Alumno: Agustin Bascoy

Profesores: Carlos Di Cicco y Nelson Omar Di Grazia.

Tema abordado: Streams en java.

1. Un stream es un medio utilizado para leer datos de una fuente y para escribir datos en un destino. Tanto la fuente como el destino pueden ser archivos, sockets, memoria, cadena de caracteres, y también procesos. Se puede entender como un flujo de datos que se crea a partir de la lectura de datos de una fuente.
2. Hay diferentes formas de crear un stream. Algunas de ellas son:

* A través de colecciones 🡪 collection.stream()
* A través de un Array 🡪 Arrays.stream()
* Crear un stream dado valores específicos 🡪 Stream.of(“valores)
* Un stream vacío 🡪Stream <tipo> nombre= Stream.empty()
* A través de un Builder 🡪 Stream.<tipo>builder() (En caso de no especificar el tipo, se creará un stream de tipo Object)
* Crear un Stream infinito 🡪 Stream.generate(() ->” valor”).limit(cantidadDeElementos) (Si no se indica la cantidad de elementos, se creará un stream infinito hasta que alcance el límite de memoria)
* Crear un stream infinito a través de un iterador 🡪 Stream.iterate(condiciones).limit(cantidadDeElementos) (Si no se pone la cantidad de elementos, se creará un stream infinito hasta que se alcance el límite de memoria).
* A partir de datos primitivos como Int, Long y Double 🡪

Int/Long/DoubleStream nombre= Int/Long/DoubleStream.range(valores)

Ejemplo: List<String>miLista=ArrayList()

miLista.stream()

Ejemplo 2: Stream.of(“a1”,”c2”,”b1”)

1. Las operaciones intermedias devuelven un nuevo Stream modificado, es decir, son los métodos que modifican los stream y lo devuelven modificado. Se ejecutan siempre de manera lazy, es decir, que en caso de ejecutarse una operación de búsqueda que por ejemplo quisiera encontrar el primer número primo de una colección, esta no debería examinar todos los valores Algunas operaciones intermedias son: Map(), filter(), sorted(), limit(), skip() y distinct().

Cada operación, crea un flujo de datos nuevo con los parámetros que se les pase.

Ejemplo:

List<String>miLista=ArrayList()

miLista.stream()

.filter(s ->s.startwith(“c”)) 🡪 Esta funcion, filtra los elementos que coincidan con la condición que se le pasa como parámetro.En este caso, devuelve un nuevo flujo de datos con todos los elementos que comiencen con la letra “c”.

.sorted() 🡪 Devuelve otro nuevo flujo de datos, donde los elementos empiecen con la letra “c” y esten ordenados.

.map(String :: toupperCase) 🡪 El map modifica los elementos de un stream en base a la función que se le indique como parámetro. En este caso, todos los elementos serán convertidos a mayúsculas.

.distinct() 🡪Esta función devuelve un nuevo flujo de datos, con los elementos no repetidos del stream anterior.

.limit(cantidadDeElementos)🡪Esta función devolverá un nuevo stream con la cantidad de elementos que se le indique como parámetro.

1. Las operaciones terminales a diferencia de las intermedias, debe pasar por todos los elementos del stream para generar un resultado. Cuando se ejecuta una operación de este tipo, el stream se da por concluido/cerrado y no se pueden realizar más funciones de cualquier tipo. Estas operaciones no devuelven ningún Stream. Algunas operaciones terminales son: forEach(), toArray(), reduce(),collect(), max(), min(), Ejemplo:

List<String>miLista=ArrayList()

miLista.stream()

.forEach(System.out::println)🡪 Esta función realiza la acción que se le pase por parámetro a cada elemento del stream. En este caso, se imprimirían todos los elementos que el stream contenga.

.toArray(String[]::new))🡪 Esta función pasa el stream, a un array, del tipo que se le pase por parámetro. En este caso, crea un nuevo array de tipo string y pasa los elementos del stream, a ese nuevo array.

.reduce(valorIncial, queSeSuma) 🡪 Esta función permite hacer una reducción de los resultados del stream, la cual consiste en guardar el resultado total, en una variable. Los parámetros indican desde que valores toma hasta que valores. Un ejemplo puede ser si tenemos un stream de números. Entonces usaríamos .reduce(0,integer::sum), la cual sumaria el valor incial y cada uno de los números que tenemos en el stream.

.collect(Collectors.toList())🡪Esta función, recoge los datos del stream, y los devuelve en un contenedor especifico, en este caso, los devuelve en una lista.

.max(comparador)🡪Esta función, devuelve el elemento mas grande de un stream. Tambien se usa mucho con tipos de datos numéricos. Un ejemplo: Tenemos un stream de una lista de estadios, cada estadio, posee una capacidad, y se quiere saber cual es lamayor capacidad de un estadio, entonces implementamos:

.max(Comparator.comparating(Estadio::getCapacidad()))🡪Entonces, retorna la máxima capacidad q tiene un estadio.

.min(comparador)🡪 Se utiliza igual que el max() pero devuelve el menor elemento de la comparación.

1. Las operaciones intermedias son lazy debido a que cuando se ejecutan, estas 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. Un ejemplo de esto es la función filter(), en realidad, no realiza ningún filtrado, sino que crea un nuevo stream que contiene los elementos del stream inicial que coincidan con la función lambda que se le indico.
2. Orden de ejecución: Primero se obtiene el stream, luego, se realizan las operaciones intermedias, y por último se realizan las operaciones terminales.

Ejemplo:

List<int>números=ArrayList<int>(1,2,3,4,5,6,7)

Números.stream()

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

forEach(System.out:.println)

Este código, crea un stream a partir de una lista de números, luego filtra ese stream para crear otro nuevo, solo con los números que sea pares, y por último, imprime cada numero par de ese stream.

1. El uso de streams facilita mucho la programación debido a que simplifica mucho un código con respecto a la programación imperativa. Lo simplifica de manera que es más corto, fácil de entender y más eficiente a la hora de trabajar con estructuras de datos o contenedores.