Trabajo Practico 3. Ejercicio 13

Java Streams

1. Un Stream es un medio utilizado para leer datos de una fuente y para escribir datos en un destino. Esta es similar a una collection permitiendo además realizar operaciones directamente sobre el Stream. Cada operación devuelve un nuevo stream sobre el cual podemos seguir encadenando operaciones.

1. Los streams se crean de diferentes maneras, de las cuales algunas de ellas son:

* A través de colecciones, ya que estas tienen métodos stream() y parallelStream() a partir de los cuales se puede construir un Stream. 🡪 collection.stream()
* A través de un array 🡪 Arrays.stream()
* Un stream vacio 🡪 Stream<tipo> nombreStream= stream.empty()
* A partir de datos primitivos como Int,Long y Double 🡪 Int/Long/Double Stream nombre= int/Long/DoubleStream.range(valores)
* Crear un stream dando valores especificos 🡪 Stream.off(“Valores”)
* A través de un builder 🡪 Stream<Tipo> builder()
* Stream infinito 🡪 Stream.generate(() -> ‘valor’).limit(cantidadElementos); si la cantidad de elementos no es especificada, se crea un Stream infinto hasta que alcance el limite de la memoria.
* Stream infinito a través de un iterador🡪 Stream.iterate(condiciones).limit(CantidadElementos); si la cantidad de elementos no es especificada, se crea un Stream infinito hasta que alcance el limite de la memoria.

**Ejemplos de como crear un stream:**

Vector<int>nuevoVector=new Vector<int>();

nuevoVector.stream()

**Ejemplo 2:**

Stream.off(‘A’,’B’,’C’,’D’)

1. Las operaciones intermedias devuelve un nuevo Stream permitiendo la continuidad de los pasos o funciones sobre el mismo. Las funciones no se ejecutan hasta que se invoquen en una operación terminal.

Alguna operaciones intermedias son:

* Filter: Produce un Stream que contiene solo los elementos que satisfacen una condición.
* Sorted: Produce un flujo en el que los elementos están ordenados; el nuevo flujo tiene el mismo numero de elementos que el flujo original
* Distinct: Produce un flujo que contiene sólo los elementos únicos.
* Limit : Produce un flujo con el número especificado de elementos a partir del inicio del flujo original.
* Map: Produce un flujo en el que cada elemento del flujo original esta asociado a un nuevo valor; por ejemplo asociar valores numéricos a los cuadrados de los valores numéricos. El nuevo flujo tiene el mismo numero de elementos que el flujo original.



1. Finalizadores u Operaciones terminales

Una operacion terminal inicia el procesamiento de las operaciones intermedias de una canalización de flujo y produce un resultado; no realizan la tarea de recorrer sino que crean un nuevo flujo de datos que contienen los valores del stream inicial que coinciden con el predicado dado.

Las operaciones terminales son:

* forEach: realiza procesamiento sobre cada elemento en un Stream
* Average: Calcula promedio de los elementos de un Stream numérico
* Max: Localiza el valor mas grande de un Stream
* Min: Localiza el valor mas chico de un Stream
* Count(): Devuelve un entero con el numero de elementos que contiene el Stream
* Average(): Devuelve la media aritmética del Stream
* Sum(): Devuelve la suma de los elementos del Stream
* Reduce(): Realiza una reducción en el Stream utilizando Operaciones Binarias
* findFirst: Devuelve el primer elemento del Stream que coincida con la condición del predicado, o un optional vacío si no se cumple la condición.
* findAny: Devuelve elemento aleatorio del Stream que coincida con la condición del predicado, o un Optional vacia si no se cumple con la condición.
* anyMatch: Devuelve True si la condición del predicado se comple para cualquier elemento del Stream; False para el resto de los casos.
* allMatch: Devuelve True si la condición del predicado se cumple para todos los elementos del Stream, False para el resto de los casos.

**Ejemplo:**

**Int[] valores={3,10,9,1,4,8};**

**intStream.of(valores);**

**.forEach(valor -> System.out.println(“%d”,valor);**

**System.out.println(“CONTADOR: “, IntStream.of(valores).count());**

**System.out.println(“Maximo: “, IntStream.of(valores).max().getAsInt());**

**System.out.println(“Minimo: “, IntStream.of(valores).min().getAsInt());**

1. Los Streams son Lazy porque las operaciones intermedias no se evalúan hasta que se invoque la operación terminal.

Cada operación intermedia crea una nueva secuencia, almacena la operación y devuelve una nueva secuencia. Un ejemplo de esto es la función filter(), que crea un nuevo stream que contiene los elementos del stream inicial que coincidan con la función Lambda que se le indico.

1. Orden de ejecución:

El procesamiento de un Stream puede ser secuencial o paralelo.

En el secuencial, los elementos se procesan en el orden de la fuente del Stream. Si se ordena el Stream, se garantiza que el procesamiento coincidirá con el orden de la fuente.

El modo paralelo permite el uso de múltiples hilos en múltiples núcleos, pero no hay garantía del orden que se procesan los elementos. Si se invocan varios métodos en una secuencia Stream, no es necesario invocar todos los métodos. Por ejemplo, si un Stream se filtra y el numero de elementos reduce a uno, no se producirá una llamada posterior a un método como la Sort.

**Buenas practicas:**

**El uso de los Streams en Java facilitan mucho la programación debido a la simplificación de código con respecto a la programación imperativa. Estos ayudan a la hora de trabajar con gran cantidad de datos. Uno de los casos en el cual los Streams nos favorecen es en el For.**

**De forma tradicional:**

****

****

**Usando Streams**

