¿QUÉ SON LAS *OPERACIONES* DEL API STREAM?

En la programación funcional java, normalmente los problemas se resuelven usando objetos Streams sobre los que se definen una cadena de operaciones que procesan los datos de dicho Stream. Esas operaciones las podemos pensar como funciones aunque realmente son métodos de ahí que se eviten los términos "método" y "función" y se use el de *operación*.

Como indicamos, las operaciones se van encadenando de forma que la salida de una es la entrada de la siguiente formando una cadena o tubería. Otra característica de una "operación" es que suelen tener como parámetro una expresión lambda que sí es "una función".

Por lo tanto trabajamos con dos tipos de funciones:

- expresiones lambda que son auténticas funciones anónimas
- operaciones. Las operaciones las podemos ver como pseudo funciones que realmente son métodos que admiten lambdas de parámetro imitando al comportamiento de las funciones high-order de la auténtica programación funcional.

Seguiremos fundamentalmente la siguiente página:
https://www.oracle.com/technetwork/es/articles/java/expresiones-lambda-api-stream-java-2737544-esa.html

Pero haremos alguna variante y simplificación.

Recuerda que para trabajar con un Stream típico tengo que:

1. Crear el Stream
2. Ejecutar operaciones intermedias sobre el Stream: filter,map,order, ...
3. Ejecutar operaciones de terminación: collect, sum,...

OPERACIONES DE CREACIÓN DE STREAMS

ya vimos cómo crear Streams en el boletín pasado:generate(), of(), stream(), ... y hay muchas más, por ejemplo podemos convertir un array en un Stream con la utilidad stream de Arrays.

```
public class App {
   public static void main(String[] args) {

      String cadena="hola a todos los que me escuchan";
      String[] miArray=cadena.split(" ");
      Arrays.stream(miArray).forEach(s->  System.out.println(s)  );
   }
}
```

y como último ejemplo de creación de streams, los típicos bucles de programación iterativa podemos concebirlos como un stream de valores que se pueden generar con range() e iterate()

Por ejemplo

```
import java.util.stream.IntStream;

class App{
   public static void main(String ... args){
      IntStream.range(1, 6).forEach(i ->  System.out.print(i+ " "));
      System.out.println();
      IntStream.iterate(0, n -> n < 15, n -> n + 2).forEach(i ->  System.out.print(i+ " "));
      }
}
```

OPERACIONES INTERMEDIAS

Veremos algunas relativas al filtrado, mapeo, ordenación y mixtas de creación y filtrado.

OPERACIONES DE FILTRADO:

- **filter()**
- **distinct()**
- **limit()**
- **skip()**

filter()

filter(Predicate<T>):Stream<T>

Se ejecuta sobre los datos de un Stream y devuelve un Stream que contiene sólo los elementos que cumplen con el predicado pasado por parámetro.

Ejemplo: Ciudades que empiezan por "C" (mayúscula)

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class App {
    public static void main(String... args) {
        List<String> ciudades = Arrays.asList("Lugo", "Coruña", "Zaragoza", "cádiz","Córdoba");
        //Stream de ciudades cuya primera letra es C
        Stream<String> stream = ciudades.stream().filter(s -> s.charAt(0) == 'C');
        List<String> resultado=stream.collect(Collectors.toList());
        System.out.println(resultado);
    }
}
```

Observaciones:

- En el ejemplo original el Stream tiene tipo crudo. No sé si es un "despiste" del programador que hizo el manual o bien es simplemente una forma correcta de trabajar. Yo utilicé el tipo.
- Para poder imprimir el resultado del procesamiento del String utilizamos ya una función de acabado que introduce los datos del Stream en una list. Se trata de la operación collect() a la que para indicarle que queremos una lista le pasamos como parámetro Collectors.toList()

distinct()

+distinct():Stream<T>

Retorna un Stream sin elementos duplicados. Depende de la implementación de *+equals(Object):boolean*.

Ejemplo: Ciudades sin duplicados

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class App {
    public static void main(String... args) {
        List<String> ciudades = Arrays.asList("Lugo", "Coruña", "Zaragoza", "Coruña","Coruña");
        Stream<String> stream = ciudades.stream().distinct();
        List<String> resultado=stream.collect(Collectors.toList());
        System.out.println(resultado);
    }
}
```

limit()

limit(long):Stream<T>

Retorna un Stream cuyo tamaño no es mayor al número pasado por parámetro. Los elementos son cortados hasta ese tamaño.

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class App {
    public static void main(String... args) {
        List<String> ciudades = Arrays.asList("Lugo", "Coruña", "Zaragoza", "Coruña","Coruña");
        Stream<String> stream = ciudades.stream().limit(3);
        List<String> resultado=stream.collect(Collectors.toList());
        System.out.println(resultado);
    }
}
```

skip()

skip(long):Stream<T>

Retorna un Stream que descarta los primeros N elementos, donde N es el número pasado por parámetro. Si el Stream contiene menos elementos que N, entonces retorna un Stream vacío.

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class App {
    public static void main(String... args) {
        List<String> ciudades = Arrays.asList("Lugo", "Coruña", "Zaragoza", "Coruña","Coruña");
        Stream<String> stream = ciudades.stream().skip(3);
        List<String> resultado=stream.collect(Collectors.toList());
        System.out.println(resultado);
    }
}
```

MAPEO DE DATOS

Otra operación intermedia típica es el mapeo de datos. Una vez hemos realizado la selección (filtro), podemos "transformar" los elementos de un Stream haciendo corresponder, es decir, mapeando, cada elemento del Stream entreante en otro transformado en el Stream Saliente. La transformación puede no ser sólo de valor, puede ser también de tipo. Para lograrlo usamos alguno de los siguientes métodos:

- map(Function<T, R>): Stream<R>
- mapToDouble(ToDoubleFunction<T>): DoubleStream
- mapToInt(ToIntFunction<T>): IntStream
- mapToLong(ToLongFunction<T>): LongStream

map(Function<T, R>): Stream<R>

Retorna un Stream que contiene el resultado de aplicar la función pasada por parámetro a todos los elementos del Stream. Transforma los elementos del tipo T al tipo R, aunque como en el ejemplo de abajo puede ocurrir que T y R sean del mismo tipo, concretamente en el ejemplo de abajo en el map entre un string y sale otro String, el mismo en mayúsculas.

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class App {
    public static void main(String... args) {
        List<String> ciudades = Arrays.asList("Lugo", "Coruña", "Zaragoza", "Cádiz","Córdoba","Cartagena");
```

```
        //Stream<String> stream = ciudades.stream().filter(s->s.startsWith("C")).map(s->s.toUpperCase());
        //más fácil todavía con sintaxis de referencia a métodos
        Stream<String> stream = ciudades.stream().filter(s->s.startsWith("C")).map(String::toUpperCase);
        List<String> resultado=stream.collect(Collectors.toList());
        System.out.println(resultado);
    }
}
```

En los siguientes métodos de mapeo que vamos a ver se ejecutan sobre un objeto de tipo Stream pero sale otro tipo de objeto que es una variante de Stream como IntStream, DoubleStream, LongStream, ...Estos interfaces tienen otros métodos propios de ellos lo que justifica el mapeo, por ejemplo IntStream tiene un método sum() que suma los enteros.

Ejemplos con IntStream

Para usar el sum() de IntStream primero tengo que pasar el Stream<T> a IntStream para eso utilizo mapToInt(). En el siguiente ejemplo mapeo cada Transaction de un Stream<Transaction> a un Integer obteniendo finalmente un IntStream

```
import java.util.Arrays;
import java.util.List;

class Transaction {

    public static final int GROCERY = 20;
    private int Id;
    private int value;
    private int type;

    public Transaction(int Id, int value, int type) {
        this.Id = Id;
        this.value = value;
        this.type = type;
    }

    public int getId() {
        return Id;
    }

    public void setId(int Id) {
        this.Id = Id;
    }

    public int getValue() {
        return value;
    }

    public void setValue(int value) {
        this.value = value;
    }

    public int getType() {
        return type;
    }

    public void setType(int type) {
        this.type = type;
    }

}

class App {

    public static void main(String[] args) {
```

```
        Transaction[] arrayTransacciones = {new Transaction(1, 100, 33), new Transaction(3, 80, 20), new Transaction(6, 120, 20), new Transaction(7, 40, 35), new Transaction(10, 50, 20)};
        List<Transaction> transactions = Arrays.asList(arrayTransacciones);
        int suma
                = transactions.stream()
                        .filter(t -> t.getType() == Transaction.GROCERY)
                        .mapToInt(Transaction::getValue)
                        .sum();
        System.out.println("suma: " + suma);

    }

}
```

mapToInt(Transaction::getValue) escrito como lambda mapToInt(t->t.getValue())

Ya usamos previamente el método static range(). Observa que es de la clase IntStream. Si consultas el API observas que no es una clase genérica y que es una secuencia de tipos primitivos int, no de Objetos.

```
import java.util.stream.IntStream;
public class App {
    public static void main(String[] args) {
        IntStream it = IntStream.range(0, 10);
        it.forEach(System.out::println);
    }
}
```

el forEach() con lambda

```
it.forEach(num->System.out.println(num));
```

IntStream trabaja con tipos primitivos de forma que num es un int no Integer

flatMap()

No viene el tutorial de oracle pero se usa mucho

flatMap() es la combinación de un mapear + aplanar. Primero aplica un mapeo pero genera una estructura compuesta que pasa a aplanarse extrayendo automáticamente sus elementos y generando una estructura más simple.

Antes de ver flatMap veamos uno ejemplos de que es aplanar

Tengo una lista de listas, o visto de otra manera, una lista con sublistas

[[2, 3, 5], [7, 11, 13], [17, 19, 23]]

Aplanarla significa obtener una lista sin sublistas

[2, 3, 5, 7, 11, 13, 17, 19, 23]

```
import java.util.*;
import java.util.stream.Collectors;

class App {
  // Driver code
  public static void main(String[] args) {
      // Creating a list of Prime Numbers
      List<Integer> PrimeNumbers = Arrays.asList(5, 7, 11, 13);

      // Creating a list of Odd Numbers
      List<Integer> OddNumbers = Arrays.asList(1, 3, 5);

      // Creating a list of Even Numbers
      List<Integer> EvenNumbers = Arrays.asList(2, 4, 6, 8);

      List<List<Integer>> listOfListofInts = Arrays.asList(PrimeNumbers, OddNumbers, EvenNumbers);

      System.out.println("The Structure before flattening is : " +
```

```
            listOfListofInts);

        // Using flatMap for transformating and flattening.
        List<Integer> listofInts = listOfListofInts.stream()
                .flatMap(list -> list.stream()) //con el método stream() aplano porque extraigo los enteros de la lista
                .collect(Collectors.toList());

        System.out.println("The Structure after flattening is : " +
                listofInts);
    }
}
```

Es decir el flatMap añade al map una iteración automática e interna sobre una lista para aplanarla y devolver enteros de forma que esos enteros que se van extrayendo se van incorporando al stream de salida que es es de tipo Integer

ORDENACIÓN

no viene en la página web que seguimos pero sería otra operación intermedia típica. Hay muchas posibilidades lo más básico es utilizar sorted()

sorted()

Sin parámetros se pueden utilizar cuando los objetos del Stream implementan Comparable como por ejemplo los de la clase String

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class App {
    public static void main(String... args) {
        List<String> ciudades = Arrays.asList("Lugo", "Coruña", "Zaragoza", "Cádiz","Córdoba","Cartagena");
                                                                    Stream<String> stream =
ciudades.stream().filter(s->s.startsWith("C")).map(String::toUpperCase).sorted();
        List<String> resultado=stream.collect(Collectors.toList());
        System.out.println(resultado);
    }
}
```

si los objetos no son comparables o no nos interesa el orden de comparable podemos pasarle un comparador a sorted().

Ejemplo ordenar por longitud

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class App {
    public static void main(String... args) {
        List<String> ciudades = Arrays.asList("Lugo", "Coruña", "Zaragoza", "Cádiz","Córdoba","Cartagena");
        Stream<String> stream = ciudades.stream().filter(s->s.startsWith("C")).map(String::toUpperCase).sorted((a,b)->a.length()>b.length()?-1:1);
        List<String> resultado=stream.collect(Collectors.toList());
        System.out.println(resultado);
    }
}
```

Cómo "ordenar" es un concepto importante para el procesamiento masivo de datos, hay un montón de cosas al respecto, pero con lo anterior nos llega por el momento.

OPERACIONES TERMINALES

El último paso son las operaciones terminales, las cuales provocan que todas las operaciones intermedias sean ejecutadas y pueden catalogarse así:

- *Operaciones terminales cuyo propósito es el consumo de los elementos del Stream, por ejemplo: +foreach(Consumer<T>):void*

- *Operaciones terminales que permiten obtener datos del Stream como: conteo, mínimo, máximo, búsqueda, y en general reducir el Stream a un valor.*
- *Operaciones terminales que permiten recolectar los elementos de un Stream en estructuras mutables como por ejemplo listas.*

En los ejemplos que haremos ahora como ya tenemos totalmente claro que hay una cadena de operaciones, vamos a escribir dicha cadena en varias líneas para no comprometer la legibilidad.

OPERACIONES TERMINALES DE CONSUMO

foreach()

ya lo conocemos, pues lo usamos sobre listas y mapas. Como ves también se puede usar sobre un Stream. Observa también que foreach no devuelve otro Stream, simplemente consume, por eso cambiamos un poco la estructura.

```
import java.util.Arrays;
import java.util.List;
public class App {
    public static void main(String... args) {
                List<String> ciudades = Arrays.asList("Lugo", "Coruña", "Zaragoza", "Cádiz", "Córdoba",
"Cartagena");
        ciudades.stream()
                .filter(s -> s.startsWith("C"))
                .map(String::toUpperCase)
                .sorted((a, b) -> a.length() > b.length() ? -1 : 1)
                .forEach(System.out::println);
    }
}
```

OPERACIONES TERMINALES QUE OBTIENE UN VALOR DE UN STREAM

count():long

Retorna la cantidad de elementos en el Stream

```
import java.util.Arrays;
import java.util.List;
public class App {
    public static void main(String... args) {
                List<String> ciudades = Arrays.asList("Lugo", "Coruña", "Zaragoza", "Cádiz", "Córdoba",
"Cartagena");
        Long cuantasCiudades=ciudades.stream()
                .filter(s -> s.startsWith("C"))
                .count();
        System.out.println("Ciudades que empiezan por C "+ cuantasCiudades);
    }
}
```

allMatch(Predicate<T>):boolean

Verifica si todos los elementos del Stream satisfacen el predicado pasado por parámetro. Si durante la verificación alguno no lo cumple entonces se detiene la verificación y retorna falso, es decir, no requiere procesar todo el Stream para producir el resultado.

```
import java.util.Arrays;
import java.util.List;
public class App {
    public static void main(String... args) {
                List<String> ciudades = Arrays.asList("Lugo", "Coruña", "Zaragoza", "Cádiz", "Córdoba",
"Cartagena");
        boolean masDe5=ciudades.stream()
                .filter(s -> s.startsWith("C"))
                .allMatch(s->s.length()>5);
```

```
        System.out.println("El nombre de todas la ciudades que empiezan por C tienen más de 5 caracteres?: "+ masDe5);
    }
}
```

otros similares son:

- anyMatch(Predicate<T>):boolean
- y noneMatch(Predicate<T>):boolean.

Por ejemplo:

```
import java.util.Arrays;
import java.util.List;
public class App {
    public static void main(String... args) {
                List<String> ciudades = Arrays.asList("Lugo", "Coruña", "Zaragoza", "Cádiz", "Córdoba", "Cartagena");
        boolean masDe5=ciudades.stream()
                .filter(s -> s.startsWith("C"))
                .anyMatch(s->s.length()>5);
            System.out.println("El nombre de alguna de las ciudades que empiezan por C tienen más de 5 caracteres?: "+ masDe5);
    }
}
```

Mini explicación de la clase Optional.

Algunos métodos de Stream quieren devolver un valor en lugar de un Stream. Muchos de estos métodos en lugar de devolver un valor directamente necesitan devolver dicho valor encapsulado dentro de un objeto Optional. Esto es debido a que dicho valor podría valer null.
En una cadena de operaciones sería muy engorroso verificar si algo es null ya que suele hacerse en base a la sentencia if o similar. Si no se hace, podría ocurrir una *nullpointerexception*.
Por ejemplo el siguiente ejemplo genera *nullpointerexception*

```
import java.util.Arrays;
import java.util.List;

class App {

    public static void main(String[] args) {
        List<String> lista = Arrays.asList("Hola", "Mundo", null, "!");
        lista.stream()
                .filter(s -> s.startsWith("H"))
                .forEach(System.out::println);

    }
}
```

En este caso tiene una solución poco aparatosa incluyendo condición en la lambda de filter, pero esto no gusta en programación funcional porque es incluir algo "imperativo" en el estilo "declarativo"

```
import java.util.Arrays;
import java.util.List;

class App {
    public static void main(String[] args) {
        List<String> lista = Arrays.asList("Hola", "Mundo", null, "!");
        lista.stream()
                .filter(s -> s!=null && s.startsWith("H"))
```

```
            .forEach(System.out::println);
    }
}
```

Usar Optional para contemplar la posibilidad de nulos en programación declarativa.
Con Streams si una operación necesita devolver un valor de tipo T, y ese valor puede ser nulo hacemos que el valor devuelto sea un Optional<T>. Por ejemplo el método findAny() que veremos a continuación podría no encontrar ningún valor entonces devolvería nulo pero lo va a hacer encapsulando este valor en un objeto Optional para evitar posibles nullpointerexcepction.

findAny():Optional<T>
Retorna algún elemento del Stream. Nótese que el retorno es de tipo Optional (por si acaso el Stream se encuentra vacío).

```
import java.util.Arrays;
import java.util.List;
import java.util.Optional;
public class App {
    public static void main(String... args) {

        List<String> ciudades = Arrays.asList("Lugo",  "Zaragoza");
       Optional<String> unoCualquiera=ciudades.stream()
             .filter(s -> s.startsWith("C"))
             .findAny();
       System.out.println("El nombre de una  ciudad que empiezan por C: "+ unoCualquiera);
    }
}
```

findAny() en lugar de devolver un null, devuelve un Optional que encapsua dicho nudo

reduce(BinaryOperator<T>):Optional<T>

La forma genérica de "reducir" un Stream a un valor es con el método reduce() que tiene como parámetro una lambda que le indica con un BiFunction el criterio de reducción.

Observa en el siguiente ejemplo que no tenemos que iterar para averiguar el máximo lo hace internamente reduce(). Observa también que usamos el nuevo método estático *+Integer.max(int, int):int*

```
import java.util.Arrays;
import java.util.List;
import java.util.Optional;
public class App {
    public static void main(String... args) {
        List<Integer> numeros = Arrays.asList(1,2,3,14,5,6,7,8,9,13,4);
        Optional<Integer> elMayorPar=numeros.stream()
              .filter(n -> n%2==0)
              .reduce( (a,b)->Integer.max(a, b));
        System.out.println("El mayor numero Par: "+ elMayorPar);
    }
}
```

si utilizamos una lista de impares a reduce() llegará un Stream vacío pero como devuelve algo optional funciona correctamente, pruébalo por ejemplo con:

```
List<Integer> numeros = Arrays.asList(1,3,5,7,9,13);
```

OPERACIONES TERMINALES DE RECOLECCIÓN CON Collect()

recolectar es una operación de terminación que genera a partir de los datos del Stream una colección como una List o un Map. Se hace con el método Collect()

Collect()

Al método Collect() se le pasa a su vez como parámetro métodos static que son utilidades de Collectors.

Merece la pena que consultes en el API el aspecto de Collectors para que veas que hay un montón de métodos static. Ahora nos fijamos en uno sencillo que ya utilizamos *toList()*

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
public class App {
    public static void main(String... args) {
        List<Integer> numeros = Arrays.asList(1,3,5,7,2,9,13,44);
        List<Integer> pares=numeros.stream()App
                .filter(n -> n%2==0)
                .collect(Collectors.toList());
        System.out.println(pares);
    }
}
```

Para no tener que poner el nombre de la clase con el método y ser conciso escribiendo se usa normalmente el siguiente import

```
Import  static java.util.stream.Collectors.*;
```

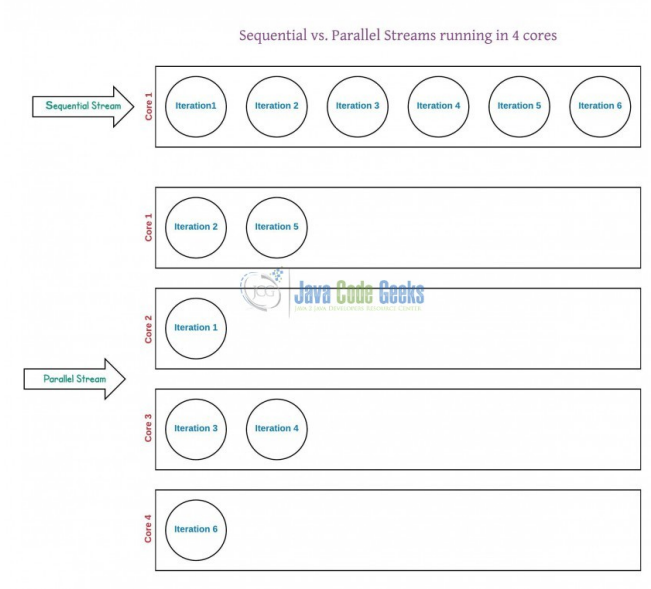
con esto podemos escribir simplemente toList(). Pero recuerda que el * puede hacer que usemos una clase que no queremos cuando usamos varias APIs que tienen nombre de clase repetido.

```
import java.util.Arrays;
import java.util.List;
import static java.util.stream.Collectors.*;
public class App {
    public static void main(String... args) {
        List<Integer> numeros = Arrays.asList(1,3,5,7,2,9,13,44);
        List<Integer> pares=numeros.stream()
                .filter(n -> n%2==0)
                .collect(toList());
        System.out.println(pares);
    }
}
```

STREAMS PARALELOS

El api Streams nos ofrece procesamiento concurrente sin necesidad de tener que lidiar directamente con hilos. El api lo hace por nosotros. Al trabajar con grandes masas de datos normalmente ocurren internamente muchas iteraciones y en muchos casos se podrían repartir dichas iteraciones entre los diversos núcleos del procesador sin que tenga efectos este reparto en el resultado final del cálculo, simplemente todo es más rápido. Ojo porque hay procesos que precisan secuencialidad, recuerda la necesidad de sincronizar métodos y colecciones al trabajar con hilos en muchos casos.

Para generar streams paralelos hay diversos métodos. Nosotros nos limitamos a ver un ejemplo con el método parallelStream()



En el siguiente ejemplo, el origen de datos del stream es un arraylist de 20 millones de empleados(si tu equipo tiene poca memoria reduce a 10 millones por ejemplo) Luego pasamos a calcular cuantos empleados ganan más de 15000 y a continuación para observar mejor la brecha entre secuencial y paralelo hacemos otro filtrado para quedarnos con los empleados con nombre mayor que "C", lo imprimimos también imprimimos cuánto tiempo le llevó al stream hacer el cálculo. Hacemos esto con un stream secuencial generado por stream() y otro paralelo generado por parallelStream(). Si el volumen de datos y la cantidad de operaciones no es muy grande podría ocurrir que en diversas ejecuciones el resultado sea favorable para el stream secuencial, pero en principio se debería apreciar que el stream paralelo es más veloz.

```
import java.util.ArrayList;
import java.util.List;
class Empleado {
  String Nombre;
  int sueldo;
  public Empleado(String nombre, int sueldo) {
     Nombre = nombre;
     this.sueldo = sueldo;
  }

}

public class App{
  public static void main(String[] args) {

     long t1, t2;
     List<Empleado> eList = new ArrayList<Empleado>();
     //Creamos  empleados, tiene datos repetidos pero es igual para el ejemplo
     for(int i=0; i<20_000_000; i++) {
        eList.add(new Empleado("A", 20000));
        eList.add(new Empleado("B", 3000));
        eList.add(new Empleado("C", 15002));
        eList.add(new Empleado("D", 7856));
        eList.add(new Empleado("E", 200));
        eList.add(new Empleado("F", 50000));
     }

     // con stream secuencial contamos empleados con sueldo >15000
     t1 = System.currentTimeMillis();
          System.out.println("número de empleados con sueldo > 15000= " + eList.stream().filter(e -> e.sueldo > 15000).filter(e->e.Nombre.compareTo("C")>0).count());
     t2 = System.currentTimeMillis();
     System.out.println("tiempo de calculo del stream secuencial= " + (t2-t1) + "\n");
     //lo mismo con stream paralelo evidentemente también serán la misma cantidad de  empleados con sueldo > 15000
     t1 = System.currentTimeMillis();
       System.out.println("número de empleados con sueldo > 15000= " + eList.parallelStream().filter(e -> e.sueldo > 15000).filter(e->e.Nombre.compareTo("C")>0).count());
```

```
        t2 = System.currentTimeMillis();
        System.out.println("tiempo de calculo del stream paralelo= " + (t2-t1));


    }
}
```

ESTILO DE PROGRAMACIÓN FUNCIONAL

Observa pues, que la programación funcional nos permite descomponer la solución de un problema en subproblemas y que cada subproblema se resuelve con una función. En el estilo de programación funcional, el grado de descomposición debe ser tal que las funciones son sencillas, habitualmente de una instrucción, no contienen ifs ni mucho menos while ya que las iteraciones son un mecanismo interno de los métodos de los streams.

Observa que gracias a las expresiones lambda le doy a la programación java ese "aire declarativo" tan deseado indicando lo que quiero sin indicar los pasos para conseguirlo. Las "operaciones" de los streams por su parte manejan internamente hilos, inmutabilidad para generar seguridad y eficiencia, y por otro las interacciones y condiciones internas que contribuyen junto a las lambda a generar el estilo declarativo.

Ejercicio U8_B7_E1: Indicamos en otro boletín que una lambda puede tener un cuerpo complejo pero "no es habitual" en la programación funcional. Concretamente indicamos que es raro ver un cuerpo de una lambda con un if y mucho más con un while. El if se puede evitar si trabajamos con streams añadiendo una operación filter(). Escribe el siguiente ejemplo utilizando un Stream.

```
import java.util.ArrayList;
import java.util.List;

class App{
    public static void main(String[] args){
        List<String> items = new ArrayList<>();
                items.add("A");
                items.add("B");
                items.add("C");
                items.add("D");
                items.add("E");

        items.forEach((item) -> {
            if(item.compareTo("B")>0)
                System.out.println(item);});

    }
}
```

Ejercicio U8_B7_E2: El siguiente ejemplo genera un Stream y filtra sus elementos quedándose con lo que comienzan por "c". Se pide añadir al conjunto de operaciones el método count() que devuelve en un long el número de elementos del stream.

```
import java.util.Arrays;
import java.util.List;

class App {

    public static void main(String[] args) {
        List<String> myList
                = Arrays.asList("a1", "a2", "b1", "c2", "c1");
        myList
                .stream()
                .filter(s -> s.startsWith("c"));

    }
}
```

Ejercicio U8_B7_E3: Añadir código para que la lista filtrados contenga todos los artículos con precio mayor que 30.0

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

class Articulo {

   String codigo;
   double pvp;

   public Articulo(String codigo, double pvp) {
      this.codigo = codigo;
      this.pvp = pvp;
   }

   public String getCodigo() {
      return codigo;
   }

   public Double getPvp() {
      return pvp;
   }

   @Override
   public String toString() {
      return "Articulo{" + "codigo=" + codigo + ", pvp=" + pvp + '}';
   }


}

public class App {

   public static void main(String[] args) {
            Articulo[] articulos = {new Articulo("A1", 10.0), new Articulo("A2", 30.5), new Articulo("B1", 30.0), new
Articulo("B2", 100.0), new Articulo("c1", 66.5),};
      List<Articulo> listaArticulos = Arrays.asList(articulos);
      List<Articulo> filtrados;
      //hacer tubería aquí para obtener filtrados
      System.out.println(filtrados);

   }
}
```

```
run:
[Articulo{codigo=A2, pvp=30.5}, Articulo{codigo=B2, pvp=100.0}, Articulo{codigo=c1, pvp=66.5}]
```

Ejercicio U8_B7_E4:Añade a la solución anterior que la lista de filtrados esté ordenada por pvp ascendentemente.

Ejercicio U8_B7_E5: De nuevo con la clase Articulo. Queremos sumar el pvp de todos los importes cuyo código empieza por "B". Para ello debes utilizar las funciones stream(), filter(),mapToDouble() y sum(). Además consultando API intenta explicar porque para este ejercicio necesito utilizar mapToDouble() en lugar de map()

```
public class App {

   public static void main(String[] args) {
```

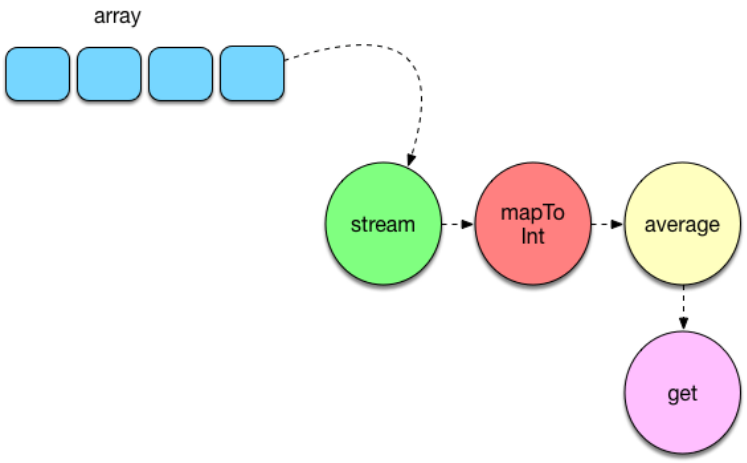
```
            Articulo[] articulos = {new Articulo("A1", 10.0), new Articulo("A2", 30.5), new Articulo("B1", 30.0), new Articulo("B2", 100.0), new Articulo("c1", 66.5),};
        List<Articulo> listaArticulos = Arrays.asList(articulos);
        double total = listaArticulos
                .stream()
                //acabar esto
        System.out.println(total);

    }
}
```

Ejercicio U8_B7_E6: El siguiente código calcula la media de las longitudes de una lista de cadenas de la forma tradicional.

```
class App{
    public static void main(String[] args) {
        String[] cadenas= {"a","ab","abc"};
        double total=0;
        double media=0;
        for (int i=0;i<cadenas.length;i++) {
            total+=cadenas[i].length();
        }
        media=total/cadenas.length;
        System.out.println(media);
    }
}
```

Se pide que lo escribas basándote en Streams. Si consultas la clase Arrays() observarás que hay un stream() para obtener un stream de un array. Usaremos una función average() que como devuelve un OptionalDouble utilizamos la función getAsDouble() para obtener su double (en gráfico solo pone get)



Ejercicio U8_B7_E7: Recuerda el problema contando en al arena

http://www.aceptaelreto.com/problem/statement.php?id=369

escríbe una solución sin usar if ni bucles. En la plataforma de acepta el reto ino podrás probar tu solución funcional porque usa un JDK 7!