INFORMATORIO

Taller de Programación Web

Practica Guiada (Streams)

INFORMATORIO

Práctica Guiada (Streams)

OBJETIVO

El objetivo de este apunte es dar una guía para la resolución de problemas con Streams.

OPTIONAL:

En Java, `null` es usado para representar “nada” o un resultado “vacío”. Frecuentemente, un método retorna `null` si este no posee un resultado a retornar. Esto ha sido fuente de frecuentes `NullPointerException` en programas de Java.

Supongamos que tenemos una clase `Persona`, que posee atributos id, nombre, fecha de nacimiento y sueldo. Y deseamos obtener solo el año de nacimiento.

```
Person homero = new Persona(1, "Homero Simpson", null, 5000.0);
int year = homero.getFechaNacimiento().getYear();
System.out.println(homero.getNombre() + " nació en el año " + year);
```

El código se compila, pero lanzará `NullPointerException` en tiempo de ejecución. El problema es en el valor de retorno del método **`getFechaNacimiento()`** que retorna `null`. Java 8 introduce una clase Optional<T> para lidiar con estos `NullPointerExceptions`. Podemos decir

ahora que los métodos que no retornan nada deberían retornar un `Optional` en lugar de null.

Un Optional es un wrapper (contenedor o envoltura) para valores no nulos que puede contener o no un valor no nulo.

Algunos métodos que nos provee esta clase:

- **isPresent()**: Retorna true si contiene un valor no nulo, caso contrario falso.
- **get()**: Nos devuelve el valor no nulo si contiene un valor no nulo, caso contrario lanzará `NoSuchElementException`.
- **empty()**: Método de clase. Que nos retorna un Optional con un valor no nulo.
- **of(T value)**: Método de clase. Retorna un Optional que contiene un valor especificado como valor no nulo. Si le pasamos nulo, lanzará `NullPointerException`..
- **ofNullable(T value)**: Retorna un Optional que contiene un valor especificado como valor no nulo. Si le pasamos nulo, retornara un Optional.empty().

STREAMS

Una operación de agregación procesa un valor individual de una colección de valores. El resultado de una operación de agregación puede ser un valor primitivo, un objeto, o void. Nótese que un objeto puede ser la representación de una entidad simple como una persona o una colección de valores como un List, Set, o Map, etc.

Un stream es una secuencia de elementos de datos que soportan operaciones de agregación de forma secuencial y paralela. Ejemplos de operaciones de agregación son: Procesar la suma de todos los números

enteros en un stream de integers, obtener el empleado más antiguo (por la fecha de contratación) en un stream de objetos empleados, etc.

Si vemos la definición de streams, nos parecerá que son como colecciones. Pero en qué difieren una de otra? Ambas son abstracciones para una colección de elementos de datos. Las Collections se enfocan en el almacenamiento de los elementos de datos para un acceso eficiente mientras que los Streams se enfocan en el procesamiento de las operaciones de agregación de una fuente de datos que típicamente es una colección, pero no necesariamente.

EJERCICIOS (Streams):

1. Crear un Stream vacío:

   Solución:

```java
public class StreamsEjemplo1 {
    Run | Debug
    public static void main(String[] args) {
        Stream<String> streamVacio = Stream.empty();
        System.out.println(streamVacio);
    }
}
```

   Observación: Ver lo que imprime la consola. ¿Qué nos devuelve?

2. Crear un Stream a partir de una colección. Crear una lista con todas las vocales y pasarla a stream:

Solución:

```java
public class StreamsEjemplo2 {
    Run | Debug
    public static void main(String[] args) {
        Collection<String> vocales = Arrays.asList("a", "e", "i", "o", "u");
        Stream<String> vocalesStream = vocales.stream();
        System.out.println(vocalesStream);
    }
}
```

3. Cargar un stream que contenga N números. Y que comience con el número 40:

```java
public class StreamsEjemplo3 {
    Run | Debug
    public static void main(String[] args) {
        Stream<Integer> numeros = Stream.iterate(40, n -> n +2).limit(20);
        numeros.forEach(n -> System.out.println(n));
        System.out.println(numeros);
    }
}
```

En este caso N = 20 (argumento del método `limit`).
El número 40 es la semilla, que dará el valor inicial, y el segundo argumento de iterate es la función incremento.
Para este caso, el segundo número será 42.

Prestar atención a la operación de agregación `limit`, a la cual le pasaremos el valor de cuantas veces deberá iterar.

EJERCICIOS (Colecciones a Streams):

4. Supongamos que tenemos una lista con objetos de tipo Producto, que poseen atributos nombre, tipo, precio unitario (String, String, BigDecimal), encontrar y devolver el primer producto que su precio unitario sea menor a 200000.00. Caso contrario mostrar mensaje de No encontrado (ver formato del mensaje en la imagen debajo).

```java
package stream.guiada;

import java.math.BigDecimal;
import java.util.List;
import java.util.Optional;

public class StreamsEjemplo4 {
    /**Utilizaremos una constante para esta prueba, pero en un escenario real sera una variable que
     * el usuario/sistema proveera
     */
    private static final BigDecimal PRECIO_BUSQUEDA = new BigDecimal( val: "2000000.00");

    public static void main(String[] args) {
        List<Producto> catalogoProductos = List.of(
                new Producto( nombre: "iPhone 13 Pro",  tipo: "Celulares", new BigDecimal( val: "400000.00")),
                new Producto( nombre: "Samsung Galaxy S21 Ultra",  tipo: "Celulares", new BigDecimal( val: "200000.00")),
                new Producto( nombre: "Motorola Edge Special Edition",  tipo: "Celulares", new BigDecimal( val: "159899.00")));

        Optional<Producto> productoBarato = catalogoProductos.stream()
                .filter(producto -> PRECIO_BUSQUEDA.compareTo(producto.getPrecioUnitario()) > 0)
                .findFirst();
        if(productoBarato.isPresent()) {
            System.out.println("El primer producto que se encontro menor a " + PRECIO_BUSQUEDA + " es: " + productoBarato.get());
        } else {
            System.out.println("No se encontro producto menor a " + PRECIO_BUSQUEDA);
        }
    }
```

Analicemos el código:

- Creamos una lista de Producto
- Al objeto catalogoProductos lo pasamos a stream con el método `stream()`
- Con la operación de agregación `filter`, evaluaremos si la condición que 200000.00 sea mayor que el precioUnitario de cada producto.
- Como usamos un objeto `BigDecimal` para definir el precio, no

podré usar operadores de `<, =, >` (que son para primitivos). Usaremos el método `compareTo`. De forma que: `objeto1.compareTo(objeto2)` -> retorna -1, 0, 1 si `objeto1` es menor, igual, o mayor a `objeto2`. Ese resultado si es primitivo.

- Luego `findFirst()` captura el primer objeto que cumpla la condición del `filter` anterior y su valor es encerrado en un objeto `Optional`. Si ningún objeto cumpliera la condición del filter se retornará un `Optional.empty()` (que dentro encierra un valor `null`).
- Para imprimir el mensaje encontrado o no, preguntaremos si productoBarato.isPresent() (que devolverá `true` si encierra un valor distinto a `null`, caso contrario `false`).
- Para obtener el valor encerrado de un objeto `Optional`, invocamos el método `get()`.

5. Supongamos que del ejercicio anterior queremos lanzar una excepción si no llegáramos a encontrar el primer producto que su precio unitario sea menor a 200000.00 .
   Usaremos una excepción no chequeada (`RuntimeException`) para simplificar el ejercicio. Pero en escenarios reales se prefiere excepciones chequeadas.

```java
package stream.guiada;

import java.math.BigDecimal;
import java.util.List;

public class StreamsEjemplo4 {
    /**Utilizaremos una constante para esta prueba, pero en un escenario real sera una variable que
     * el usuario/sistema proveera
     */
    private static final BigDecimal PRECIO_BUSQUEDA = new BigDecimal( val: "2000000.00");

    public static void main(String[] args) {
        List<Producto> catalogoProductos = List.of(
                new Producto( nombre: "iPhone 13 Pro", tipo: "Celulares", new BigDecimal( val: "400000.00")),
                new Producto( nombre: "Samsung Galaxy S21 Ultra", tipo: "Celulares", new BigDecimal( val: "200000.00")),
                new Producto( nombre: "Motorola Edge Special Edition", tipo: "Celulares", new BigDecimal( val: "159899.00")));

        Producto productoBarato = catalogoProductos.stream()
                .filter(producto -> PRECIO_BUSQUEDA.compareTo(producto.getPrecioUnitario()) > 0)
                .findFirst()
                .orElseThrow(() -> new RuntimeException("No se encontro producto menor a " + PRECIO_BUSQUEDA));
        System.out.println("El primer producto que se encontro menor a " + PRECIO_BUSQUEDA + " es: " + productoBarato);
    }
}
```

Analicemos el código:

- En el ejercicio anterior si ningún producto cumplía la condición del filter, la operación devolverá `Optional.empty()`. Para estos escenarios se puede encadenar el método `orElseThrow()` para lanzar una excepción.
- De esa forma podemos decir que si encontramos un objeto producto que cumpla la condición lo asignaremos a la variable productoBarato, caso contrario lanzaremos exception.

6. Sigamos con el mismo ejemplo. Pero ahora en base a la lista de productos, queremos obtener otra lista que contendrá sólo los productos con precio menor a 200000.00.

```java
package stream.guiada;

import java.math.BigDecimal;
import java.util.List;
import java.util.stream.Collectors;

public class StreamsEjemplo4 {
    /**Utilizaremos una constante para esta prueba, pero en un escenario real sera una variable que
     * el usuario/sistema proveera
     */
    private static final BigDecimal PRECIO_BUSQUEDA = new BigDecimal( val: "2000000.00");

    public static void main(String[] args) {
        List<Producto> catalogoProductos = List.of(
                new Producto( nombre: "iPhone 13 Pro", tipo: "Celulares", new BigDecimal( val: "400000.00")),
                new Producto( nombre: "Samsung Galaxy S21 Ultra", tipo: "Celulares", new BigDecimal( val: "200000.00")),
                new Producto( nombre: "Motorola Edge Special Edition", tipo: "Celulares", new BigDecimal( val: "159899.00")));

        List<Producto> productoBaratos = catalogoProductos.stream()
                .filter(producto -> PRECIO_BUSQUEDA.compareTo(producto.getPrecioUnitario()) > 0)
                .collect(Collectors.toList());
        System.out.println(productoBaratos);
    }
}
```

Analicemos el código:

- En este caso todos los productos que cumplan la condición del filter, serán agregados en una nueva colección que en este caso es una lista (`Collectors` es una clase utilitaria, y el método `toList()` es el método que creará una lista nueva y agrega consecutivamente los productos que reciba).

7. Ahora necesitamos solamente el nombre de los productos con precio menor a 200000.00. Para ellos queremos extraer de un List<Product> a un List<String>.

```java
package stream.guiada;

import java.math.BigDecimal;
import java.util.List;
import java.util.stream.Collectors;

public class StreamsEjemplo4 {
    /**Utilizaremos una constante para esta prueba, pero en un escenario real sera una variable que
     * el usuario/sistema proveera
     */
    private static final BigDecimal PRECIO_BUSQUEDA = new BigDecimal( val: "2000000.00");

    public static void main(String[] args) {
        List<Producto> catalogoProductos = List.of(
                new Producto( nombre: "iPhone 13 Pro", tipo: "Celulares", new BigDecimal( val: "400000.00")),
                new Producto( nombre: "Samsung Galaxy S21 Ultra", tipo: "Celulares", new BigDecimal( val: "200000.00")),
                new Producto( nombre: "Motorola Edge Special Edition", tipo: "Celulares", new BigDecimal( val: "159899.00")));

        List<String> nombreProductoBaratos = catalogoProductos.stream()
                .filter(producto -> PRECIO_BUSQUEDA.compareTo(producto.getPrecioUnitario()) > 0)
                .map(producto -> producto.getNombre())
                .collect(Collectors.toList());
        System.out.println(nombreProductoBaratos);
    }
}
```

Analicemos el código:

- En este caso usaremos la operación de agregación `.map()`. Que tiene como objetivo mapear un objeto producto a uno String, por medio del método `getNombre()` de la clase Producto.

8. Teniendo una lista de Productos queremos imprimir todos los precios unitarios.

```java
package stream.guiada;

import java.math.BigDecimal;
import java.util.List;

public class StreamsEjemplo4 {
    /**Utilizaremos una constante para esta prueba, pero en un escenario real sera una variable que
     * el usuario/sistema proveera
     */
    private static final BigDecimal PRECIO_BUSQUEDA = new BigDecimal( val: "2000000.00");

    public static void main(String[] args) {
        List<Producto> catalogoProductos = List.of(
                new Producto( nombre: "iPhone 13 Pro", tipo: "Celulares", new BigDecimal( val: "400000.00")),
                new Producto( nombre: "Samsung Galaxy S21 Ultra", tipo: "Celulares", new BigDecimal( val: "200000.00")),
                new Producto( nombre: "Motorola Edge Special Edition", tipo: "Celulares", new BigDecimal( val: "159899.00")));

        catalogoProductos.stream()
                .forEach(producto -> System.out.println(producto.getPrecioUnitario()));
    }
}
```

Analicemos el código:

- Aquí utilizaremos `.forEach()` para realizar una operación con cada producto. En este caso imprimir por consola.
- Como no estamos convirtiendo, extrayendo elementos o buscando un elemento específico. No necesitaremos una variable para asignar.

9. Deseamos ahora actualizar el precio unitario de cada producto de la lista en +15%.

```java
package stream.guiada;

import java.math.BigDecimal;
import java.util.List;

public class StreamsEjemplo4 {

    public static void main(String[] args) {
        List<Producto> catalogoProductos = List.of(
                new Producto( nombre: "iPhone 13 Pro", tipo: "Celulares", new BigDecimal( val: "400000.00")),
                new Producto( nombre: "Samsung Galaxy S21 Ultra", tipo: "Celulares", new BigDecimal( val: "200000.00")),
                new Producto( nombre: "Motorola Edge Special Edition", tipo: "Celulares", new BigDecimal( val: "159899.00")));

        catalogoProductos.stream()
                .forEach(producto ->
                        producto.setPrecioUnitario(producto.getPrecioUnitario().multiply(new BigDecimal( val: "1.15"))));
        catalogoProductos.stream()
                .forEach(producto -> System.out.println(producto));
    }
}
```

Output:

```
{ nombre='iPhone 13 Pro', tipo='Celulares',
precioUnitario='460000.0000'}
{ nombre='Samsung Galaxy S21 Ultra', tipo='Celulares',
precioUnitario='230000.0000'}
{ nombre='Motorola Edge Special Edition', tipo='Celulares',
precioUnitario='183883.8500'}
```