17/09/2025

Trabajo Práctico 10 – Lambdas, Streams, Comparator , Optional

Objetivos

- Comprender la sintaxis y el uso de lambdas y streams a través de un ejemplo simple.
- Aplicar los conocimientos para resolver un problema más complejo en un contexto realista.
- Estructurar la lógica de negocio en una clase de servicio, separando la manipulación de datos de la ejecución principal de la aplicación.
- Utilizar la API de Streams para filtrar, transformar, ordenar y agrupar colecciones de objetos de forma eficiente y declarativa.
- Crear y reutilizar instancias de Comparator para lógicas de ordenamiento complejas.
- Manejar resultados que pueden estar ausentes de forma segura y declarativa utilizando Optional.

1. Ejemplo Introductorio.

Ejemplo simple usando una lista de números. Esto nos permitirá conocer la sintaxis de Streams.
Ver archivo: EjemploIntroductorio.java

2. Servicio para Tienda Online

Imagine que estamos desarrollando el backend para una tienda online. Tenemos una lista de productos y necesitamos implementar varias funcionalidades para consultar y procesar esta información: buscar productos, ordenarlos, agruparlos por categoría, realizar cálculos, etc.
Producto.java (El Modelo de Datos), Esta clase representa un producto en nuestro catálogo.

Tarea 1: Filtrar Productos de una Categoría Específica

Consigna: Encuentra todos los productos que pertenecen a la categoría "Electrónica" y guárdalos en una nueva lista.

- Pista: Utiliza stream(), filter() y collect(Collectors.toList()).
- Conceptos clave: Stream, expresión Lambda como Predicate.

Tarea 2: Encontrar el Producto más Caro

Consigna: Encuentra el producto con el precio más alto en toda la tienda.

- Pista: Utiliza stream(), max() y un Comparator. El resultado de max() es un Optional.
- Conceptos clave: Comparator.comparing(), Method Reference (Producto::getPrecio), Optional.

Tarea 3: Transformar Datos - Nombres de Productos con Poco Stock

Consigna: Crea una lista que contenga únicamente los nombres de los productos cuyo stock sea menor a 10.

- Pista: Encadena las operaciones filter() y map().
- Conceptos clave: map() para transformar un Stream<Producto> en un Stream<String>.

Tarea 4: Ordenamiento Complejo

Consigna: Ordena la lista de productos. El criterio principal es por categoría (alfabéticamente) y, para productos de la misma categoría, el segundo criterio es por precio (de mayor a menor).

- Pista: Utiliza sorted() con Comparator.comparing() encadenado con thenComparing(). No olvides usar .reversed() para el orden descendente.
- Conceptos clave: Comparator encadenado (thenComparing), ordenamiento inverso (reversed).

Tarea 5: Agrupar Productos por Categoría

Consigna: Crea un Map donde las claves sean las categorías y los valores sean una lista de los productos que pertenecen a esa categoría.

- Pista: Utiliza el colector Collectors.groupingBy().
- Conceptos clave: Collectors, groupingBy, Map.

Tarea 6: Calcular el Valor Total del Inventario

Consigna: Calcula el valor total del inventario de la tienda (la suma de precio * stock para cada producto).

- Pista: Usa mapToDouble() para convertir el stream a un DoubleStream y luego sum(). O también puedes usar reduce().
- Conceptos clave: Streams primitivos (mapToDouble), operaciones de reducción (sum, reduce).

Tarea 7: Creación y Uso de un Comparator Reutilizable

A veces, un criterio de ordenamiento es complejo o se usa en varias partes de la aplicación. En lugar de definirlo siempre "en línea" con una lambda, podemos crear una instancia de Comparator para reutilizarla.

Consigna: Crea un Comparator que ordene los productos por la cantidad de stock de menor a mayor. Si dos productos tienen el mismo stock, deben ordenarse alfabéticamente por nombre. Luego, úsalo para obtener una lista de productos con bajo stock, priorizando los que tienen menos unidades.

Conceptos:

Comparator.comparingInt(): Una versión optimizada de comparing para tipos primitivos int, que evita el "boxing".

Comparator como objeto: Se demuestra que un Comparator es un objeto que implementa una interfaz funcional y puede ser almacenado en una variable, pasado como argumento y reutilizado.

Tarea 8: Manejo Avanzado de Optional

Optional es más que un simple contenedor para evitar NullPointerException. Ofrece un API funcional muy rica para manejar la ausencia de un valor de forma declarativa.

Consigna: Crea un método que busque un producto por su nombre exacto. Luego, en el main, demuestra tres formas de manejar el Optional resultante:

1. Obtener el producto o un valor por defecto si no se encuentra.
2. Obtener el producto o lanzar una excepción si no se encuentra.
3. Realizar una acción (transformar el valor) solo si el producto está presente.

Conceptos:
Optional.orElse(defaultValue): Devuelve el valor contenido si está presente; de lo contrario, devuelve el defaultValue proporcionado. Es la forma más simple de resolver un Optional.

Optional.orElseThrow(exceptionSupplier): Devuelve el valor si está presente; de lo contrario, lanza la excepción creada por el Supplier provisto. Es la forma idiomática de manejar casos en los que la ausencia de un valor es un error que debe detener el flujo normal.

Optional.map(function): Si el Optional contiene un valor, aplica la función a ese valor y devuelve un nuevo Optional con el resultado. Si está vacío, devuelve un Optional vacío. Permite encadenar operaciones sobre el valor de forma segura sin necesidad de verificar explícitamente si está presente.