad específica, lo que facilita realizar cambios sin afectar otras áreas.

2. Gestión de Habitaciones

La clase **Habitación** se encarga de representar cada habitación del hotel. Incluye información básica como el número de habitación, tipo (individual, suite, etc.), y su precio base. Sin embargo, para mantener el código limpio, la lógica más compleja, como la disponibilidad o el cálculo del precio final, se delega a otras clases especializadas.

Atributos principales:

- `numero`: Número de la habitación.
- `tipo`: Tipo de habitación (individual, doble, suite).
- `precioBase`: Precio estándar de la habitación.

Métodos principales:

- `getNumero()`: Devuelve el número de la habitación.
- `getTipo()`: Devuelve el tipo de habitación.
- `getPrecioBase()`: Devuelve el precio base.
- `getPrecioFinal(Date fechaInicio, Date fechaFin)`: Calcula el precio final considerando fechas y promociones.

3. Gestión de Disponibilidad

La clase **GestorDisponibilidadHabitacion** es la encargada de gestionar la disponibilidad de las habitaciones. Aquí se almacena la lista de reservas y promociones, y se verifica si una habitación está disponible en un rango de fechas. Esto sigue el principio de **responsabilidad única**, lo que significa que **Habitación** no se encarga de gestionar su propia disponibilidad, sino que delega esta tarea al **GestorDisponibilidadHabitacion**.

Atributos principales:

- `reservas`: Lista de reservas actuales de la habitación.
- `promociones`: Lista de promociones activas.

Métodos principales:

- `verificarDisponibilidad(Date fechaInicio, Date fechaFin)`: Verifica si la habitación está disponible en las fechas indicadas.
- `calcularPrecio(Habitacion habitacion, Date fechaInicio, Date fechaFin)`: Calcula el precio de la habitación considerando las promociones activas.

4. Promociones

La clase **Promoción** maneja los descuentos que pueden aplicarse a las reservas. Una promoción tiene un porcentaje de descuento y unas fechas específicas de validez. Esto permite que el sistema aplique descuentos automáticamente cuando una reserva entra dentro del rango de fechas de una promoción.

Atributos principales:

- `descuento`: Porcentaje de descuento.
- `fechaInicio` y `fechaFin`: Fechas en las que la promoción es válida.

Métodos principales:

- `getDescuento()`: Devuelve el valor del descuento.
- `esValida(Date fecha)`: Verifica si la promoción es válida en una fecha específica.

5. Reservas

La clase **Reserva** centraliza toda la información sobre una reserva, como la habitación reservada, el cliente que la realizó, las fechas de la estancia, y la política de cancelación aplicable. Esta clase también permite cancelar una reserva, pero delega la lógica de cancelación a un objeto que representa la política de cancelación, lo que sigue el **principio de inversión de dependencias**.

Atributos principales:

- `habitacion`: La habitación reservada.

- `cliente`: El cliente que hizo la reserva.
- `fechaInicio` y `fechaFin`: Fechas de la estancia.
- `politicaCancelacion`: La política de cancelación asociada a la reserva.
- `promocion`: Promoción aplicada (si hay una).

Métodos principales:

- `cancelar()`: Permite cancelar la reserva según la política de cancelación aplicada.

6. Políticas de Cancelación

El sistema admite diferentes políticas de cancelación que se definen mediante la interfaz **PoliticaCancelacion**. Existen varias implementaciones que permiten flexibilidad en cómo se gestionan las cancelaciones. Esto hace que el sistema sea fácil de extender si en el futuro se quieren añadir más políticas.

Implementaciones:

- **PoliticaCancelacionFlexible**: Permite cancelaciones sin penalización hasta 24 horas antes del check-in.
- **PoliticaCancelacionModerada**: Penaliza con un 50% si se cancela hasta 72 horas antes del check-in.
- **PoliticaCancelacionEstricta**: No permite cancelaciones una vez hecha la reserva.

7. Clientes

La clase **Cliente** contiene la información básica del cliente, como su nombre, correo electrónico y teléfono. Cada cliente puede tener múltiples reservas asociadas, y este registro es crucial para poder consultar el historial de reservas y otras operaciones.

Atributos principales:

- `nombre`: Nombre del cliente.
- `email`: Correo electrónico del cliente.
- `telefono`: Número de contacto del cliente.

8. Personal de Limpieza

El personal de limpieza es gestionado a través de la clase **PersonalLimpieza**. Cada empleado puede tener varias habitaciones asignadas para limpiar, y el sistema permite asignar o liberar habitaciones según sea necesario.

Métodos principales:

- `asignarHabitacion(Habitacion habitacion)`: Asigna una habitación al personal de limpieza.
- `removerHabitacion(Habitacion habitacion)`: Libera una habitación asignada.

9. Servicios Especiales de Habitaciones

El sistema también ofrece servicios adicionales como limpieza, comida y lavandería, los cuales se gestionan mediante interfaces especializadas. Esta organización permite que solo las habitaciones que realmente ofrezcan estos servicios los implementen, siguiendo el **principio de segregación de interfaces**.

- **HabitacionEstandar**: Implementa solo el servicio de limpieza.
- **HabitacionSuite**: Implementa limpieza, comida, y lavandería.

10. Notificaciones

El sistema notifica al cliente sobre confirmaciones y cancelaciones de reservas a través de diferentes canales (correo electrónico, SMS o Slack). La clase **NotificadorReserva** sigue el **principio de inversión de dependencias**, ya que utiliza una interfaz para los canales de notificación, permitiendo agregar nuevos canales sin modificar el código existente.

Canales disponibles:

- **EnviadorCorreo**: Notificaciones por correo electrónico.
- **EnviadorSMS**: Notificaciones por SMS.
- **NotificadorSlack**: Notificaciones en Slack.

11. Generación de Informes

La clase **GeneradorInformes** genera informes de ocupación de habitaciones y de ingresos por períodos de tiempo. Estos informes son cruciales para la gestión operativa y estratégica del hotel, permitiendo tomar decisiones basadas en datos.

Métodos principales:

- `generarInformeOcupacion(Date fechaInicio, Date fechaFin)`: Genera un informe sobre la ocupación del hotel en un período de tiempo.
- `generarInformeIngresos(Date fechaInicio, Date fechaFin)`: Genera un informe sobre los ingresos obtenidos en ese período.

12. Controlador de Reservas

Finalmente, el **ControladorReservas** es el punto central donde se integran todas las funciones del sistema. Desde aquí se pueden gestionar las reservas, consultar disponibilidad, cancelar reservas, y coordinar las interacciones entre las diferentes clases del sistema.

Cuestionario

1. **¿Por qué es importante aplicar los principios SOLID en el diseño de software?¿Qué beneficios concretos has observado al aplicarlos en tus proyectos?**

   Los principios SOLID son fundamentales en el diseño de software porque promueven la creación de sistemas más mantenibles, escalables y comprensibles. Estos principios ayudan a evitar problemas comunes como el código espagueti y las dependencias cíclicas, que dificultan la evolución del software [1].

   Beneficios observados:

   - **Mantenibilidad:** El código es más fácil de modificar sin introducir errores, ya que los cambios en un módulo no afectan negativamente a otros módulos.
   - **Reutilización de código:** Al diseñar componentes modulares y con responsabilidades bien definidas, estos pueden reutilizarse en diferentes partes de la aplicación o incluso en diferentes proyectos.
   - **Escalabilidad:** Es más sencillo agregar nuevas funcionalidades al sistema sin necesidad de modificar código existente, gracias a la adherencia al principio de abierto/cerrado.
   - **Comprensión:** El código que sigue los principios SOLID es más intuitivo y fácil de entender para otros desarrolladores, lo que facilita el trabajo en equipo.

2. **¿Qué problemas pueden surgir si no se aplica el Principio de Responsabilidad Única (SRP) en una clase? Proporciona un ejemplo concreto.**

   No aplicar el SRP puede llevar a que una clase tenga múltiples razones para cambiar, lo que incrementa el riesgo de errores y hace que el mantenimiento sea más difícil. Una clase que viola este principio tiende a convertirse en una clase "Dios", con muchas responsabilidades que deberían estar separadas en diferentes clases[2].

   Ejemplo concreto: Imagina una clase ReportGenerator que no solo se encarga de generar un reporte, sino también de formatearlo, guardarlo en disco y enviarlo por correo electrónico. Si se requiere cambiar el formato del reporte, podrías introducir errores en la lógica de envío de correos o en el almacenamiento en disco, porque todas estas responsabilidades están mezcladas en la misma clase.

3. **Analiza un fragmento de código que viola el Principio Abierto/Cerrado (OCP). ¿Cómo podrías refactorizarlo para cumplir con este principio?**

   Fragmento que viola OCP:

```
class Invoice {
    public double calculateTotalAmount(String type) {
        if (type.equals("standard")) {
            return standardAmountCalculation();
        } else if (type.equals("discounted")) {
            return discountedAmountCalculation();
        } else {
            return defaultAmountCalculation();
        }
    }
}
```

Este código viola el principio OCP porque para agregar un nuevo tipo de cálculo, es necesario modificar la clase Invoice, lo cual es contrario a estar "cerrado para modificaciones".

Refactorización:

```
interface AmountCalculator {
    double calculate();
}

class StandardAmountCalculator implements AmountCalculator {
    public double calculate() {
        return standardAmountCalculation();
    }
}

class DiscountedAmountCalculator implements AmountCalculator {
    public double calculate() {
        return discountedAmountCalculation();
    }
}

class Invoice {
    private AmountCalculator calculator;

    public Invoice(AmountCalculator calculator) {
        this.calculator = calculator;
    }

    public double calculateTotalAmount() {
        return calculator.calculate();
    }
}
```

Aquí, la clase Invoice está abierta para extensiones (puedes agregar nuevas implementaciones de AmountCalculator) pero cerrada para modificaciones (no necesitas cambiar Invoice para agregar nuevas estrategias de cálculo).

4. **¿Consideras que el Principio de Sustitución de Liskov (LSP) es siempre aplicable en todas las situaciones de herencia? Justifica tu respuesta.**

El principio de Sustitución de Liskov no siempre es aplicable en todas las situaciones de herencia, especialmente cuando la herencia se utiliza de manera inapropiada. Por ejemplo, si una subclase no puede reemplazar a su superclase sin alterar el comportamiento esperado del sistema, entonces no cumple con LSP[1].

Justificación: LSP es aplicable cuando la relación entre la superclase y la subclase es de tipo "es-un" genuino. Sin embargo, en algunos casos, la herencia se utiliza incorrectamente para compartir código, lo que lleva a violaciones de LSP. En tales casos, podría ser más adecuado utilizar composición en lugar de herencia.

5. **Describe una situación en la que hayas aplicado el Principio de Segregación de Interfaces (ISP) para mejorar el diseño de una aplicación. ¿Qué beneficios obtuviste?**

   En un proyecto, teníamos una interfaz UserOperations que incluía métodos para agregar, actualizar, eliminar y listar usuarios. Algunas clases solo necesitaban el método de listado, pero estaban forzadas a implementar todos los métodos[1].

   Aplicación del ISP: Separamos UserOperations en varias interfaces más pequeñas, como UserReader y UserWriter. De esta manera, las clases solo implementaban las interfaces que realmente necesitaban.

   Beneficios obtenidos:

   - **Menor acoplamiento:** Las clases no estaban obligadas a implementar métodos que no usaban.
   - **Mayor claridad:** Cada interfaz tenía un propósito claro y específico, lo que hacía el código más fácil de entender y mantener.

6. **¿Qué desafíos puedes enfrentar al aplicar el Principio de Inversión de Dependencias (DIP) en un proyecto grande y complejo?**

   - **Complejidad inicial:** Implementar DIP requiere una planificación cuidadosa y puede aumentar la complejidad inicial del diseño del sistema.

   - **Necesidad de abstracciones:** Crear interfaces y clases abstractas para aplicar DIP puede resultar en una proliferación de clases que, si no se gestionan adecuadamente, pueden complicar el mantenimiento.

   - **Adaptación del equipo:** No todos los desarrolladores pueden estar familiarizados con los conceptos de DIP, lo que puede requerir formación adicional y un cambio de mentalidad[1].

7. **Compara y contrasta las ventajas y desventajas de utilizar archivos planos versus bases de datos SQLite para almacenar datos en una aplicación Java.**

   Ventajas de archivos planos:

   - Simplicidad: Los archivos planos son fáciles de crear y manipular, ideales para aplicaciones pequeñas o para almacenar configuraciones simples.
   - Portabilidad: No requieren la instalación de un sistema de gestión de bases de datos (SGBD), lo que simplifica su uso en entornos con recursos limitados.

   Desventajas:

   - Escalabilidad limitada: Manejar grandes volúmenes de datos con archivos planos puede volverse complejo y propenso a errores.
   - Falta de estructura: Los archivos planos no tienen una estructura rígida, lo que puede llevar a inconsistencias en los datos.

Ventajas de SQLite:

- Estructura: SQLite proporciona un SGBD ligero con soporte para SQL, lo que facilita la gestión de datos estructurados.
- Fiabilidad: Ofrece transacciones ACID, asegurando la integridad de los datos incluso en casos de fallos.

Desventajas:

- Sobrecarga: Aunque es ligero, SQLite es más complejo que los archivos planos y requiere un poco más de recursos.
- Configuración inicial: Aunque mínima, existe una curva de aprendizaje inicial para configurar y utilizar SQLite en comparación con archivos planos.

8. ¿En qué situaciones recomendarías utilizar SQLite en lugar de un sistema de gestión de bases de datos más robusto como MySQL o PostgreSQL?

- **Aplicaciones de escritorio o móviles pequeñas:** SQLite es ideal para aplicaciones que requieren un almacenamiento de datos ligero, como aplicaciones móviles o software de escritorio con una base de datos integrada.

- **Prototipos o desarrollo rápido:** Para proyectos en etapas iniciales o prototipos donde se necesita un SGBD simple sin la complejidad de configurar MySQL o PostgreSQL.

- **Entornos con recursos limitados:** SQLite es preferible cuando no se dispone de un servidor dedicado o los recursos son limitados.

9. ¿Qué dificultades has encontrado al trabajar con archivos en Java? ¿Cómo las has superado?

Dificultades comunes:

- **Manejo de excepciones:** Trabajar con archivos requiere manejar diversas excepciones (como IOException) que pueden ocurrir durante la lectura/escritura de archivos.
- **Encoding y decodificación de datos:** Asegurar que los datos sean leídos y escritos con el encoding correcto puede ser complicado.
- **Sincronización de acceso:** En aplicaciones concurrentes, manejar el acceso a archivos desde múltiples hilos puede llevar a problemas de sincronización.

Superación de dificultades:

- **Uso de try-with-resources:** Este constructo facilita la gestión de recursos y asegura que los archivos sean cerrados correctamente.
- **Utilización de Charset:** Al especificar explícitamente el charset al leer o escribir archivos, se evita la pérdida o corrupción de datos.
- **Bloqueo de archivos:** Implementar mecanismos de bloqueo (como FileLock) para controlar el acceso concurrente a archivos.

10. ¿Es la serialización de objetos siempre la mejor opción para almacenar datos en archivos? ¿Qué alternativas existen y cuándo serían más adecuadas?

Serialización de objetos:

- **Ventajas:** Es útil cuando se necesita almacenar el estado completo de un objeto, incluyendo sus referencias a otros objetos. Es simple y automática, sin necesidad de escribir mucho código adicional.
- **Desventajas:** Los archivos serializados no son fácilmente legibles o editables por humanos, lo que los hace menos adecuados para almacenamiento de datos que requieren ser revisados o modificados manualmente.

Alternativas:

- **Archivos de texto o JSON:** Ideales cuando los datos necesitan ser humanamente legibles o cuando se integran con otros sistemas que utilizan JSON.
- **Bases de datos:** Adecuadas para datos estructurados que requieren consultas complejas o relaciones entre diferentes entidades.
- **XML:** Útil para datos estructurados que deben ser compartidos entre diferentes plataformas o sistemas.

11. Examina un fragmento de código que utiliza JDBC para interactuar con una base de datos SQLite.¿Identificas alguna posible mejora en términos de seguridad o rendimiento?

Código ejemplo:

```
String query = "SELECT * FROM users WHERE username = '" + username + "'";
Statement statement = connection.createStatement();
ResultSet resultSet = statement.executeQuery(query);
```

Posibles mejoras:

- **Seguridad:** El código es vulnerable a ataques de inyección SQL. Se debe usar PreparedStatement en lugar de Statement para evitar la inyección SQL.
- **Rendimiento:** El uso de PreparedStatement no solo mejora la seguridad, sino que también puede mejorar el rendimiento, ya que permite que las consultas sean precompiladas por el motor de la base de datos.

Código refactorizado:

```
String query = "SELECT * FROM users WHERE username = ?";
PreparedStatement preparedStatement = connection.prepareStatement(query);
preparedStatement.setString(1, username);
ResultSet resultSet = preparedStatement.executeQuery();
```

12. ¿Consideras que JDBC es una API adecuada para trabajar con bases de datos en Java? ¿Qué ventajas y desventajas tiene en comparación con otras tecnologías como Hibernate o JPA?

Ventajas de JDBC:

- **Control detallado:** JDBC permite un control muy preciso sobre las operaciones SQL y la gestión de conexiones.
- **Ligereza:** Es más ligero y directo que frameworks ORM como Hibernate, lo que lo hace adecuado para aplicaciones con necesidades específicas o cuando se requiere un alto rendimiento.

Desventajas de JDBC:

- **Código repetitivo:** JDBC puede llevar a la duplicación de código y a un manejo manual detallado de las conexiones y las transacciones.
- **Menos abstracto:** No proporciona las abstracciones y facilidades que ofrecen frameworks como Hibernate o JPA, que automatizan muchas tareas comunes y mejoran la productividad.

Comparación con Hibernate/JPA:

- **Hibernate/JPA:** Simplifican el desarrollo mediante mapeo objeto-relacional (ORM), lo que reduce la cantidad de código necesario para manejar operaciones de base de datos, pero pueden introducir una sobrecarga adicional en el rendimiento.

13. ¿Qué has aprendido sobre la importancia de manejar las excepciones SQLException al trabajar con JDBC?

Importancia del manejo de excepciones:

- **Robustez:** Un manejo adecuado de SQLException es crucial para garantizar que los errores en la interacción con la base de datos no provoquen fallos no controlados en la aplicación.
- **Registro y depuración:** Capturar y registrar excepciones permite identificar y corregir problemas en las operaciones con la base de datos.
- **Transacciones:** El manejo adecuado de excepciones asegura que las transacciones se manejen correctamente, evitando inconsistencias en la base de datos.

Mejores prácticas:

- Siempre usar bloques try-catch alrededor de las operaciones JDBC.
- Asegurarse de cerrar ResultSet, Statement, y Connection en un bloque finally o usar try-with-resources.

14. ¿Cómo se pueden prevenir los ataques de inyección SQL al utilizar JDBC? ¿Qué prácticas de seguridad son esenciales?

Prevención de inyección SQL:

- **Uso de PreparedStatement:** Evitar concatenar cadenas para construir consultas SQL. En su lugar, utilizar PreparedStatement, que permite separar los datos de la consulta.
- **Validación de entrada:** Validar y sanitizar todas las entradas de usuario antes de utilizarlas en consultas SQL.
- **Uso de ORM:** Considerar el uso de frameworks ORM como Hibernate que abstractan el manejo de SQL y reducen el riesgo de inyección.

Prácticas de seguridad esenciales:

- Siempre validar y sanitizar las entradas del usuario.
- Limitar los permisos de la base de datos según el principio de menor privilegio, asegurando que la aplicación solo tenga los permisos necesarios para realizar sus funciones.

15. Describe los pasos necesarios para crear una conexión a una base de datos SQLite desde una aplicación Java utilizando JDBC.

1) **Carga del controlador JDBC:**

```
Class.forName("org.sqlite.JDBC");
```

2) **Establecimiento de la conexión:**

```
Connection connection =
DriverManager.getConnection("jdbc:sqlite:sample.db");
```

3) **Creación de una declaración SQL (`Statement` o `PreparedStatement`):**

```
Statement statement = connection.createStatement();
```

4) **Ejecución de la consulta:**

```
ResultSet resultSet = statement.executeQuery("SELECT * FROM users");
```

5) **Procesamiento de los resultados:**

```
while (resultSet.next()) {
System.out.println(resultSet.getString("username"));
}
```

6) **Cierre de recursos:**

```
resultSet.close();
statement.close();
connection.close();
```

16. **¿Qué criterios utilizarías para elegir entre diferentes controladores JDBC para SQLite? ¿Qué factores son importantes a considerar?**

    - **Compatibilidad:** Asegurarse de que el controlador sea compatible con la versión de Java y de SQLite que se está utilizando.
    - **Rendimiento:** Evaluar el rendimiento del controlador, especialmente si la aplicación requiere operaciones intensivas de base de datos.
    - **Características soportadas:** Verificar que el controlador soporte todas las características necesarias para la aplicación, como transacciones, seguridad, etc.
    - **Soporte y comunidad:** Preferir controladores que tengan un buen soporte, documentación y una comunidad activa que pueda ayudar en caso de problemas.

17. **¿Qué beneficios has encontrado al utilizar el patrón MVC en el desarrollo de aplicaciones Java?¿Qué desafíos has enfrentado?**

    Beneficios del patrón MVC:

    - **Separación de responsabilidades:** MVC separa claramente la lógica de negocio (Modelo), la interfaz de usuario (Vista) y el control de flujo (Controlador), lo que facilita la mantenibilidad y escalabilidad.
    - **Facilita pruebas unitarias:** Al separar la lógica de negocio del código de la interfaz de usuario, es más fácil probar cada componente de manera independiente.
    - **Reutilización de código:** Los modelos y controladores pueden ser reutilizados en diferentes vistas, promoviendo la reutilización del código.

    Desafíos:

    - **Complejidad inicial:** Configurar el patrón MVC puede ser complejo, especialmente para desarrolladores principiantes.
    - **Comunicación entre componentes:** Asegurar una correcta comunicación entre el modelo, la vista y el controlador puede ser complicado y propenso a errores.

18. **¿Es el patrón MVC la mejor opción para todos los tipos de aplicaciones Java? ¿En qué situaciones podría ser menos adecuado?**

    - **Aplicaciones simples:** Para aplicaciones muy pequeñas o con una lógica de negocio mínima, el overhead de implementar MVC puede no estar justificado.

    - **Aplicaciones con lógica muy distribuida:** En aplicaciones donde la lógica de negocio está muy distribuida o es muy específica de la interfaz, otros patrones como MVP (Model-View-Presenter) podrían ser más apropiados.

19. **¿Cómo se relacionan los principios SOLID con el patrón MVC? ¿Cómo puedes aplicar SOLID al diseñar los componentes de una aplicación MVC?**

    - **S**: El modelo, la vista y el controlador tienen responsabilidades únicas, respetando el SRP.

    - **O**: Los controladores y vistas pueden extenderse con nuevas funcionalidades sin modificar su código existente, cumpliendo con el OCP.

- **L**: Las clases derivadas de los modelos pueden sustituir a sus clases base sin romper la funcionalidad, respetando el LSP.
- **I**: Cada interfaz que se implementa en MVC puede estar diseñada para ser lo más específica posible, cumpliendo con el ISP.
- **D**: Los controladores dependen de abstracciones (interfaces de modelos), no de implementaciones concretas, cumpliendo con el DIP.

20. Evalúa tu propio proyecto final en términos de su adherencia a los principios SOLID y al patrón MVC. ¿Qué aspectos podrías mejorar en el futuro? ¿Pregunta 3?

Para evaluar tu proyecto final, podrías considerar cómo se han aplicado los principios SOLID en el diseño de clases y si has seguido la separación de responsabilidades propuesta por el patrón MVC. Podrías identificar áreas donde se han violado principios como SRP o OCP, y planificar refactorizaciones para mejorar la mantenibilidad y la escalabilidad del código.

Referencias

[1] C. S. Horstmann, *Big Java : Early Objects*. Hoboken, Nj: Johnwiley and Sons, 2019.

[2] H. Deitel y P. J. Deitel, *Java How to Program, Late Objects, Global Edition*, 11th ed. Pearson, 2019

Anexos

Los códigos fuente de las actividades y ejercicios están en la carpeta Trabajo_3:
https://github.com/JoaquinLoaizaUCSM/Trabajos_LP_3

Video explicando el codigo del ejercicio

Explicasion_codigo