# Excepciones

Primero tienes que crear la clase de la excepcion que sea hija de Exception, y en el constructor le pasas el string al padre:

public class DificultadExcepcion extends Exception{

    public DificultadExcepcion(String mensaje){

        super(mensaje);

    }

}

Luego haces otra clase que sirve para emplear la logica de la excepcion que has creado

public class VerificarDificultadExcepcion{

    public static void verificandoDificultadExcepcion(String dificultad) throws DificultadExcepcion{

        boolean respuesta=true;

        if (dificultad.equalsIgnoreCase("verde") || dificultad.equalsIgnoreCase("azul")

                || dificultad.equalsIgnoreCase("roja"))

            respuesta=false;

        if (respuesta)

            throw new DificultadExcepcion("La dificultad introducida es incorrecta");

    }

}

Listo, ahora ya puedes llamar al metodo que la comprueba y tratarla con try catch las veces que quieras

# Manejo de archivos

## ESCRIBIR

### Para archivos no Binarios:

public boolean escribirFichero() {

        boolean respuesta = true;

        try (PrintWriter writer = new PrintWriter(new FileWriter("FicheroTXT"))) {

            for (Pista pista : pistas) {

                writer.println(pista.toString());

            }

        } catch (IOException e) {

            respuesta = false;

            System.out.println("Error al guardar el usuario en el archivo: "

                    + e.getMessage());

        }

        return respuesta;

    }

ES IMPORTANTE HACER LAS CLASES SERIALIZABLES.

### Para archivos Binarios

public boolean escribirFicheroBinario() {

        try {

            FileOutputStream archivoBinario = new FileOutputStream("FicheroBIN");

            ObjectOutputStream escribir = new ObjectOutputStream(archivoBinario);

            escribir.writeObject(pistas);//un arraylist

            archivoBinario.close();

            escribir.close();

            System.out.println("Todo guardado.");

            return true;

        } catch (IOException e) {

            System.out.println("Ha habido un error al guardar el archivo

binario.");

            return false;

        }

    }

## LEER

### Para archivos no Binarios:

Partes de la base que en el archivo está separado por un ; cada valor

public void cargarFichero() {

        Scanner scanner = null;

        pistas = new ArrayList<>();

        try {

            scanner = new Scanner(new File("FicheroTXT"));

            while (scanner.hasNextLine()) {

                String[] resultado = scanner.nextLine().split(";");

                if (resultado[0].equalsIgnoreCase("skialpino")) {

                    int id = Integer.parseInt(resultado[1]),km =

Integer.parseInt(resultado[2]);

                    String provincia = resultado[3],nombre =resultado[4],

dificultad = resultado[5];

                    pistas.add(new SkipAlpino(id, km, provincia, nombre,

dificultad));

                } else if(resultado[0].equalsIgnoreCase("skifondo")){

      // Extrae la parte correspondiente a la lista de string

                   int pueblosStartIndex = resultado[5].indexOf("[") + 1;

                  int pueblosEndIndex = resultado[5].lastIndexOf("]")-1;

                   String pueblosText = resultado[5].substring(

pueblosStartIndex,pueblosEndIndex);

                 pueblosText = pueblosText.trim();// Elimina los espacios

                 List<String> pueblos = new ArrayList<String>();

                 String[] pueblosarray= pueblosText.split(",");

                 for (String pueblo : pueblosarray) {

                     pueblos.add(pueblo);

                 }

                 pistas.add(new SkiFondo(Integer.parseInt(resultado[1]),

Integer.parseInt(resultado[2]),

resultado[3], resultado[4], pueblos));

                }

            }

        } catch (FileNotFoundException e) {

            System.out.println("Archivo no encontrado");

        } catch (DificultadExcepcion e) {

            System.out.println("la dificultad no existe");

        }

    }

### Para archivos Binarios:

public void cargarFicheroBinario() {

        try {

            // Abrir el archivo para leer

            FileInputStream archivoEntrada = new

FileInputStream("archivo.bin");

            ObjectInputStream lectorObjeto = new

ObjectInputStream(archivoEntrada);

            // Leer el objeto del archivo

            pistas = (ArrayList<Pista>) lectorObjeto.readObject();

            // Cerrar el archivo de lectura

            lectorObjeto.close();

        } catch (IOException e) {

            System.out.println("Ha habido un error al cargar el archivo

binario: " + e.getMessage());

        } catch (ClassNotFoundException e) {

            System.out.println("Error al leer el objeto del archivo

binario: " + e.getMessage());

        }

    }

# JSON

## Escribiendo

### Parte lógica:

public void escribirJSON() {

        Pista pista = new Pista(0, 0, null, null);

        String json = pista.toJson();

        try (FileWriter writer = new FileWriter("data.json",true)) {

            writer.write(json);

        } catch (IOException e) {

            e.printStackTrace();

        }

    }

### En el objeto:

public String toJson() {

        Gson gson = new GsonBuilder().create();

        return gson.toJson(this);

    }

## Leyendo

import java.lang.reflect.Type;

public void cargarJSON() {

        BufferedReader bufferedReader = null;

        pistas = new ArrayList<>();

        try {

            bufferedReader = new BufferedReader(new

FileReader("FicheroJSON.json"));

            Gson gson = new Gson();

            Type listType = TypeToken.getParameterized(List.class,

Pista.class).getType();

            List<Pista> pistaList = gson.fromJson(bufferedReader,

listType);

            pistas.addAll(pistaList);

            bufferedReader.close();

        } catch (FileNotFoundException e) {

            System.out.println("Archivo no encontrado");

        } catch (Exception e) {

            e.printStackTrace();

        }

    }

# REGEX (Expresiones regulares)

* find(): Busca la siguiente coincidencia de la expresión regular en la cadena de texto. Retorna true si se encuentra una coincidencia, y luego puedes utilizar otros métodos como start(), end() y group() para obtener información sobre la coincidencia encontrada. Si no se encuentra ninguna coincidencia, retorna false.
* matches(): Intenta encontrar una coincidencia completa de la expresión regular en toda la cadena de texto. Es decir, verifica si toda la cadena coincide con el patrón completo. Retorna true si hay una coincidencia completa, de lo contrario, retorna false.
* start(): Retorna la posición de inicio de la última coincidencia encontrada.
* end(): Retorna la posición de finalización de la última coincidencia encontrada.
* group(): Retorna la subcadena que coincide con el patrón en la última coincidencia encontrada.
* replaceAll(): Reemplaza todas las coincidencias de la expresión regular en la cadena de texto por una nueva subcadena.
* replaceFirst(): Reemplaza la primera coincidencia de la expresión regular en la cadena de texto por una nueva subcadena.
* reset(): Restablece el estado del Matcher para volver a buscar coincidencias en la cadena de texto.

Se hacen así:

public static void main(String[] args) {

            String tel="+34 692992292";

            String patron = "+34 [0-9]{9}";

            Pattern pattern = Pattern.compile(patron);

            Matcher matcher = pattern.matcher(texto);

        }

# Stream

En Java, un Stream es una secuencia de elementos que se puede procesar de manera secuencial o paralela. Proporciona una forma declarativa y funcional de realizar operaciones en colecciones de datos, como listas o arreglos. Los Streams permiten escribir código más conciso y legible, evitando bucles explícitos y condicionales.

#### Creación de un Stream:

1. A partir de una colección: Puedes crear un Stream a partir de una colección existente, como una lista o un conjunto, utilizando el método stream() o parallelStream().
2. A partir de un arreglo: Puedes crear un Stream a partir de un arreglo utilizando el método Arrays.stream().
3. Utilizando métodos generadores: También puedes crear Streams utilizando métodos generadores, como Stream.of() o Stream.iterate().

#### Operaciones intermedias:

Después de crear un Stream, puedes realizar operaciones intermedias en él. Estas operaciones no producen un resultado final, pero transforman o filtran los elementos del Stream. Algunas operaciones intermedias comunes son:

1. filter(Predicate): Filtra los elementos del Stream según un predicado.
2. map(Function): Transforma cada elemento del Stream mediante una función.
3. distinct(): Elimina los elementos duplicados en el Stream.
4. sorted(): Ordena los elementos del Stream.
5. limit(long): Limita el número de elementos del Stream.
6. skip(long): Salta los primeros elementos del Stream.

#### Operaciones terminales

Después de aplicar las operaciones intermedias, debes realizar una operación terminal para obtener un resultado final. Algunas operaciones terminales comunes son:

• forEach(Consumer): Ejecuta una acción en cada elemento del Stream. Por ejemplo:

    List<String> nombres = Arrays.asList( "Juan" , "Pedro" , "Ana" );

    nombres.stream().forEach(System.out::println);

• count(): Cuenta el número de elementos del Stream. Por ejemplo:

    List<Integer> numeros = Arrays.asList( 1 , 2 , 3 , 4 , 5 ); long

cantidad = numeros.stream().count();

• anyMatch(Predicate): Comprueba si algún elemento cumple con

un predicado. Por ejemplo:

    List<Integer> numeros = Arrays.asList( 1 , 2 , 3 , 4 , 5 ); boolean

hayPar = numeros.stream().anyMatch(n -> n % 2 == 0 );

• allMatch(Predicate): Comprueba si todos los elementos

cumplen con un predicado. Por ejemplo:

    List<Integer> numeros = Arrays.asList( 2 , 4 , 6 , 8 , 10 ); boolean

sonPares = numeros.stream().allMatch(n -> n % 2 == 0 );

•noneMatch(Predicate): Comprueba si ningún elemento cumple

con un predicado. Por ejemplo:

    List<Integer> numeros = Arrays.asList( 1 , 3 , 5 , 7 , 9 ); boolean sonPares = numeros.stream().noneMatch(n -> n % 2 == 0 );

• collect(Collector): Recolecta los elementos del Stream en una

colección o valor. Por ejemplo, puedes recolectar los números pares en una lista:

    List<Integer> numeros = Arrays.asList( 1 , 2 , 3 , 4 , 5 ); List<Integer> pares = numeros.stream().filter(n -> n % 2 == 0 ).collect(Collectors.toList());

# LIST SET Y MAP

## List

Una List es una colección ordenada de elementos en la que se permite la duplicación de valores. Puedes acceder a los elementos de una List por su índice, lo que significa que los elementos están ordenados según el orden en el que se agregaron. En Java, la interfaz más

común para las Listas es java.util.List. Algunas implementaciones conocidas de la interfaz List son ArrayList y LinkedList:

import java.util.ArrayList; import java.util.List; public class ListExample {

public static void main(String[] args) {

// Crear una List de enteros

List<Integer> numeros = new ArrayList<>();

// Agregar elementos a la List numeros.add(10); numeros.add(20); numeros.add(30);

// Acceder a elementos por índice int primerElemento = numeros.get(0);

System.out.println("Primer elemento: " + primerElemento);

// Recorrer la List

for (int numero : numeros) { System.out.println(numero);

}

// Eliminar un elemento numeros.remove(1);

System.out.println("Después de eliminar un elemento: " + numeros);

}

}

## Sets:

Un Set es una colección que no permite elementos duplicados. No hay un orden definido en un Set, lo que significa que no puedes acceder a los elementos por su posición. En Java, la interfaz más común para los Sets es java.util.Set. Algunas implementaciones conocidas son HashSet y TreeSet.

import java.util.HashSet; import java.util.Set; public class SetExample {

public static void main(String[] args) {

// Crear un Set de cadenas Set<String> nombres = new HashSet<>();

// Agregar elementos al Set nombres.add("Juan"); nombres.add("María"); nombres.add("Pedro");

nombres.add("María"); // No se permite duplicados

// Recorrer el Set

for (String nombre : nombres) { System.out.println(nombre);

}

// Verificar si un elemento existe en el Set boolean contienePedro = nombres.contains("Pedro");

System.out.println("Contiene Pedro: " + contienePedro);

// Eliminar un elemento nombres.remove("Juan");

System.out.println("Después de eliminar a Juan: " + nombres);

}

}

## Map

Un Map es una estructura de datos que asocia claves (keys) con valores (values). Cada clave es única y se utiliza para acceder a su correspondiente valor. En Java, la interfaz más común para los Maps es java.util.Map. Algunas implementaciones conocidas son HashMap y TreeMap. import java.util.HashMap;

import java.util.Map; public class MapExample {

public static void main(String[] args) {

// Crear un Map de cadenas a enteros Map<String, Integer> notas = new HashMap<>();

// Agregar elementos al Map notas.put("Juan", 90);

notas.put("María", 85);

notas.put("Pedro", 95);

// Acceder a un valor por clave

int notaDeJuan = notas.get("Juan"); System.out.println("Nota de Juan: " + notaDeJuan);

// Recorrer el Map

for (Map.Entry<String, Integer> entry : notas.entrySet()) { String nombre = entry.getKey();

int nota = entry.getValue(); System.out.println(nombre + ": " + nota);

}

// Verificar si una clave existe en el Map

boolean contieneMaria = notas.containsKey("María"); System.out.println("Contiene a María: " + contieneMaria);

// Eliminar una entrada notas.remove("Pedro");

System.out.println("Después de eliminar a Pedro: " + notas);

}

}

## Importante

La diferencia principal entre HashSet/HashMap y TreeSet/TreeMap radica en la forma en que se almacenan y ordenan los elementos.

HashSet / HashMap:

HashSet es una implementación de la interfaz Set que utiliza una tabla hash para almacenar los elementos. Esto significa que no hay un orden específico en los elementos almacenados.

HashMap es una implementación de la interfaz Map que también utiliza una tabla hash, pero almacena pares de clave-valor. Tampoco hay un orden específico en los pares clave-valor almacenados.

La ventaja principal de HashSet y HashMap es su eficiencia en términos de búsqueda, inserción y eliminación de elementos. Sin embargo, no garantizan ningún orden particular de los elementos almacenados.

## TreeSet / TreeMap:

TreeSet es una implementación de la interfaz Set que mantiene los elementos ordenados en orden ascendente o según un comparador personalizado. Utiliza una estructura de árbol (generalmente un árbol rojo-negro) para almacenar y organizar los elementos.

TreeMap es una implementación de la interfaz Map que también mantiene los pares clave-valor ordenados según la clave.

La ventaja principal de TreeSet y TreeMap es que los elementos se almacenan en orden, lo que facilita la iteración en un orden específico. Sin embargo, esta ventaja conlleva una pequeña pérdida de eficiencia en comparación con HashSet y HashMap.

En resumen, mientras que HashSet/HashMap se enfocan en la eficiencia y no garantizan un orden específico, TreeSet/TreeMap se centran en mantener los elementos ordenados, lo que puede ser útil en situaciones donde el orden es importante. La elección entre ellos dependerá de tus necesidades

específicas en términos de eficiencia y ordenamiento.

# For each

El bucle "for-each" (también conocido como "enhanced for loop" en Java) es una forma conveniente de recorrer elementos en una colección o matriz (array) en Java. A diferencia de

otros tipos de bucles, como el "for" tradicional, el bucle "for-each" no utiliza un contador o un índice explícito para iterar sobre los elementos. En su lugar, se encarga automáticamente de recorrer cada elemento en la colección. Veamos cómo se utiliza el bucle "for-each" en Java. for (tipoDeElemento elemento : colección) {

// Cuerpo del bucle

// Se ejecuta para cada elemento en la colección

}

Aquí tienes una explicación paso a paso de cómo funciona:

* tipoDeElemento: Esto representa el tipo de datos de cada elemento en la colección. Por ejemplo, si estás recorriendo una matriz de enteros, tipoDeElemento sería int. Si estás recorriendo una lista de cadenas, tipoDeElemento sería String.
* elemento: Esto es una variable que se utilizará para representar cada elemento en la colección durante cada iteración del bucle. Puedes darle cualquier nombre válido que desees.
* colección: Esto representa la colección o matriz sobre la cual deseas iterar.

Durante cada iteración del bucle, la variable elemento tomará el valor del siguiente elemento en la colección. El bucle se repetirá automáticamente hasta que se hayan recorrido todos los elementos en la colección.

Aquí tienes un ejemplo práctico que muestra cómo se utiliza el bucle "for-each" para recorrer una matriz de enteros:

int[] numeros = {1, 2, 3, 4, 5}; for (int numero : numeros) {

System.out.println(numero);

}

En este ejemplo, la variable numero toma el valor de cada elemento en la matriz numeros durante cada iteración del bucle. El bucle imprime cada número en una línea separada.

El bucle "for-each" también se puede utilizar con otras colecciones en Java, como listas, conjuntos o mapas. Aquí tienes un ejemplo de cómo recorrer una lista de cadenas: List<String> palabras = new ArrayList<>();

palabras.add("Hola"); palabras.add("Mundo");

for (String palabra : palabras) { System.out.println(palabra);

}

En este caso, la variable palabra tomará el valor de cada elemento en la lista palabras. El bucle imprime cada palabra en una línea separada.

Recuerda que el bucle "for-each" solo permite iterar sobre los elementos de la colección en orden, de principio a fin. No proporciona acceso a un índice o posición específica en la

colección.

# Lambda

En programación, una expresión lambda es una forma concisa de representar una función anónima, es decir, una función que no tiene un nombre asociado. Las expresiones lambda son utilizadas para implementar el paradigma de programación funcional en Java, permitiendo

tratar las funciones como objetos de primera clase. Esto significa que puedes pasar una función como argumento, devolverla como resultado y almacenarla en una variable.

La sintaxis básica de una expresión lambda en Java es la siguiente:

(parametros) -> { cuerpo de la función }

Donde "parámetros" representa los parámetros de la función (si los hay) y "cuerpo de la

función" contiene las instrucciones que se ejecutarán al llamar a la función. A continuación, te mostraré algunos ejemplos para ilustrar cómo funcionan las expresiones lambda en diferentes contextos.

1. Expresiones lambda sin parámetros:

() -> {

// Código de la función

}

Este ejemplo representa una función sin parámetros. Puedes colocar el código que desees dentro del cuerpo de la función.

2. Expresiones lambda con un parámetro:

(parametro) -> {

// Código de la función que utiliza el parámetro

}

Aquí, "parámetro" representa el único parámetro de la función. Puedes acceder y utilizar este parámetro dentro del cuerpo de la función.

3. Expresiones lambda con múltiples parámetros:

(parametro1, parametro2) -> {

// Código de la función que utiliza los parámetros

}

En este caso, "parametro1" y "parametro2" son los parámetros de la función. Puedes utilizar ambos parámetros en el cuerpo de la función.

Una vez que hayas definido una expresión lambda, puedes utilizarla en diferentes contextos, como en métodos de orden superior, interfaces funcionales y colecciones.

#### Métodos de orden superior:

Los métodos de orden superior son aquellos que toman una función como argumento o devuelven una función. Puedes utilizar expresiones lambda para pasar una función como argumento a un método. Por ejemplo, supongamos que tienes un método filtrar que toma una lista y una función de filtro, y devuelve una nueva lista con los elementos que cumplen el criterio de filtro. Puedes utilizar una expresión lambda para definir el

filtro en el lugar donde llamas al método filtrar.

List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10); List<Integer> numerosPares = filtrar(numeros, (numero) -> numero % 2

== 0);

En este ejemplo, (numero) -> numero % 2 == 0 es una expresión lambda que representa una función que toma un número y devuelve true si es par.

#### Interfaces funcionales:

Java proporciona interfaces funcionales predefinidas que se pueden utilizar como tipos para expresiones lambda. Una interfaz funcional es una interfaz que contiene exactamente un método abstracto. Puedes utilizar una expresión lambda para

implementar el método abstracto de una interfaz funcional. Por ejemplo, la interfaz funcional Runnable se utiliza para representar una tarea que se ejecutará en un hilo separado. Tiene un único método abstracto llamado run(). Puedes utilizar una expresión lambda para implementar este método.

Runnable runnable = () -> {

// Código de la tarea que se ejecutará en un hilo separado

};

Aquí, la expresión lambda () -> { // Código de la tarea } se utiliza para implementar el método run() de la interfaz Runnable.

Además de las interfaces funcionales predefinidas, también puedes crear tus propias

interfaces funcionales. Una interfaz funcional personalizada es aquella que defines con un único método abstracto y puedes utilizar expresiones lambda para implementar ese método en diferentes contextos.

#### Colecciones:

Las expresiones lambda también son útiles para operaciones en colecciones, como

filtrar, mapear, reducir, entre otras. Estas operaciones se pueden realizar utilizando el nuevo API de Streams introducido en Java 8. Por ejemplo, supongamos que tienes una lista de números y quieres filtrar solo los números pares y luego imprimirlos. Puedes usar una expresión lambda con el método filter() y forEach() de la clase Stream para

lograrlo.

List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10); numeros.stream()

.filter(numero -> numero % 2 == 0)

.forEach(System.out::println);

Aquí, la expresión lambda numero -> numero % 2 == 0 se utiliza como el predicado de filtro para seleccionar los números pares. Después de aplicar el filtro, se utiliza el método forEach() para imprimir cada número par.

# Comparador

Si aux es menor que 0, significa que arg0 es menor que arg1. Si aux es mayor que 0, significa que arg0 es mayor que arg1. Si aux es igual a 0, significa que arg0 es igual a arg1 en términos de ordenación.

## En la clase:

public int compare(Pista arg0, Pista arg1) {

        int aux=arg0.getProvincia().compareTo(arg1.getProvincia());

        if(aux==0){

            aux=Double.compare(arg0.getKm(),arg1.getKm());

        }

        return aux;

    }

Implementándolo en los filtrados:

public List<Pista> listadoOrdenadoProvinciaKm(String provincia) {

        return pistas.stream()

                .filter(pista -> getPistasProvincia().equals(provincia))

                .sorted(new ComparacionPorProvincia())

                .collect(Collectors.toList());

    }

# Lombok

import lombok.AllArgsConstructor;

import lombok.Data;

@AllArgsConstructor

@Data

# LocalDate

LocalDate fechaActual=LocalDate.now();

    int año = fechaActual.getYear();

    int mes = fechaActual.getMonthValue();

    int dia = fechaActual.getDayOfMonth();

# Estructura de capas Funcional

