Lenguaje Java Streams.

Ref: https://stackify.com/streams-guide-java-8/

- https://www.baeldung.com/java-streams
- https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html
- https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html
- https://www.baeldung.com/java-streams
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/stream/package-
- ummary.html
- https://www.adictosaltrabajo.com/2015/12/04/expresiones-lambda-con-java-8/
- https://www.ecodeup.com/entendiendo-paso-a-paso-las-expresiones-lambda-en-java/
- https://www.arquitecturajava.com/el-concepto-de-java-8-reference-method/
- http://tutorials.jenkov.com/java-collections/streams.html
- https://www.arquitecturajava.com/java-stream-collectors-y-su-uso/

Stream

Interface Stream<T>

- Secuencia de elementos sobre los que se pueden realizar diferentes operaciones de procesamiento
- Permiten ejecución anidada de funciones, concatenación de operaciones sobre una lista de datos para facilitar la comprensión de la operación.
- Después de crearlos solo se pueden utilizar una vez.
- Permite procesamiento paralelo
- Las operaciones se pueden encolar
- No es más eficaz que la programación tradicional.
- https://docs.oracle.com/javase/9/docs/api/java/util/stream/Stream.html

Stream

Sin almacenamiento. Un stream no es una estructura de datos que almacena elementos sino que transmite elementos de una fuente, como una estructura de datos, una matriz, una función generadora o un canal de E/S, a través de un canal de operaciones.

De naturaleza **funcional**. Una operación en una secuencia produce un resultado, pero no modifica su fuente. Por ejemplo, filtrar una secuencia obtenida de una colección produce una nueva secuencia sin los elementos filtrados, en lugar de eliminar elementos de la colección de origen.

Búsqueda de pereza. Muchas operaciones de transmisión, como el filtrado, el mapeo o la eliminación de duplicados, se pueden implementar de manera perezosa, lo que expone oportunidades de optimización. Por ejemplo, "encontrar la primera cadena con tres vocales consecutivas" no necesita examinar todas las cadenas de entrada. Las operaciones de flujo se dividen en operaciones intermedias (que producen flujo) y operaciones terminales (que producen valor o efectos secundarios). Las operaciones intermedias siempre son vagas.

Posiblemente ilimitado. Si bien las colecciones tienen un tamaño finito, los flujos no lo tienen Las operaciones de cortocircuito como limit(n) o findFirst() pueden permitir que los cálculos en flujos infinitos se completen en un tiempo finito.

Consumible. Los elementos de un stream sólo se visitan una vez durante la vida de un stream. Al igual que un iterador, se debe generar una nueva secuencia para volver a visitar los mismos elementos de la fuente.

Utilización

La utilización de la secuencia de valores de un Stream se puede procesar con una interfaz funcional de las que hemos visto en el tema anterior:

```
strPersonas.filter((Predicate<? super Persona>) new Predicate<Persona>() {
  @Override
  public boolean test (Persona p) {
    if (p.edad>50)
       return true;
    else
       return false;
  }
}).forEach(System.out::println);
```

```
lista.add(new Persona("Ana",130));
lista.add(new Persona("Oscar",45));
lista.add(new Persona("Carlos",70));
lista.add(new Persona("Antonio",5));
Stream<Persona> strPersonas =
lista.stream();
```

Pero es más fácil usando lambdas:

```
  strPersonas = lista.stream();
//IMPORTANTE:Hay que generarlo otra vez porque una vez que lo utilizas ya no es procesable
strPersonas.filter(p->p.edad>50).forEach(System.out::println);
```

Stream- Tipos básicos

- IntStream
- LongStream
- DoubleStream

El método boxed permite conversión a Stream <T>

```
int[] v={3,4,5};
IntStream strInt= Arrays.stream(v);
Stream<Integer> strInteger=strInt.boxed();
```

Stream- Obtención

Métodos que devuelven un Stream:

- Stream.of (lista de valores separados por coma): ordenado y secuencial con los valores indicados.
- Arrays.stream(T[]). Secuencial basado en el array
- Stream.empty(). Secuencial y vacío.
- Stream.iterate (T semilla, UnaryOperator<T> f). Secuencial, infinito y ordenado compuesto por semilla, f(semilla), f(f(semilla)),…. :

```
Stream.iterate(453,
        s->s+1).limit(4).forEach(System.out::println);
```

- Collection<T>.stream(), Collection<T>.parallelStream () : secuencial, o paralelo basado en la colección
- static <T> Stream<T> generate(Supplier<? extends T> s) : infinito, secuencial y no ordenado creado por medio del supplier (implementado normalmente con una exp. Lambda)
- Stream <String> Files.lines(Path path): Stream con las líneas leídas del fichero

Stream- Operaciones Intermedias

Intermedias: las que toman un Stream y devuelven un Stream

- **Filtrado**
  - filter (predicate <T>); Filtra los que no cumplan la condición
  - limit(n): Obtiene los n primeros elementos
  - skip(m) : salta los m primeros elementos
- **Mapeo**:
  - map (Function<T,R>) transforma valores utilizando expresión lambda de tipo Function
  - mapToInt(..), mapToDouble(..).. Transforma en tipos básicos y obtiene IntStream, DoubleStream,..
- **Procesamiento**:
  - Stream<T> peek(Consumer<? super T> action) Devuelve los elementos recibidos después de aplicarle la acción indicada en el parámetro.

Stream- Operaciones Terminales

Terminales: toman un Stream y devuelven algo que no es Stream (un valor, una colección, etc.) o realizan una operación

Ejemplos:

- forEach(..)
- Transformación en una colección
- Reducción
- Búsqueda

Streams. Ejem 1

```
public class FiltradoPersona {
  public static void main(String[] args) {
Persona p1= new Persona("Juan",10);
Persona p2= new Persona("Ana",20);
Persona p3= new Persona("Oscar",15);
Persona p4= new Persona("Pepe",18);
List<Persona> lista= new ArrayList<Persona>();
lista.add(p1);
lista.add(p2);
lista.add(p3);
lista.add(p4);
//se van pasando diferentes filtros al stream
Persona filtroPersona=
 lista.stream().filter(elemento->elemento.getEdad()>15).findFirst().get();
//La menor persona de los mayores de 15
 System.out.println(filtroPersona.getNombre()+" "+filtroPersona.getEdad());
  }
}
```

Streams. Ejem 2

```
public class StreamConLambda1 {
 public static void main(String[] args) {
 ArrayList<Persona> lista= new ArrayList<Persona>();

 lista.add(new Persona("Ana",10));
 lista.add(new Persona("Oscar",45));
 lista.add(new Persona("Carlos",70));
 lista.add(new Persona("Antonio",5));

 double resultado=lista.stream()
 .mapToDouble(pers->pers.getEdad()*12)
 .filter(edad->edad<=120)  //menos de 120 meses
 .sum();

 System.out.println(resultado);
 }
}
```

Streams. Ejem 3

```
public class StreamConLambda1 {
 public static void main(String[] args) {
 ArrayList<Persona> miLista= new ArrayList<Persona>();
 miLista.add(new Persona("Miguel",15));
 miLista.add(new Persona("Alicia",34));
 miLista.add(new Persona("Carlos",72));
 miLista.add(new Persona("Alicia",12));
 List<String> listaNombres= miLista.stream().
   map((p)->p.getNombre()).  //transformo con Function implicita
   collect(Collectors.toList());  //convierto en lista de String
 for (String s: listaNombres)
   System.out.println(s);
 }
}
```

Ver stream reduction para aplicación en paralelo de reducciones

Stream- Operaciones Terminales búsqueda

- allMatch(Predicate<T>): verifica si todos los elementos de un stream satisfacen un predicado.
- anyMatch(Predicate<T>): verifica si algún elemento de un stream satisface un predicado.
- noneMatch(Predicate<T>): opuesto de allMatch(…)
- findAny(): devuelve en un Optional<T> un elemento (cualquiera) del stream. Recomendado en streams paralelos.
- findFirst() devuelve en un Optional<T> el primer elemento del stream. NO RECOMENDADO en streams paralelos.

Nota: Optional es una clase que puede o no contener un valor no-null

Streams. Ejem Búsqueda

```
public class Busquedas {
public static void main(String[] args) {
 ArrayList<Persona> lista= new ArrayList<Persona>();

 lista.add(new Persona("Ana",130));
 . . .
 Stream<Persona> strPersonas= lista.stream();
 boolean jubilados= strPersonas.anyMatch((p)->p.edad>65);
 System.out.println("Hay jubilados:"+jubilados);

 strPersonas= lista.stream();//Hay que volver a crearlo porque solo se
puede usar 1 vez.
 Optional<Persona> unJoven= strPersonas.filter((p)->p.edad<18).findAny();
 if (unJoven.isPresent())
   System.out.println("Un menor de edad es: "+unJoven.get());
 else
 System.out.println("No hay menores de edad");
}
}
```

Stream- Oper. Terminales Cálculos

- Optional<T> reduce(BinaryOperator<T> accumulator) Reduce el Stream usando una función asociativa.
- T reduce(T identity, BinaryOperator<T> accumulator) Reduce el Stream usando un valor inicial y una función asociativa.
- <U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner) Reducción utilizando la identity , acumulación y función indicadas.
- long count() : Devuelve el número de elementos del Stream
- Optional<T> min(Comparator<? super T> comparator) / Optional<T> max(Comparator<? super T> comparator) Valor mínimo/máximo del Stream de acuerdo al comparador
- https://www.baeldung.com/java-stream-reduce
- https://docs.oracle.com/javase/tutorial/collections/streams/reduction.html

Streams. Ejem Búsqueda

```
IntStream vs=new Random().ints(10,0,100);// genero 10 nums aleat entre [0,100)
int [] v= vs.toArray();

System.out.println(Arrays.toString(v));
vs=Arrays.stream(v);
System.out.println("La suma es:"+vs.reduce(0, (subtot,dato) -> subtot+dato));
//System.out.println("La suma es:"+vs.reduce(0,Integer::sum));
vs=Arrays.stream(v);
OptionalInt m=vs.reduce(Integer::max);
System.out.println("El mayor es:"+m.orElse(-1));
```

Stream- Oper. Terminales Ordenación

- Stream<T> sorted() Ordenación de acuerdo al orden natural
- Stream<T> sorted(Comparator<? super T> comparator) Ordenación de acuerdo al comparador

- Ejem

```
Comparator<Persona> comparadorMultiple=
Comparator.comparing(Persona::getNombre).thenComparing(Comparator.comparing(Persona::getApellidos)).thenComparing(Comparator.comparing(Persona::getEdad));
   lista.stream().sorted(comparadorMultiple).forEach(System.out::println);
```

Streams. Ejem Ordenación

```
IntStream vs=new Random().ints(10,0,100);
int [] v= vs.toArray();
System.out.println(Arrays.toString(v));

vs=Arrays.stream(v).sorted();
System.out.println("Ordenado:"+Arrays.toString(v));

vs=Arrays.stream(v);
Stream<Integer> vsi=vs.boxed();
System.out.print("En orden inverso:");
vsi.sorted(Comparator.reverseOrder()).forEach(i->System.out.print(i+","));
System.out.println(); //sorted con Comparator no es aplicable a IntStream

vs=Arrays.stream(v);
vsi=vs.boxed();
System.out.print("Ordenando solo por últimos dígitos :");
vsi.sorted((n1, n2)-> n1%10-n2%10).forEach(i->System.out.print(i+","));
```

Stream- Oper. Intermedias Map

- <R> Stream<R> map(Function<? super T,? extends R> mapper) : Map transforma un objeto en otro usando la Function<T, R>
- <R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper): Aplana los datos cuando se tiene anidamiento de estructuras (por ejem. un atributo de los objetos de la lista es otra lista).
- Para Streams básicos temenos:
  - IntStream mapToInt(ToIntFunction<? super T> mapper)
  - LongStream mapToLong(ToLongFunction<? super T> mapper)
  - DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper)

Streams. Ejem Map

```
ArrayList<Persona> miLista= new ArrayList<Persona>();
 miLista.add(new Persona("Miguel",45));
 miLista.add(new Persona("Ana",34));
 miLista.add(new Persona("Carlos",32));
 Stream<Persona> strPersonas= miLista.stream();
 strPersonas.map(Persona::getNombre)
   .map(String::toUpperCase) //.map(String::length)
   .forEach(System.out::println);
 miLista.get(0).addHijo(new Persona ("Pepito",12));
 miLista.get(0).addHijo(new Persona ("Juanito",8));
 miLista.get(2).addHijo(new Persona ("Rosita",10));
 System.out.println("Edades de los hijos de todas las personas.");
 miLista.stream().map((Persona p) -> p.hijos).flatMap(hijos ->
hijos!=null?hijos.stream():Stream.empty()) //si no tiene hijos no se procesa nada
   .map((Persona p) -> p.getEdad()).forEach(System.out::println);
 System.out.println("Números distintos.");
 int[][] mat= { {11, 2, 3}, {9, 5, 6}, {11}, {10, 8, 9,10} };
 Stream.of( mat)
    .flatMapToInt(x -> Arrays.stream(x))//unifica en un solo stream
  //  .map(IntUnaryOperator.identity())
    .distinct()
    .forEach(i->System.out.print(i+","));
```

Streams. Ejem Map

```
public class Persona {
private String nombre;
 int edad;
 List<Persona> hijos=null;
public Persona(){}
public Persona(String nombre, int e) {
this.nombre = nombre;
this.edad = e;
}
void addHijo(Persona hijo) {
if (hijos==null) hijos= new ArrayList<Persona>();
hijos.add(hijo);
}
```

Stream- Oper. Terminales Collectors

- Permite generar una coleccion mutable desde un Stream.
- Son métodos de la clase Collectors
- static <T> Collector<T,?,Long> counting() Devuelve un Collector que cuenta el número de elementos
- static <T> Collector<T,?,Optional<T>> minBy/maxBy(Comparator<? super T> comparator) Devuelve el menor/mayor elemento de acuerdo al comparador.
- static <T> Collector<T,?,Integer> summingInt/Long/Double(ToIntFunction<? super T> mapper) Devuelve la suma (int) de los elementos
- static <T> Collector<T,?,Double> averagingInt/Long/Double(ToIntFunction<? super T> mapper) Devuelve la media(int) de los elementos
- static <T> Collector<T,?,T> reducing(T identity, BinaryOperator<T> op) Devuelve la reducción de los elementos de acuerdo a la operación indicada. Hay varios tipos de reducing con diferentes parámetros. En los parámetros de la expression lambda primero está el acumulador y luego el nuevo elemento.o.
- static Collector<CharSequence,?,String> joining(CharSequence delimiter) Devuelve un Collector que concatena los elementos en un string separados por el delimitador

Streams. Ejem Collectors

```
static Stream<Persona> crearPersonas(){
 ArrayList<Persona> miLista= new ArrayList<Persona>();
 miLista.add(new Persona("Miguel",15));
 miLista.add(new Persona("Alicia",34));
 miLista.add(new Persona("Carlos",32));
 Stream<Persona> strPersonas= miLista.stream();
 return strPersonas;
}

Stream<Persona> strPersonas=crearPersonas();
long numPer = strPersonas.collect(Collectors.counting());
System.out.println("número de personas " + numPer);
strPersonas=crearPersonas();
Persona mayor = strPersonas.collect(Collectors.maxBy(
       Comparator.comparingInt(Persona::getEdad))).get();
System.out.println("Persona más vieja " + mayor);

strPersonas=crearPersonas();
Persona menor = strPersonas.collect(Collectors.minBy(
   (x, y) -> x.getEdad() - y.getEdad())).get();
System.out.println("Persona más joven " + menor);
```

Streams. Ejem Collectors

```
strPersonas=crearPersonas();
int sumEdades = strPersonas.collect(Collectors.summingInt(p-> p.edad));
System.out.println("Suma edades " + sumEdades);

Stream<Integer> s = Stream.of(5,7,12,34);
Integer tot = s.collect(Collectors.reducing(
                        (acum, nuevo)-> nuevo - acum)).orElse(-1);
System.out.println(tot);  //24= (34-(12-(7-5)))

strPersonas=crearPersonas();
String nombres= strPersonas.map(p->p.getNombre())
  .collect(Collectors.joining("->"));
System.out.println("nombres: " + nombres);
```

Stream- Oper. Term. Collectors Grouping by

- Agrupan los elementos del Stream por un valor.
- Devuelve un Map (HashMap) con los elementos seleccionados de cada grupo
- Hay varios tipos como :
  - static <T,K> Collector<T,?,Map<K,List<T>>> **groupingBy**(Function<? super T,? extends K> classifier)

- Otra forma es agrupar según cumplan o no una condición
  - static <T> Collector<T,?,Map<Boolean,List<T>>> **partitioningBy**(Predicate<? super T> predicate)

Streams. Ejem groupingBy

```
Stream<Persona> strPersonas=crearPersonas();
Map<Integer, List<Persona>> listaPersPorEdades = strPersonas
  .collect(Collectors.groupingBy(Persona::getEdad));
System.out.println("lista de personas por edades " + listaPersPorEdades);

strPersonas=crearPersonas();
Map<Integer, Long>  numEdades = strPersonas
  .collect(Collectors.groupingBy(Persona::getEdad,Collectors.counting()));
System.out.println("número de personas de cada edad " + numEdades);

strPersonas=crearPersonas();
Map<Boolean, List<Persona>> edadesMasMenos18 =
  strPersonas.collect(Collectors.partitioningBy(e -> e.getEdad() >= 18));
System.out.println("número de personas mayores y menores de edad " +
edadesMasMenos18);
```

Streams. Ejem groupingBy

```
Stream<Persona> strPersonas=crearPersonas();
Map<String, Optional<Persona>> mayorDeCadaApellido =
strPersonas.collect(Collectors
    .groupingBy(Persona::getApellido,
       Collectors.reducing(BinaryOperator
          .maxBy(Comparator
            .comparing(Persona::getEdad))));

System.out.println("El mayor de cada apellido
es:"+mayorDeCadaApellido);
```

Stream- Oper. Term. Collectors Collection

- Permiten obtener colecciones a partir de Stream

- static <T> Collector<T,?,Set<T>> **toSet**() :convierte en **Set**
- public static <T> Collector<T,?,List<T>> **toList**() :convierte en **List**
- static <T,K,U> Collector<T,?,Map<K,U>> **toMap**(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper) :convierte en **Map**.

Streams. Ejem paso a Collections

```
Stream<Persona> strPersonas=crearPersonas();
Set<Persona> setPersonas = strPersonas.collect(Collectors.toSet());

strPersonas=crearPersonas();
List<Persona> listaPersonas =
strPersonas.collect(Collectors.toList());

strPersonas=crearPersonas();
Map<String, Integer> mapPersonas= strPersonas.distinct()
  .collect(Collectors.toMap(Persona::getNombre, Persona::getEdad));

System.out.println("lista de personas  " + listaPersonas);
```

Stream- Oper. Intermedia Filter

Stream<T> filter(Predicate<? super T> predicate)

- Permite seleccionar los elementos que cumplen una condición.
- La condición se indica con un predicado

```
lista= new ArrayList<Persona>();
lista.add(new Persona("Ana",130));
lista.add(new Persona("Oscar",45));
lista.add(new Persona("Carlos",70));
lista.add(new Persona("Antonio",5));
lista.stream().filter(p -> p.getEdad() >= 18 && p.getEdad() <= 100)
   .forEach(persona -> System.out.printf("%s (%d años)%n",
      persona.getNombre(), persona.getEdad()));
```

Referencias a métodos

Para hacer el código más conciso.

No se utilizan argumentos, los deduce.

- Clase::metodoEstatico: referencia a un método estático.
- objeto::metodoInstancia: referencia a un método de instancia de un objeto concreto.
- Tipo::nombreMetodo: referencia a un método de instancia de un objeto arbitrario de un tipo en particular.
- Clase::new: referencia a un constructor.

```
class Persona {
      public static int
      compararPorNombre(Persona a,
            Persona b) {
     return
a.nombre.compareTo(b.nombre);
 }
```

Ejem. 3 sistemas de referencia

lista.sort((Persona p1, Persona p2) -> {
return p1.getNombre().compareTo(p2.getNombre()); });
lista.sort((p1, p2) -> p1.getNombre().compareTo(p2.getNombre()));
//con referencia a métodos. Los parámetros los obtiene de la colección
lista.sort(Persona::compararPorNombre); //método static

Ref. métodos de objeto instancia obj. concreto

```
public class ComparadorPersonas {
  public int compararPorNombres(Persona a, Persona b){
  return a.getNombre().compareTo(b.getNombre());
  }
  public int compararPorEdad(Persona a, Persona b){
  return a.getEdad()-b.getEdad();
  }
}
static void refMetodo2(){
   ComparadorPersonas cp= new ComparadorPersonas();
   //método no estático
    lista.sort(cp::compararPorEdad);
}.
```

Ref. métodos de objeto arbitrario

```
static void refMetodo3(){
//método de instancia (getNombre y compareToIgnoreCase) de un tipo pero conocido
  List<String> soloNombres=
        lista.stream().map(Persona::getNombre).collect(Collectors.toList());
  soloNombres.sort(String::compareToIgnoreCase);
  System.out.println(soloNombres);
}
```

Ref. método constructor

```
//referencia a un constructor
List<Empleado> empleados=lista.stream().filter(p->p.edad>=18).map(Persona::getNombre)
.map(Empleado::new).collect(Collectors.toList());
empleados=empleados.stream().map((Empleado e)->{return new Double(1000);})
.map(Empleado::new).collect(Collectors.toList());
empleados.forEach(System.out::println);

public class Empleado {
  String nombre;
  double sueldo;
  public Empleado(String nombre) {
     this.nombre = nombre;
  }
  public Empleado(double sueldo) {
     this.sueldo = sueldo;
  }
public Empleado setSueldo(Double s){
  sueldo=s;
  return this;
```

Comparación stream frente a bucles

- En general los Streams son más lentos que los bucles.
- Existen comparaciones que así lo demuestran: https://www.baeldung.com/java-streams-vs-loops