Unidad 9. Java 8. Arrays y la API Stream

Contenido

1 Actividad inicial

Preguntamos a los **alumnos** si han **oído hablar de la API Stream o si la han usado alguna vez.**

Probamos el siguiente ejemplo en clase de comparativa de velocidades

Pregunta: porque es más rápido el último ejemplo

```
package comparativavelocidad;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.concurrent.TimeUnit;

public class ParalelismoStreamsComparativa {
```

```
        private static List<Integer> buildIntRange() {
            List<Integer> numbers = new ArrayList<>(5);
            for (int i = 0; i < 6000 ;i++)
              numbers.add(i);
            return Collections.unmodifiableList(numbers);
          }

        public static void main(String[] args) {
            List<Integer> source = buildIntRange();


            long start = System.currentTimeMillis();
            for (int i = 0; i < source.size(); i++) {
              try {
                TimeUnit.MILLISECONDS.sleep(1);
              } catch (InterruptedException e) {
                e.printStackTrace();
              }
            }
            System.out.println("Modo tradicional: " +
(System.currentTimeMillis() - start) + "ms");


            start = System.currentTimeMillis();
            source.stream().forEach(r -> {
              try {
                TimeUnit.MILLISECONDS.sleep(1);
              } catch (Exception e) {
                e.printStackTrace();
              }
            });
            System.out.println("stream Con procesado secuencial: " +
(System.currentTimeMillis() - start) + "ms");


            start = System.currentTimeMillis();
            source.parallelStream().forEach(r -> {
              try {
                TimeUnit.MILLISECONDS.sleep(1);
              } catch (Exception e) {
                e.printStackTrace();
              }
            });
            System.out.println("parallelStream :, Con procesado paralelo " +
(System.currentTimeMillis() - start) + "ms");

        }

}
```

2 Introducción

En este tema **vamos a trabajar con estructuras de datos más complejas** como son los arrays. En programación **declarativa los arrays se tratan habitualmente con bucles** para recorrerlos, como habréis hecho en los contenidos Scorm del

tema 6. En programación funcional vamos a favorecer como vimos en el tema anterior otro estilo de resolución de problemas y algoritmos:

1. **Recursividad sobre bucles:** vamos a intentar que nuestros programas hagan los **recorridos de los bucles de manera recursiva**, o con Streams. Recuerdo que recursividad es llamarse a si mismo.

2. **Inmutabilidad:** intentaremos que **las estructuras de datos que modificamos no cambien de variable**, **no haya asignaciones**, siempre y cuando el algoritmo que aplicamos exija el tratamiento de datos, modificaciones puntuales, de objetos dentro del array, pero que el array en si mismo,su referencia, su variable no cambie.

3. **Sin estado y sin efectos secundarios**: usaremos **funciones y expresiones lambda para resolver nuestros problemas** todo el rato, sin modificación del estado de objetos, a no ser que sea necesario.

4. **Procesado de datos Stream**: es una **nueva técnica de programar** que nos permite **la programación secuencial o paralela** de **arrays** y **colecciones** de datos.

3 Programación Stream

El **procesamiento de secuencias o Streams** es **un paradigma de programación informática**, **equivalente a la programación de flujo de datos**, el procesamiento **de flujos de eventos y la programación reactiva**, que permite que **algunas aplicaciones exploten más fácilmente una forma limitada de procesamiento paralelo**. Estas aplicaciones **pueden utilizar varias unidades de cálculo, como la unidad de punto flotante** en una **unidad de procesamiento de gráficos** (tarjeta gráfica), sin administrar explícitamente la **asignación, sincronización o comunicación** entre esas unidades.

El paradigma de **procesamiento de streams simplifica el software y el hardware paralelos al restringir el cálculo paralelo** que se puede realizar. Dada una secuencia de datos (una secuencia), se **aplica una serie de operaciones (funciones de kernel**) a cada **elemento de la secuencia**. Las **funciones del núcleo se canalizan generalmente**, y se **intenta la reutilización óptima de la memoria local** en el procesador, con el fin de **minimizar la pérdida de ancho de banda**, asociada con la **interacción de memoria externa**. **La transmisión uniforme**, donde **se aplica una función del kernel a todos los elementos de la secuencia**, es típica.

Dado que **las abstracciones del kernel y de la secuencia exponen dependencias de datos**, las **herramientas del compilador pueden automatizar y optimizar** completamente las tareas de **administración en el**

procesador. El **hardware de procesamiento de secuencias** puede utilizar el marcador, por ejemplo, para **iniciar un acceso directo a la memoria (DMA)** cuando **se conocen las dependencias**. La **eliminación de la administración manual de DMA** reduce la complejidad del software y una **eliminación asociada para la E/S almacenada en caché** de hardware, **reduce la extensión del área de datos** que debe estar **implicada con el servicio** por parte de **unidades computacionales especializadas**, como las **unidades lógicas aritméticas**.

En resumén, **se ha optimizado los procesadores con multiples nucleos** para permitir el **tratamiento de secuencias de objeto** de manera paralela. Se han **introducido nuevas operaciones en los procesadores** para permitir este **tipo de nueva programación**, con **instrucciones específicas para paralelizar las secuencias de objetos** (Streams) y las **operaciones de manera paralela** que se aplican sobre **cada objeto**. Se ha **optimizado los accesos a memoria** para el uso Streams igualmente.

Un **Stream es una secuencia de objetos de una clase**. Por ejemplo una **secuencia de Empleados**, una secuencia de Fechas, Secuencia de Decimales, una **secuencia de cualquier objeto** para el **que podamos construir su clase en Java**.

Los **primeros referentes en programación Stream**, los **tenemos durante la década de los 80s** cuando **se exploró el procesamiento de flujos de datos** dentro de la programación de flujo de datos. Un ejemplo es el **lenguaje SISAL**.

4 La API Stream para el manejo de arrays en java

La API de Java Stream **proporciona un enfoque funcional** para **procesar colecciones de objetos**. **La API Java Stream se agregó en Java 8** junto con varias otras **características de programación funcional**. Este **tema de Java Stream** explicará cómo funcionan estas secuencias funcionales y cómo se utilizan. Nos va a permitir aprovechar **las nuevas carácteristicas de los procesadores**, para **el manejo y manipulación de secuencias de objetos**, Streams de objetos.

La API Stream está compuesta por los siguientes interfaces y clases

Interface	Description
BaseStream<T,S **extends** BaseStream<T,S >>	Interfaz base para Streams, que son secuencias de elementos que admiten operaciones de agregado secuenciales y paralelas.

Collector<**T,A,R**>	Una **operación de reducción mutable** que acumula elementos de entrada en un contenedor de resultados mutable, transformando opcionalmente el resultado acumulado en una representación final después de que se hayan procesado todos los elementos de entrada.
DoubleStream	Secuencia de elementos primitivos de doble valor que admiten operaciones de agregado secuenciales y paralelas.
DoubleStream.Builder	Un Builder para un **DoubleStream.**
IntStream	Secuencia de elementos primitivos con valores int que admiten operaciones de agregado secuenciales y paralelas.
IntStream.Builder	Un Builder para un IntStream.
LongStream	Secuencia de elementos primitivos de valor largo que admiten operaciones de agregado secuenciales y paralelas.
Clases	**Descripcion**
Collectors	Implementaciones de Collector que implementan diversas operaciones de reducción útiles, como la acumulación de elementos en colecciones, la integración de elementos según diversos criterios, etc.
StreamSupport	Métodos de utilidad de bajo nivel para crear y manipular secuencias.
Collector.Characteristics	Características que indican las propiedades de un collector, que se pueden utilizar para optimizar las implementaciones de reducción.

La **API java Stream no está relacionada** con **Java InputStream** y **Java OutputStream de Java IO**. **InputStream y OutputStream** están relacionados con **secuencias de bytes**. La **API de Java Stream** es para procesar **secuencias de objetos**, sin bytes.

4.1 El operador ::

Hay **operadores nuevos introducidos con java 8**, como el operador :: Vamos a **describirlos con breves ejemplos**. En primer lugar el operador:: . Este nos va a servir **para simplificar nuestras expresiones lambda** cuando usamos **métodos estáticos**. Lo hemos usado varias veces en el ejemplo anterior con el System.out.println.

```
Stream.of(miArray)
                  .filter(s -> s.startsWith("c"))
                     .map(String::toUpperCase)
                     .sorted()
                     .forEach(System.out::println);
```

System.out.println es un método estático que llamamos. Lo podemos hacer igualmente **con cualquiera de nuestros métodos estáticos** que creemos para nuestras clases. O con **métodos estáticos de otras clases**. System.out:println es equivalente a la expresión lambda s->System.out.println(s). Podéis probar a sustituirlo y funcionará exactamente igual.

```
Stream.of(miArray)
                  .filter(s -> s.startsWith("c"))
                     .map(String::toUpperCase)
                     .sorted()
                     .forEach(s->System.out.println(s));
```

En el siguiente caso **usamos el Operador :: para transformar los enteros** en **cadenas** con el método estático **valueOf** de la clase **String.** Es equivalente a la expresión lambda s->String.valueOf(s).

OperadorDosPuntos.java

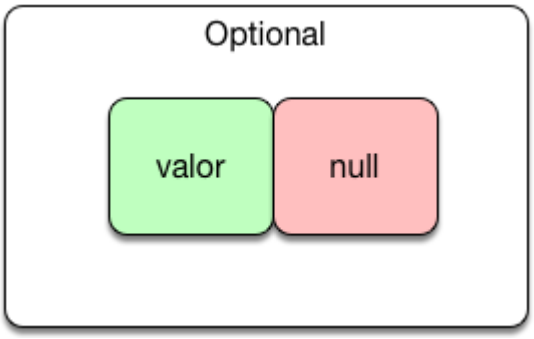
```
import java.util.Arrays;
import java.util.Random;
import java.util.stream.Stream;

public class OperadorDosPuntos {


        public static void main(String[] args) {
        System.out.println("\nFiltramos por números impares y creamos un nuevo
array ordenado");
         int miArray[] =  {1, 2, 3,4,5,6,7,8,9,10};

         Stream.of(miArray).forEach(String::valueOf);




        }

}
```

Mas adelante en los apuntes, cuando expliquemos el método map() veremos que también hay otra manera de usar el operador ::.

4.2 La clase Optional. Practica guiada clase Optional

Class Optional<T> es una clase muy útil en Java. Nos permite definir una clase que tiene un valor o no.
Objeto contenedor **que puede o no contener un valor distinto de null**. Si hay

un valor presente, `isPresent()` devolverá `true` y `get()` devolverá el valor.
Se proporcionan métodos adicionales que dependen de la presencia o ausencia de un valor contenido, como `orElse()` (devolver un valor predeterminado si el valor no está presente) y `ifPresent()` (ejecutar un bloque de código si el valor está presente).
Se trata de una clase basada en valores; el uso de operaciones sensibles a la identidad, (incluida la igualdad de referencia == o hashCode) en instancias de `Optional` puede **tener resultados impredecibles y debe evitarse.**



La **vamos a usar en combinación con métodos terminales de la API Stream** que veremos **posteriormente**. Pero ahora **vamos a explicarlo con un ejemplo**. Es muy útil porque **nos da la posibilidad de recoger valores nulos** de nuestras operaciones lambda sin que se produzca la excepción NullPointerException.

Para **crear un Optional usamos el método estático** de la clase **Optional of**. No tiene constructor. Como parámetro recibe el objeto que queremos almacenar en el contenedor. Observar que hay que **definir un tipo genérico para Optional**.

```
Optional<Empleado> empl1=
Optional.of(new Empleado(5,"Carlos Lopez"));
```

El **método IsPresent()** , comprueba si el valor que co**ntiene el contenedor es nulo o un objeto. Si isPresent devuelve true** es que contiene un **objeto**. Sino contendrá null.

```
if(empl1.isPresent()){
```

El método **get nos va a devolver el objeto que hay dentro del contenedor Optional**. La signatura del método **es public T get().** Como veis devuelve el Tipo que hemos definido **con el tipo genético T**. Nuestro consumer lo hemos definido como tipo Empleado. El método get **devolverá el empleado almacenado**.

```
empl1.get();
```

Y el método más complejo **es ifPresent**, que recibe como **parámetro un Consumer, para ejecutar código, una expresión lambda** en caso de que **el objeto esté presente**. Esa **expresión lambda recibirá de entrada el objeto** que contiene el **Optional**, en el ejemplo **un Empleado**. Está es la signatura del método y recibe un Consumer como parámetro.

```
public void ifPresent(Consumer<? super T> consumer)
```

El **consumer que le pasamos al método ifPresent** es la expresión lambda para imprimir al empleado.

```
        empl1.ifPresent(emp->System.out.println(emp));
```

```
import java.util.Optional;

class Empleado {

        int id;
        String nombre;
        public Empleado (int id, String nombre) {

                this.id=id;
                this.nombre=nombre;

        }

        public int getId() {
                return id;
        }

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

        public String getNombre() {
                return nombre;
        }

        public void setNombre(String nombre) {
                this.nombre = nombre;
        }


        public String toString () {

        return "{id:"+ this.getId() + ", nombre:" + this.getNombre() + "}";
        }

}

public class EjemploOptional {


        public static void main(String[] args) {

        Optional<Empleado>   empl1   =   Optional.of(new   Empleado(5,"Carlos
Lopez"));



        if(empl1.isPresent()){
```

```
            Empleado emp1 = empl1.get();

            System.out.println(emp1);
        }else{
                  System.out.println("No hay nada en el Optional empl1");
        }

        empl1.ifPresent(emp->System.out.println(emp));
        empl1.ifPresent(System.out::println);

}

}
```

Como apunte extra tenemos implementaciones de la clase Option para los tipos básicos o primitivos **OptionalInt, OptionalDouble, OptionalLong**, etc. Tenéis un ejemplo en los métodos max() y min() que veremos posteriormente.

4.2.1 Practica independiente clase Optional.

1. En nuestro modelo de Trabajador en el programa principal creamos una variable de tipo clase Optional de tipo Trabajador. Creamos una vacia y una con un Objeto de tipo profesor
2. Para las dos variables comprobamos que están llenos. Si están llenos escribimos por pantalla el objeto. Si están vacíos indicamos que la variable optional contiene null.

4.3 Como funciona la API Stream

Un **Stream representa una secuencia** de **elementos y admite diferentes tipos de operaciones** para realizar **cálculos sobre esos elementos**. Lo primero que veremos es **como transformar un array a Stream** en java. Luego veremos que operaciones podemos aplicar sobre la secuencia.

4.3.1 Transformacion de arrays a Stream. Practica guiada transformar un Array a tipo Stream

Lo primero **es convertir el array a un Stream**. Java nos proporciona **diferentes maneras** de transformar un array a Stream.

Con la clase Arrays

Usaremos **el metodo stream de la clase arrays** que **transforma un array en un**

Stream. Primero transformamos el Array en un Stream. **Luego aplicamos las operaciones sobre él**. El forEach aplica un bucle for mediante una función para cada elemento del Stream. Lo veremos más adelante, es **un Interfaz Consumer** de los que hemos visto en el tema anterior.

```
String miArray[] = {"a1", "a2", "b1", "c2", "c1"};

                Stream<String> streamArray = Arrays.stream(miArray);

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

Para hacerlo totalmente funcional sólo **asignamos la variable array**. El **resto con funciones**, para hacer la **operación con inmutabilidad**. El siguiente ejemplo es igual que el anterior, pero nos quitamos la asignación en la conversión a Stream. Es más funcional. En este caso estamos aplicando más operaciones. **Las detallaremos una a una** a lo largo del tema.

```
Arrays.stream(miArray)
                .filter(s -> s.startsWith("c"))
                   .map(String::toUpperCase)
                   .sorted()
                   .forEach(System.out::println);
```

En el siguiente Código se os proporciona **el ejemplo completo**.
TransformaArrayStream.java

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

public class TransformaArrayStream {


        public static void main(String[] args) {


                String miArray[] = {"a1", "a2", "b1", "c2", "c1"};

                Stream<String> streamArray = Arrays.stream(miArray);

                streamArray
                           .forEach(System.out::println);

                System.out.println("Filtramos  quedandonos  con  los  que
empiecen por c  y los transformamos a mayusculas");

                Arrays.stream(miArray)
                .filter(s -> s.startsWith("c"))
```

```
                    .map(String::toUpperCase)
                    .sorted()
                    .forEach(System.out::println);

        }

}
```

4.3.2 **Transformando con Stream of**

Podemos usar igualmente el método estático of de la **clase Stream**. Con **Stream.of podemos transformar de Arrays** y **otros conjuntos de objetos a Stream**. El resultado es muy similar al anterior. Os dejo el ejemplo directamente. Fijaos como ahora obtenemos el Stream a partir del array con el método of : `Stream.of(miArray);`

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

public class TransformaArrayStreamof {


        public static void main(String[] args) {


                String miArray[] = {"a1", "a2", "b1", "c2", "c1"};

                Stream<String> streamArray = Stream.of(miArray);

                streamArray
                           .forEach(System.out::println);

                System.out.println("Filtramos  quedandonos  con  los  que
empiecen por c  y los transformamos a mayusculas");

                Stream.of(miArray)
                .filter(s -> s.startsWith("c"))
                   .map(String::toUpperCase)
                   .sorted()
                   .forEach(System.out::println);

        }

}
```

4.3.3 **Practica independiente. Transformar un array a Stream.**

Dado el siguiente array de apellidos:

```
String miArray[] = {"Pérez", "Sánchez", "García", "Tolbe", "Vániz"};
```

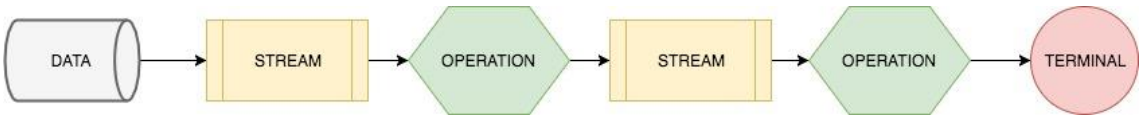
1. Transformar a tipo Stream y mostrar por pantalla con el método estático de Arrays stream().
2. Transformar a tipo Stream y mostrar por pantalla con el método estático de Stream of().

4.4 Tipos de Operaciones con Stream

Tenemos **dos tipos de operaciones sobre Streams** son **intermedias** o **terminales**. Las operaciones **intermedias devuelven un Stream** para que **podamos encadenar varias operaciones intermedias** sin usar punto y coma. Las **operaciones terminales son sin resultado** (void) o **devuelven un resultado que no es un Stream**. En el ejemplo anterior, el **filter, el map y la sort** son **operaciones intermedias**, mientras que **forEach es una operación terminal**.

Explicaremos **en el tema muchas de estas operaciones**. Para obtener una lista completa de **todas las operaciones de Stream disponibles**, podeis consultar la documentación oficial de Oracle, los creadores de Java.
https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

Una posible **ejecución de una operación completa de Stream** sería la siguiente. Vamos generando streams hasta que llegamos a una operación o varias operaciones terminales.



Esta **cadena de operaciones de Streams como se ve en el ejemplo anterior** también **se conoce como canalización o pipe** de operaciones. Se usa mucho en sistemas operativos en línea de comandos. En **Powershell de Windows** se puede usar las operaciones pipe o tubería, **en Unix** su uso es más avanzado.

La **mayoría de las operaciones de Streams aceptan algún tipo de parámetro** de **expresión lambda, una interfaz funcional** que especifica el **comportamiento exacto de la operación**. La mayoría de esas operaciones deben ser **no interferentes y sin estado.** ¿Qué significa eso?

Una función no interfiere cuando no modifica el origen de datos subyacente

de la secuencia, por ejemplo, en el ejemplo anterior **ninguna expresión lambda modifica miArray** agregando o quitando elementos de la colección.

Una **función no tiene estado cuando la ejecución de la operación es determinista**, por ejemplo, en el **ejemplo anterior ninguna expresión lambda depende de variables** o estados mutables del ámbito externo que puedan cambiar durante la ejecución. Nadie puede modificar mi Stream cuando estoy ejecutando todas estas operaciones. Porque no es un objeto al uso. No tenemos métodos para interferir en esta ejecución.

Fijaos en el **siguiente ejemplo como se aplica operaciones no terminales y terminales** a un array transformado en Stream.
`filter(n->n%2==0)` recibe un **Predicate para obtener con los números pares**. Producen **un nuevo Stream que sólo tiene números pares**. Este resultado es recogido por map.

`.map(n -> 2 * n + 1)` recibe **un Interfaz de tipo Function que transforma** el **objeto** recibido en este caso un número. Es una **operación no terminal,** de transformación.

`.Average` **calcula la media del Stream** que le **llega de map**, cada número par que hemos multiplicado por 2 y sumado 1. Reduce el Stream a un número entero. Es una operación terminal o final. Cambia **el Stream por un objeto único.**

`ifPresent` : nos indica si ha llegado **algún objeto al final de las operaciones terminales o no terminales**. Con **que llegue un solo objeto se ejecutará su interior**. Es una **operación final o terminal**. Recibe un **interfaz de tipo consumer**.

`System.out::println` es equivalente a la expresión lambda **x->System.out.println(x)**

Como veis **todas las expresiones que le pasamos a estas funciones son expresiones lambda. Los parámetros que reciben son Interfaces funcionales** como vimos con las **funciones de orden superior**. Son **funciones que reciben como parámetros funciones**.
Las **dos ultimas operaciones son terminales**. Average e ifPresent.

```
                Arrays.stream(new int[] {1, 2, 3,4,5,6,7,8,9,10})
                .filter(n->n%2==0)
                .map(n -> 2 * n + 1)
                .average()
                .ifPresent(System.out::println);  // 5.0
```

Podeis verlo completo en el siguiente ejemplo

StreamOperacionesEjemplo.java

```
import java.util.Arrays;

public class StreamOperacionesEjemplo {


        public static void main(String[] args) {

                System.out.println("Array de numeros");

                Arrays.stream(new int[] {1, 2, 3,4,5,6,7,8,9,10})
                .forEach( x->System.out.print(x+","));

                System.out.println("\nFiltramos por números pares");

                Arrays.stream(new int[] {1, 2, 3,4,5,6,7,8,9,10})
                .filter(n->n%2==0)
                .forEach( x->System.out.print(x+","));   //

                System.out.println("\nFiltramos   por   números   pares   y
multiplicamos cada numero par por dos y le sumamos 1");
                Arrays.stream(new int[] {1, 2, 3,4,5,6,7,8,9,10})
                .filter(n->n%2==0)
                .map(n -> 2 * n + 1)
                .forEach( x->System.out.print(x+","));

                System.out.println("\nFiltramos   por   números   pares   y
multiplicamos cada numero par por dos y le sumamos 1. Calculamos la media");
                Arrays.stream(new int[] {1, 2, 3,4,5,6,7,8,9,10})
                .filter(n->n%2==0)
                .map(n -> 2 * n + 1)
                .average()
                .ifPresent(System.out::println);

        }


}
```

4.5 Transformar un stream en un array

Puede darse la situación de que queramos volver a **guardar nuestros datos** tratados en el Stream en un array. Para ello tenemos el método estático de la clase Stream **toArray().**

```
Object[] toArray()
```

Podeis **observarlo en el siguiente ejemplo**. Después de que el **Stream hace todo el procesado,** transformamos **el Stream en un array** y lo guardamos en la variable **arrayImpares**. El array obtenido es del mismo tipo que los objetos que

maneja el Stream

```
        int arrayImpares[] = Arrays.stream(new int[] {1, 7,6 ,5,2,3,4,8,9,10})
                .filter(n->n%2!=0)
                .sorted()
                .toArray();
```

```
import java.util.Arrays;

public class ConvertirStreamArray {

        public static void main(String[] args) {

                System.out.println("\nFiltramos por números impares y creamos
un nuevo array ordenado");

        int arrayImpares[] = Arrays.stream(new int[] {1, 7,6 ,5,2,3,4,8,9,10})
                .filter(n->n%2==0)
                .sorted()
                .toArray();


                Arrays.stream(arrayImpares).forEach(x-
>System.out.print(x+","));
        }
}
```

4.6 Operaciones con Stream

4.6.1 Operaciones no terminales

Las operaciones de flujo no terminal de la API Java Stream son operaciones que **transforman o filtran los elementos** del Stream. Cuando **se agrega una operación que no es terminal a una secuencia**, se obtiene **una nueva Stream de nuevo como resultado.** La **nueva secuencia representa la secuencia de elementos resultantes** de la **secuencia original con la operación no terminal aplicada.** El siguiente es un ejemplo de una **operación no terminal agregada a una secuencia** - que da lugar a una nueva secuencia:

4.6.2 **filter()**

El **filtro Java Stream() se puede utilizar para filtrar elementos** de una Stream Java. El **método de filtro toma un Predicate** que se utiliza **e invoca para cada elemento de la secuencia.** Si el elemento **se va a incluir en el Stream**

resultante, el predicado **debe devolver true**. Si no se debe incluir el elemento, el predicado debe devolver false. En el siguiente Array nos quedamos con los nombres del array con longitud mayor a 4.

```
String ArrayNombres[] ={"Al", "Ankit", "Brent", "Tomas", "Alejandro", "Aitor"
,"Sarika", "amanda", "Hans", "Shivika", "Sarah", "Julius"};


Stream.of(ArrayNombres)
                .filter(x -> x.length()> 4 )
                .forEach(System.out::println);
```

4.6.3 **map()**

El **método Java Stream map()** convierte (mapea) **un elemento en otro tipo de objeto**. Por ejemplo, si **tuviera una lista de cadenas**, podría **convertir cada cadena a minúsculas, mayúsculas** o en una subcadena de la cadena original, o algo completamente diferente. En el siguiente ejemplo se puede ver.

```
Stream.of(ArrayNombres)
                .map(String::toLowerCase)
                .filter(x -> x.startsWith("a") || x.startsWith("b") ||
x.startsWith("s") )
                .forEach(System.out::println);
```

Una de las mejores funciones de map es que nos permite **transformar un stream de objetos en otro Stream de objetos diferentes**. Map es una operación no terminal porque realiza transformación de lo recibido, y de salida genera un Stream.

Vamos a **desarrollar un ejemplo para realizar esta labor**. Vamos a **convertir un array de enteros en un array de empleados.** Usaremos el método que explicaremos después **toArray para convertir el stream a un Array de objetos**. Cuando veamos Listas veréis que con el map puedo transformar a otro array al tipo que yo quiera. Pero con toArray puedo transformar mi stream a arrays **de tipo Object** solamente. Es una **limitación en el uso de Streams**, que en siguientes temas os enseñaré a resolver, y **que resolvemos para arrays con un casteo**.

En el ejemplo **tenemos dos arrays, uno de tipo Integer**, para los identificadores de **Empleado, y otro de tipo String para los nombres de Empleado**. Transformando el primero a String vamos a generar un array de tipo Object pero que como veréis aplicando polimorfismo y transformación de tipos vamos a manejar como un array de Empleados.

Lo primero los dos arrays de números y de nombres:

```
public static final Integer numeros[]=
{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24};

          public static final String nombres[] = {"ANTONIO", "MANUEL",
"JOSE", "FRANCISCO", "DAVID", "JUAN", "JOSE ANTONIO", "JAVIER", "DANIEL",
"JOSE LUIS", "FRANCISCO JAVIER",
                    "CARLOS", "JESUS", "ALEJANDRO", "FRANCISCA", "LUCIA",
"MARIA ISABEL", "MARIA JOSE", "ANTONIA", "DOLORES", "SARA",
                    "PAULA", "ELENA", "MARIA LUISA", "RAQUEL"};
```

El paso **siguiente es recorrer el array de números para generar un Stream** que nos permita crear el array de objetos. `Arrays.stream(numeros).`

Y la parte importante, el método map. Como entrada va a recibir un tipo entero, como salida un objeto de tipo Empleado. Map como veis es un interfaz de tipo Function, que hemos estudiado en temas anteriores **`map((num) -> (Empleado) new Empleado(num,nombres[num])).`** La expresión lambda recibe un num de Tipo Integer. La salida de map **es otro Stream pero de tipo Empleado** `(Empleado) new Empleado(num,nombres[num]).`

```
Object empleados[] =
                                    Arrays.stream(numeros).map((num) ->
(Empleado) new Empleado(num,nombres[num])).toArray();
```

Como **podéis observar el array resultado es de tipo Object**, Object empleados[]. Pero **lo que hay realmente guardado en cada celda del Array** es un objeto de tipo **Empleado**. Si transformo el tipo, si casteo, **el objeto que imprimo por pantalla es de tipo Empleado**.

```
         Arrays.stream(empleados).forEach((objEmpleado) ->
System.out.println(((Empleado) objEmpleado)));
```

En el **segundo Stream estoy recorriendo el array generado empleados**. Para cada objeto **de tipo Object del array en el ForEach lo imprimo**. Pero observar el casteo `(Empleado) objEmpleado.` Antes de imprimirlo, **lo transformo a tipo Empleado**. En el resultado de la ejecución **estoy imprimiendo un Empleado** no un tipo Object.

Resultado de la ejecución:

```
{id:1, nombre:MANUEL}
{id:2, nombre:JOSE}
{id:3, nombre:FRANCISCO}
{id:4, nombre:DAVID}
{id:5, nombre:JUAN}
……
```

4.6.4 **Practica guiada. Operaciones Stream y mapeo de objetos.**

MapEmpleados.java

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



        class Empleado {

                int id;
                String nombre;
                public Empleado (int id, String nombre) {

                        this.id=id;
                        this.nombre=nombre;

                }

                public int getId() {
                        return id;
                }

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

                public String getNombre() {
                        return nombre;
                }

                public void setNombre(String nombre) {
                        this.nombre = nombre;
                }


                public String toString () {

                return "{id:"+ this.getId() + ", nombre:" + this.getNombre()
+ "}";
                }

        }

        public class MapEmpleados {

        public static final Integer numeros[]=
{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24};

          public static final String nombres[] = {"ANTONIO", "MANUEL",
"JOSE", "FRANCISCO", "DAVID", "JUAN", "JOSE ANTONIO", "JAVIER", "DANIEL",
"JOSE LUIS", "FRANCISCO JAVIER",
                    "CARLOS", "JESUS", "ALEJANDRO", "FRANCISCA", "LUCIA",
"MARIA ISABEL", "MARIA JOSE", "ANTONIA", "DOLORES", "SARA",
                    "PAULA", "ELENA", "MARIA LUISA", "RAQUEL"};
```

```
        public MapEmpleados() {



        }


        public static void main(String[] args) {


                  Object empleados[] =
                                    Arrays.stream(numeros).map((num) ->
(Empleado) new Empleado(num,nombres[num])).toArray();


                   Arrays.stream(empleados).forEach((objEmpleado) ->
System.out.println(((Empleado) objEmpleado)));

                 }




}
```

Nota: IMPORTANTE. Podréis **transformar este array de objetos a Array de tipo** `Empleado` usando el **método flatMap**, el método **reduce o forEach**. Os proporciono un ejemplo con **forEach**. Estudiad el ejemplo y es **vuestra misión entended como hemos realizado** esta nueva **transformación a Array**.
Le cambiamos **el nombre a la clase interna Empleado por EmpleadoEnArray** para que se **distinga del ejemplo anterior**. Será necesario que la **variable index sea un atributo de clase** para poder usarlo dentro del Stream.
Como apuntes declaramos un array de EmpleadoEnArray y le damos la longitud de el array de números.

```
EmpleadoEnArray empleados[]= new EmpleadoEnArray[numeros.length];
```

Mapeamos como antes al objeto EmpleadoEnArray

```
.map((num) ->  new EmpleadoEnArray(num,nombres[num]))
```

Y en el **forEach** insertamos cada **empleado** generado en el **nuevo array empleados**.

```
.forEach((empleado)-> {   empleados[index]= empleado; index=index+1; });
```

MapEmpleadosArray.java

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

```
        class EmpleadoEnArray {

                int id;
                String nombre;

                public EmpleadoEnArray (int id, String nombre) {

                        this.id=id;
                        this.nombre=nombre;

                }

                public int getId() {
                        return id;
                }

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

                public String getNombre() {
                        return nombre;
                }

                public void setNombre(String nombre) {
                        this.nombre = nombre;
                }


                public String toString () {

                return "{id:"+ this.getId() + ", nombre:" + this.getNombre()
+ "}";
                }

        }

        public class MapEmpleadosArray {

        public static final Integer numeros[]=
{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24};

          public static final String nombres[] = {"ANTONIO", "MANUEL",
"JOSE", "FRANCISCO", "DAVID", "JUAN", "JOSE ANTONIO", "JAVIER", "DANIEL",
"JOSE LUIS", "FRANCISCO JAVIER",
                    "CARLOS", "JESUS", "ALEJANDRO", "FRANCISCA", "LUCIA",
"MARIA ISABEL", "MARIA JOSE", "ANTONIA", "DOLORES", "SARA",
                    "PAULA", "ELENA", "MARIA LUISA", "RAQUEL"};

          public static int index=0;

        public MapEmpleadosArray() {



        }
```

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

                index=0;
                EmpleadoEnArray empleados[]= new
EmpleadoEnArray[numeros.length];
        Arrays.stream(numeros)
.map((num) ->  new EmpleadoEnArray(num,nombres[num]))
.forEach((empleado)-> {   empleados[index]= empleado; index=index+1; });

                 Arrays.stream(empleados).forEach((empl) ->
System.out.println(( empl)));

                }




}
```

4.6.5 Practica independiente operaciones de mapeo con Streams.

Dado el siguiente array de apellidos:

```
        public static final String apellidos[] = {"García", "González",
"Rodríguez", "Fernández", "López", "Martínez", "Sánchez", "Pérez", "Gómez",
                        "Martin", "Jiménez", "Ruiz", "Hernández", "Diaz",
"Moreno", "Muñoz", "Álvarez", "Romero", "Alonso", "Gutiérrez", "Navarro",
                        "Torres", "Domínguez", "Vázquez", "Ramos", "Gil",
"Ramírez", "Serrano", "Blanco", "Molina", "Morales", "Suarez", "Ortega",
                        "Delgado", "Castro", "Ortiz", "Rubio", "Marín",
"Sanz", "Núñez", "Iglesias", "Medina", "Garrido", "Cortes", "Castillo",
"Santos"};
```

1. Añadir el atributo Apellidos en el modelo anterior a la clase Empleado
2. Modificar el ejemplo para añadir también los apellidos en la creación del array de objetos.

4.6.6 El operador :: con métodos de la clase objeto del Stream

Podemos usar el operador dos puntos para llamar a métodos de objetos de la propia clase que maneja el Stream. Lo vamos a ver con un ejemplo.
Este ejemplo es un muy parecido al anterior, pero he añadido el uso del operador :: de nueva forma

```
Arrays.stream(empleados).map(EmpleadoEnArray::getNombre).forEach(n-
>System.out.print(n+","));
```

El **método getNombre, aquí no es estático**, pero puedo llamarlo con el **operador ::**, porque **el objeto que recibe el método Map del Stream**

empleados **es de tipo EmpleadoEnArray.**

EmpleadoEnArray::getNombre **sería equivalente** a la expresión lambda e->e.getNombre().

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



        class EmpleadoEnArray {

                int id;
                String nombre;

                public EmpleadoEnArray (int id, String nombre) {

                        this.id=id;
                        this.nombre=nombre;

                }

                public int getId() {
                        return id;
                }

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

                public String getNombre() {
                        return nombre;
                }

                public void setNombre(String nombre) {
                        this.nombre = nombre;
                }


                public String toString () {

                return "{id:"+ this.getId() + ", nombre:" + this.getNombre()
+ "}";
                }

        }

        public class OperadorDosPuntosClasePropia {

        public static final Integer numeros[]=
{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24};

          public static final String nombres[] = {"ANTONIO", "MANUEL",
"JOSE", "FRANCISCO", "DAVID", "JUAN", "JOSE ANTONIO", "JAVIER", "DANIEL",
"JOSE LUIS", "FRANCISCO JAVIER",
                    "CARLOS", "JESUS", "ALEJANDRO", "FRANCISCA", "LUCIA",
"MARIA ISABEL", "MARIA JOSE", "ANTONIA", "DOLORES", "SARA",
```

```
                    "PAULA", "ELENA", "MARIA LUISA", "RAQUEL"};

          public static int index=0;



        public static void main(String[] args) {


                EmpleadoEnArray empleados[]= new
EmpleadoEnArray[numeros.length];
                                  Arrays.stream(numeros).map((num) ->  new
EmpleadoEnArray(num,nombres[num]))
                                  .forEach((empleado)-> {   empleados[index]=
empleado; index=index+1; });

                 Arrays.stream(empleados).forEach((empl) ->
System.out.println(( empl)));

Arrays.stream(empleados).map(EmpleadoEnArray::getNombre).forEach(n-
>System.out.print(n+","));
                }



}
```

4.6.7 Mapeando a un tipo básico mapToInt, mapToDouble, etc. Practica guiada de mapeo a tipo básico

Podéis **mapear un objeto a un tipo básico para obtener un Stream** de tipo **IntStream o DoubleStream** que **veremos posteriormente en los apuntes**. Es sencillo, mapeamos un objeto complejo a objetos más sencillos para luego jugar con las estadísticas. Usamos el **método mapToInt o mapToDouble** de la **Api Stream**.
Con un **sencillo ejemplo vamos a exponer estos métodos**. Nos basamos en el ejemplo anterior para que veáis como mapeamos del objeto empleado a un entero a partir de su id. Usamos **mapToInt** en este caso.

```
Arrays.stream(empleados).mapToInt((e)-> e.getId())
```

Como veis mapeamos el **empleado** a su **Id** que es **tipo básico int**, con **mapToInt**.
`MapToIntEmpleados.java`

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



                class EmpleadoArrayToInt {

                        int id;
                        String nombre;
```

```
                        private int index = 0;

                        public EmpleadoArrayToInt (int id, String nombre) {

                                this.id=id;
                                this.nombre=nombre;

                        }

                        public int getId() {
                                return id;
                        }

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

                        public String getNombre() {
                                return nombre;
                        }

                        public void setNombre(String nombre) {
                                this.nombre = nombre;
                        }


                        public String toString () {

                        return "{id:"+ this.getId() + ", nombre:" +
this.getNombre() + "}";
                        }

                }

                public class MapToIntEmpleados {

                public static final Integer numeros[]=
{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24};

                  public static final String nombres[] = {"ANTONIO",
"MANUEL", "JOSE", "FRANCISCO", "DAVID", "JUAN", "JOSE ANTONIO", "JAVIER",
"DANIEL", "JOSE LUIS", "FRANCISCO JAVIER",
                              "CARLOS", "JESUS", "ALEJANDRO", "FRANCISCA",
"LUCIA", "MARIA ISABEL", "MARIA JOSE", "ANTONIA", "DOLORES", "SARA",
                              "PAULA", "ELENA", "MARIA LUISA", "RAQUEL"};

                  public static int index=0;

                public MapToIntEmpleados() {



                }


                public static void main(String[] args) {

                        index=0;
```

```
EmpleadoArrayToInt empleados[]= new EmpleadoArrayToInt[numeros.length];
 Arrays.stream(numeros)
                 .map((num) ->  new EmpleadoArrayToInt(num,nombres[num]))
                 .forEach((empleado)-> {empleados[index]= empleado;
index=index+1; });

                         System.out.println("Mapeamos el array de objetos a
un IntStream");
        Arrays.stream(empleados).mapToInt((e)-> e.getId()).forEach((empleado)
-> System.out.println(empleado));

                         }

}
```

4.6.8 Practica independiente de mapeo a tipo básico

1. En el ejemplo anterior añadiremos una línea para mapear de Stream Empleados a Stream de Strings
2. Se mostrará por pantalla los nombres de los Empleados del Stream mapeado.

4.6.9 flatMap()

Los **métodos flatMap() de Java Stream** dividen **un único elemento en varios elementos**. La idea es que **"aplanar" cada elemento de una estructura compleja que consta de varios elementos internos**, a **un Stream "plano"** que consta sólo un tipo de elementos internos.

Por ejemplo, **imagina que tienes un objeto con objetos anidados** (objetos secundarios). A continuación, **puedes asignar ese objeto en un Stream "plano"** que consta de sí **mismo más sus objetos anidados** - o solo los **objetos anidados**. En el siguiente ejemplo **vamos a transformar una cadena en un array de cadenas**, separando cada palabra y agregandola a otro **array de cadenas, pero que contiene palabras.**

En el siguiente ejemplo vamos a convertir un string de palabras en un array de strings.

```
String[] frases =
{"En el siguiente ejemplo vamos a transformar una cadena en un array de cadenas"
,"separando cada palabra ",
"y agregandola a un array de Strings"};



                Stream.of(frases).
                flatMap((valor) -> {
```

```
                    String[] palabras = valor.split(" ");
                    return (Stream<String>) Arrays.stream(palabras);
                })
                .forEach((valor) -> System.out.print(valor+","));
```

Fijaos en la function flatMap. Recibe un interfaz funcional que transforma un valor, en un Stream. Lo hace en dos pasos

1. En valor **tenemos una cadena con palabras**. Con la función Split divide el String, la cadena, en un array de frases en palabras
   ```
   String[] palabras = valor.split(" ");
   ```

Por **ejemplo el parámetro value contendrá el contenido de frases[0]** la primera vez y me dará un array
Contenido de frases[0] será value
```
Value ="En el siguiente ejemplo vamos a transformar una cadena en un array de cadenas"
```
Y lo transformamos a **un array de Strings** pero en este **caso palabras**
```
Palabras[= {“En”, “el, “siguiente”, “ejemplo”, …..
```

2. Me vuelve a transformar el array de palabras en un Stream, para continuar la operación y lo devuelvo.
   ```
   return (Stream<String>) Arrays.stream(palabras);
   ```

Esto lo realiza para los valores de frases[0], frases[1] y frases[2], que lo hemos transformado a Stream de cadenas previamente. En total **tres Streams que serán procesados por las siguiente operación, forEach.**

4.6.10 distinct()

El **método Java Stream distinct()** es una **operación no terminal que devuelve un nuevo Stream** que solo contendrá los **elementos diferentes del Stream original**, eliminando duplicados. Cualquier **duplicado será eliminado**. A continuación se muestra un **ejemplo del método Java Stream distinct():**

OperacionesNoTerminales2.java

```
import java.util.Arrays;

public class OperacionesNoTerminales2 {

        public static void main(String[] args) {
        int   arrayEnteros[]   =     new   int[]   {1,   2,   3,6,4,8,6,4,2,1,
5,6,7,8,6,5,8,7,9,10};
```

```
        System.out.println("\nPintamos el array");
        Arrays.stream(arrayEnteros).forEach(x->System.out.print(x+","));
        System.out.println("\nPintamos el array ordenado sin duplicados");
        Arrays.stream(arrayEnteros).
        sorted().
        distinct().
        forEach(x->System.out.print(x+","));

        }

}
```

4.6.11 **limit()**

El **método de la API Stream limit()** puede **limitar el número de elementos de un Stream** a un número dado al método limit() como parámetro. El método limit() devuelve un nuevo Stream que, como máximo, contendrá el número dado de elementos. En el siguiente ejemplo de limit(). Limitamos a los 4 primeros resultado el Stream

```
public static void main(String[] args) {
        int   arrayEnteros[]   =     new   int[]   {1,   2,   3,6,4,8,6,4,2,1,
5,6,7,8,6,5,8,7,9,10};


        System.out.println("\nPintamos el array");
        Arrays.stream(arrayEnteros).forEach(x->System.out.print(x+","));
        System.out.println("\nPintamos el array limitado a 4");
        Arrays.stream(arrayEnteros).
        sorted().
        limit(4).
        forEach(x->System.out.print(x+","));



        System.out.println("\nPintamos el array limitado a 4");
        Arrays.stream(arrayEnteros).
        limit(4).
        sorted().
        forEach(x->System.out.print(x+","));

}
```

En este ejemplo **quiero que observeis la diferencia de hacer el limit primero y luego ordenar** y hacer **el limit después de ordenar**. Debéis crear una clase vosotros mismos en este caso para probar el ejemplo. **EjemploLimit.java**

4.6.12 **peek()**

El método **Java Stream peek() es una operación no terminal** que toma un

Consumer (java.util.function.Consumer) como **parámetro**. Se **llamará al consumidor para cada elemento del Stream o secuencia**. El método **peek() devuelve la misma secuencia**. Esta construido para poder aplicaciones operaciones terminales como un Consumer, pero devolver el Stream para que la operación siga siendo no terminal.

El propósito **del método peek() es, como dice el método, revisar a los elementos de la secuencia, no transformarlos**. Tener en cuenta **que el método peek no inicia la iteración interna de los elementos de la secuencia o Stream**. Necesitas **llamar a una operación terminal** para eso.
En el siguiente ejemplo imprimimos el Stream y lo volvemos a convertir en un Array porque peek devuelve el mismo Stream

```
int arrayEnteros[] =  new int[] {1, 2, 3,6,4,8,6,4,2,1, 5,6,7,8,6,5,8,7,9,10};
        System.out.println("\nPintamos el array , y luego una coma despues de
cada elemento pintado");
        int arrayEnterosPeek[]=Arrays.stream(arrayEnteros).
        peek(x->System.out.print(x+",")).toArray();
```

4.6.13 Practica guiada de operaciones no terminales

Los ejemplos anteriores completos en sus clases java para operaciones no terminales:
OperacionesNoTerminales.java

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

public class OperacionesNoTerminales {

        public static void main(String[] args) {



                String  ArrayNombres[]  ={"Al",  "Ankit",  "Brent",  "Tomas",
"Alejandro",  "Aitor"  ,"Sarika",  "amanda",  "Hans",  "Shivika",  "Sarah",
"Julius"};


                Stream.of(ArrayNombres)
                .filter(x  ->  x.startsWith("a")  ||  x.startsWith("b")  ||
x.startsWith("s") )
                .forEach(System.out::println);

                Stream.of(ArrayNombres)
                .filter(x -> x.length()> 4 )
                .forEach(System.out::println);


                Stream.of(ArrayNombres)
                                .map(String::toLowerCase)
                                .filter(x    ->    x.startsWith("a")    ||
```

```
x.startsWith("b") || x.startsWith("s") )
                                .forEach(System.out::println);


                String[] frases =
{"En el siguiente ejemplo vamos a transformar una cadena en un array de cadenas"
,"separando cada palabra "
,"y agregandola a un array de Strings"};



                Stream.of(frases).
                flatMap((valor) -> {
                    String[] palabras = valor.split(" ");
                    return (Stream<String>) Arrays.stream(palabras);
                })
                .forEach((valor) -> System.out.print(valor+","));


        }

}
```

OperacionesNoTerminales2.java

```
import java.util.Arrays;

public class OperacionesNoTerminales2 {

        public static void main(String[] args) {
        int   arrayEnteros[]   =     new   int[]   {1,   2,   3,6,4,8,6,4,2,1,
5,6,7,8,6,5,8,7,9,10};

        System.out.println("\nPintamos el array");
        Arrays.stream(arrayEnteros).forEach(x->System.out.print(x+","));
        System.out.println("\nPintamos el array ordenado sin duplicados");
        Arrays.stream(arrayEnteros).
        sorted().
        distinct().
        forEach(x->System.out.print(x+","));


        System.out.println("\nPintamos el array limitado a 4");
        Arrays.stream(arrayEnteros).
        limit(4).
        sorted().
        forEach(x->System.out.print(x+","));



        System.out.println("\nPintamos el array , y luego una coma despues de
cada elemento pintado");
        int arrayEnterosPeek[]=Arrays.stream(arrayEnteros).
        peek(x->System.out.print(x+",")).toArray();


        }
```

```
}
```

4.6.14 Practica independiente de operaciones no terminales

Dado el siguiente array:

```
int arrayEnteros[] =  new int[] {1, 2, 3,6,4,8,6,4,2,1, 5,6,7,8,6,5,8,7,9,10};
```

Con operaciones de Stream realizaremos:

1. Ordenamos el array
2. Eliminamos repetidos
3. Mostramos con peek el resultado y seguimos procesando
4. Nos quedamos sólo con los números impares del array
5. Mostramos por pantalla los impares.

4.6.15 Operaciones Terminales

4.6.16 anyMatch()

El método de **la API Java Stream anyMatch()** es una **operación terminal** que toma un **único predicado como parámetro**, inicia la **iteración interna del Stream** y aplica el **parámetro Predicate a cada elemento**. Si el Predicate devuelve true para cualquiera de los elementos, el método anyMatch() devuelve true. Si ningún elemento coincide con el predicado, anyMatch() devolverá false. En el siguiente ejemplo vereis como detecta que algunas frases empiezan por en.

```
String[] frases =
        {"En el siguiente ejemplo hay que", " entablar cadenas"
,"en caso de que empiecen por ","en da igual mayuscula o minúscula",
        " solo si empiezan por en nos devolverá verdadero"};

boolean algunoEmpiezaPoren= Arrays.stream(frases).anyMatch((value) -> { return
value.toLowerCase().startsWith("en"); });


        if (algunoEmpiezaPoren) {

                System.out.println("Alguna de las frases empieza por 'en'");

        }
```

4.6.17 **allMatch()**

El método **Java Stream allMatch() es una operación terminal que toma un único Predicate como parámetro,** inicia la iteración interna de **los elementos de Stream y aplica el parámetro Predicate** a cada elemento. Si el predicado devuelve true para todos **los elementos de Stream, allMatch() devolverá true**. Si no todos los elementos coinciden con el predicado, **el método allMatch() devuelve false**. Lo podeis ver en el siguiente ejemplo. El **resultado es verdadero o falso.**

```
String[] frases =
        {"En el siguiente ejemplo hay que", " entablar cadenas"
        ,"en caso de que empiecen por ","en da igual mayuscula o minúscula",
        " solo si empiezan por en nos devolverá verdadero"};




boolean todasEmpiezanPoren= Arrays.stream(frases).
allMatch((value) -> { return  value.toLowerCase().startsWith("en"); });


        if (!todasEmpiezanPoren) {

        System.out.println("No todas las frases empieza por 'en'");

        }
```

En el siguiente ejemplo tenéis recogido las dos operaciones anteriores.

EjemploOperacionesTerminales1.java

```
import java.util.Arrays;

public class EjemploOperacionesTerminales1 {

        public static void main(String[] args) {


                String[] frases =
        {"En el siguiente ejemplo hay que", " entablar cadenas"
        ,"en caso de que empiecen por ","en da igual mayuscula o minúscula",
        " solo si empiezan por en nos devolverá verdadero"};
```

```
        boolean algunoEmpiezaPoren= Arrays.stream(frases).anyMatch((value) ->
{ return  value.toLowerCase().startsWith("en"); });


        if (algunoEmpiezaPoren) {

                System.out.println("Alguna de las frases empieza por 'en'");

                }

                boolean todasEmpiezanPoren=
Arrays.stream(frases).allMatch((value) -> { return
value.toLowerCase().startsWith("en"); });


        if (!todasEmpiezanPoren) {

                System.out.println("No todas las frases empieza por 'en'");

                }


        }

}
```

4.6.18 noneMatch()

El método java Stream noneMatch() es una operación de terminal que iterará los elementos del Stream y devolverá true o false, dependiendo de si ningún elemento de la secuencia coincide con el predicado pasado a noneMatch() como parámetro. El método noneMatch() devolverá true si no hay ningún elemento que coincida con el Predicado y false si uno o más elementos coinciden. Preparad vosotros un ejemplo esta vez

4.6.19 Ejercicio noneMatch

6. Sobre el ejemplo anterior probar el funcionamiento de noneMatch.

4.6.20 collect()

El método Java `Stream collect()` es una operación terminal que realiza una iteración interna de los elementos del Stream y recopila los elementos de la secuencia en una colección u objeto de algún tipo. Lo veremos en el tema 8 con colecciones.

4.6.21 **count()**

El método count es una operación terminal que nos cuenta el número de elementos del Stream. **Usando el ejemplo de flatMap** que me convertía **a un Stream de palabras**, los **tres arrays de frases**, vamos a usar el count para contar el número de palabras total en los tres arrays. Ya lo explicamos antes, en el flatmap obtiene un stream de palabras del Stream de frases, para obtener un Stream de palabras. Luego **contamos el número total de palabras con count.**

```
long numpalabras =Stream.of(frases).
        flatMap((valor) -> {
            String[] palabras = valor.split(" ");
            return (Stream<String>) Arrays.stream(palabras);
        }).count();
```

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

public class EjemploOperacionesTerminales2 {


        public static void main(String[] args) {

        String[] frases =
                {"En el siguiente ejemplo vamos a transformar una cadena en
un array de cadenas"
        ,"separando cada palabra ","y agregandola a un array de Strings"};



        long numpalabras =Stream.of(frases).
        flatMap((valor) -> {
            String[] palabras = valor.split(" ");
            return (Stream<String>) Arrays.stream(palabras);
        }).count();


        System.out.println("El número de palabras en el array es: " +
numpalabras);

        }

}
```

Para que **el programa sea más funcional deberíamos** evitar la **asignación de variables**. En los ejemplos **estamos asignando variables** para que **se entienda mejor**. Pero más adelante dejare de asignar variables para hacerlo todo funcional. Si lo hacemos de esta manera sólo tendríamos como asignación la creación del array. El resto con operaciones inmutables, que no asignen variables en nuestros programas.

```
        System.out.println("El número de palabras en el array es: " +
Stream.of(frases).
                flatMap((valor) -> {
                String[] palabras = valor.split(" ");
                return (Stream<String>) Arrays.stream(palabras);}).count());
```

4.6.22 findAny() y findFirst()

Estos métodos devuelven **un objeto de tipo Optional** para recoger resultados de un Stream. Vamos a usar el ejemplo anterior de filter para probar estos dos métodos. Son muy útiles porque nos dan la posibilidad de devolver tipos nulos de nuestro Stream sin que se produzca la excepción NullPointerException. **findAny()** nos va a devolver si hay alguno. **findFirst()** nos va a devolver el primer elemento del Stream. En cualquier caso nos quedamos con un objeto de todos los que manda el Stream.

ejemploTerminalesBusqueda.java

```
import java.util.Optional;
import java.util.stream.Stream;

public class ejemploTerminalesBusqueda {

        public static void main(String[] args) {

String ArrayNombres[] ={"Al", "Ankit", "Brent", "Tomas", "Alejandro", "Aitor"
,"Sarika", "amanda", "Hans", "Shivika", "Sarah", "Julius"};




                Optional nombre =Stream.of(ArrayNombres)

                                .map(String::toLowerCase)
                                .filter(x -> x.startsWith("a") ||
x.startsWith("b") )
                                .findAny();

                if (nombre.isPresent()) {


                System.out.println("encontramos nombres que empiezan por A:"
+ nombre.get());
                } else {

                System.out.println("no encontramos nombres que empiezan por
A" );
                }

                Optional nombre2 =Stream.of(ArrayNombres)

                .map(String::toLowerCase)
                .filter(x -> x.startsWith("a") || x.startsWith("b") )
                .findFirst();
```

```
nombre2.ifPresent( s->   System.out.println("encontramos el primer nombre que
empiezan por A:" + s));



                Optional nombre3 =Stream.of(ArrayNombres)

                                .map(String::toLowerCase)
                                .filter(x -> x.startsWith("m"))
                                .findFirst();

        if (nombre3.isPresent()) {

                System.out.println("encontramos nombres que empiezan por m:"
        + nombre.get());
                }
                else {

        System.out.println("no encontramos nombres que empiezan por m");
                }


        }

}
```

4.6.23 **max() y min()**

El método **Java Stream min() es una operación de terminal** que devuelve **el elemento más pequeño del Stream**. El elemento más pequeño viene determinado por la implementación **de Comparator que se pasa al método min()**. Sino se pasa ninguna usara el Comparable de los objetos que maneja el Stream. El **método max() es igual pero devuelve el valor máximo**. Recordar que la mayoria de las clases ya tienen implementado el comparator. Al igual que findAny y findFirst, devuelven un tipo Optional con el objeto tipo del Stream.

```
import java.util.Arrays;
import java.util.OptionalInt;

public class OperacionesTerminalesMaxMin {

        public static void main(String[] args) {
        int arrayEnteros[] =  new int[] {1, 2, 3,6,4,8,6,4,2,1,
5,6,7,8,6,5,8,7,9,10};

                System.out.println("\nPintamos el array");
                Arrays.stream(arrayEnteros).forEach(x-
>System.out.print(x+","));

                OptionalInt minimo = Arrays.stream(arrayEnteros).min();
```

```
                System.out.println ("\nEl número mínimo del array es:"+
minimo.getAsInt());


            Arrays.stream(arrayEnteros)
            .max()
                .ifPresent((num)->System.out.println ("El número máximo del
array es:"+ num));


}
```

Hemos usado la **versión de Optional** para el tipo básico Int, **Optional**

4.6.24 reduce()

Es uno de los **métodos más complejos junto a flatMap**. Va a reducir el Stream a un solo **objeto aplicando algun tipo de operación de agregación**. Recibe como parámetro un **BynaryOperator<T>, que es una implementacion particular del BiFunction** que ya hemos visto. El **BinaryOperator se comporta exactamente igual que el UnaryOperator, pero recibe dos parámetros de entrada**, del **mismo tipo**. Ya vimos que en el UnaryOperator el **tipo de entrada y de salida es el mismo**. Por tanto, para el BinaryOperator, **la salida será del mismo tipo que las entradas**. Finalmente, el objeto al que se reduce el Stream es del mismo tipo que los objetos que maneja el Stream.
Teneis la definición para el interfaz en la página oficial de Oracle:
https://docs.oracle.com/javase/8/docs/api/java/util/function/BinaryOperator.html
El método **reduce()** **devuelve un Optional igualmente**, como los anteriores.
En el siguiente ejemplo vamos a obtener como resultado la concatenación de todos los String del Stream que hemos obtenido a partir del array.

```
((strCombinado, str)-> strCombinado+str
```

En la expresión lambda podéis apreciar el BynaryOperator. Dos parámetros de entrada y uno de salida que son del mismo tipo en este caso String. El reduce se aplica sobre cada elemento del stream, que es el primer parámetro de la expresión lambda**. El primer parámetro va a ser la salida del reduce** que se aplicó para el elemento anterior del Stream, **porque internamente es recursivo**

El primer elemento del Stream llega al reduce, strCombinado será la cadena vacia la primera vez.
(“”, “Al”) -> “Al” es el resultado sobre el primer elemento
(“Al”,” Ankit”,) -> “AlAnkit”
(“AlAnkit” ,“Brent”)-> “AlAnkitBrent”
…. Y así sucesivamente hasta que finalice el Stream

En el ejemplo en la **variable de tipo Optional concatenación** se **almacena** el **resultado del reduce**. Ya sabeis **que lo correcto sería no guardar en variables** como hemos hecho en el ejemplo anterior. Lo indicamos así **para que se entienda mejor los ejemplos.**
EjemploReduce.java

```
import java.util.Optional;
import java.util.stream.Stream;

public class EjemploReduce {

        public static void main(String[] args) {

        String ArrayNombres[] ={"Al", "Ankit", "Brent", "Tomas", "Alejandro",
"Aitor" ,"Sarika", "amanda", "Hans", "Shivika", "Sarah", "Julius"};


        Optional concatenacion = Stream.of(ArrayNombres)
        .reduce((strCombinado, str)-> strCombinado+str );

        concatenacion.ifPresent(System.out::println);
        }

}
```

La manera funcional sería;

```
Stream.of(ArrayNombres)
.reduce((strCombinado, str)-> strCombinado+str)
.ifPresent(System.out::println);
```

4.6.25 **Practica independiente de operaciones terminales**

Dado el siguiente array:

```
int arrayEnteros[] =  new int[] {1, 2, 3,6,4,8,6,4,2,1, 5,6,7,8,6,5,8,7,9,10};
```

Con operaciones de Stream realizaremos, mostrando cada resultado por pantalla:

1. Preguntar si existe un número recogido por pantalla en el array con AnyMatch.
2. Si existe con find encontrarlo y mostrarlo por pantalla
3. Contar todos los elementos que son distintos
4. Encontrar el máximo y el mínimo

4.6.26 **forEach() y toArray()**

Ya **han sido descritos y no hace falta que los expliquemos en detalle**. El primero es un **consumer** que se aplica sobre **cada elemento del Stream**. El segundo **transforma el Stream en un Array de objetos** de la clase Object().

4.7 IntStream, Double Stream, LongStream

IntStream, LongStream, DoubleStream son tipos de Stream para los tipos básicos int, long y double. **Son muy útiles cuando se trata de realizar iteraciones**, y trabajar con números. Vamos a ver algún ejemplo sobre **IntStream bastante interesante**, su utilidad es la de poder **realizar iteraciones y operaciones sobre números**. Muy útil también para **trabajar con arrays numéricos**. Podemos usar la **mayoría de funciones de la API Stream y añaden alguna nueva** que veremos en el curso. Todo lo expuesto a continuación es aplicable a DoubleStream, LongStream, etc. Podemos igualmente usar los métodos que ya hemos visto anteriormente para estos Streams.

4.7.1 IntStream.of()

Esta **función devuelve Stream de tipo int** ordenado cuyos **elementos son los valores** especificados.

Tenemos dos versiones, es decir, flujo de un solo elemento y múltiples valores de flujo

IntStream of(int t) – Devuelve **un Stream que contiene un único elemento** especificado.
IntStream of(int... values) – **Devuelve un Stream que contiene todos** los elementos especificados.
IntStream.of(10); //10
IntStream.of(1, 2, 3); //1,2,3

4.7.2 IntStream.iterate()

La función iterator() es útil para crear secuencias infinitas. Además, podemos usar
este método para producir secuencias donde los valores se incrementan en cualquier otro
valor que 1.
En el siguiente el Stream produce los primeros 10 números pares a partir de 0.

```
IntStream.iterate(0, i -> i + 2).limit(10);

//0,2,4,6,8,10,12,14,16,18
```

4.7.3 IntStream.generate()

El método `generate()` se parece **mucho a iterator(), pero difieren al no calcular los valores int por incrementar el valor anterior**. Más bien **se proporciona un IntSupplier** que es una interfaz **funcional** que se utiliza para generar una secuencia secuencial infinita **desordenada** de valores int.

El ejemplo siguiente crea una secuencia de 10 números aleatorios y, a continuación,

imprimirlos en la consola.

```
IntStream stream = IntStream.generate(()
            -> { return (int)(Math.random() * 100); });

stream.limit(10).forEach(System.out::println);
```

4.7.4 IntStream range()

El `IntStream` generado por los métodos `range()` es un Stream **secuencial de valores int.** Sería **equivalente a aumentar los valores int en un bucle for** y el **valor incrementado en 1.** Esta clase admite dos métodos.

1. **range(int startInclusive, int endExclusive)** – Devuelve un Stream de tipo int que comienza con startInclusive*(incluido)* y termina endExclusive (no incluido) con un incremento de una unidad.
2. **rangeClosed(int startInclusive, int endInclusive)** – Devuelve Stream de tipo int ordenado que comienza en startInclusive*(incluido)* y termina endInclusive *(no incluido)* con un incremento de una unidad.

Tenéis todos los ejemplos en la siguiente clase java.

EjemplosBasicos.java

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

public class EjemplosBasicos {


        public static void main(String[] args) {

                System.out.println("Con iterator");

                IntStream.iterate(0, i -> i + 2).limit(10).forEach(x-
>System.out.print(x+","));

                IntStream stream = IntStream.generate(()
                    -> { return (int)(Math.random() * 10000); });

                System.out.println("\nCon generate");

                stream.limit(10).forEach(x->System.out.print(x+","));
```

```
                System.out.println("\nCon range");
                IntStream streamRange = IntStream.range(5, 10);
                streamRange.forEach( x->System.out.print(x+",") );
//5,6,7,8,9


                System.out.println("\nCon closerange");
        //Closed Range
        IntStream streamClosedRange = IntStream.rangeClosed(5, 10);
        streamClosedRange.forEach( x->System.out.print(x+",") );
//5,6,7,8,9,10

        }

}
```

4.7.5 map() y maptoObject(). Practica guiada de mapeo de IntStream a Objetos.

La **version de Map para intStream sólo permite mapear** al **tipo básico int**. Si queremos **mapear a objetos debemos usar maptoObject**
Ejemplo para map: calculamos el cubo de los números en el Stream.

EjemploMapIntStream.java

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

public class EjemploMapIntStream {



        public static void main(String[] args) {


        System.out.println("\nMap");
        IntStream streamRange = IntStream.range(5, 10);
        streamRange.forEach(x->System.out.print(x+","));
        System.out.println("\nCalculamos el cubo para todos los elementos del
Stream");
        IntStream.range(5, 10).map(x-> (int) Math.pow(x, 3)).forEach(x-
>System.out.print(x+","));


        }


}
```

Nota: Cuando **operamos sobre un Stream que asignamos a una variable**, el **Stream se cierra no** se puede **volver a operar con el**

El **siguiente ejemplo fallará**, pero probarlo para comprobarlo:

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

public class EjemploMapIntStream {



        public static void main(String[] args) {


        System.out.println("\nMap");
        IntStream streamRange = IntStream.range(5, 10);
        streamRange.forEach(x->System.out.print(x+","));
        System.out.println("\nCalculamos el cubo para todos los elementos del
Stream");
        streamRange.range(5, 10).map(x-> (int) Math.pow(x, 3)).forEach(x-
>System.out.print(x+","));


        }


}
```

Con **mapToObject** vamos **a modificar el ejemplo que usamos con Map** para construir un **array de objetos a partir de un Stream**. Como el ejemplo es el mismo que usamos antes, vamos a explicar los cambios.

Hemos **eliminado el array de int que usamos para el otro ejemplo**, ya no hace falta. Generamos **el Stream de tipo int con IntStream**. En **lugar de map, usamos maptoObject**. El resto del ejemplo es exactamente igual.
Ejemplo actual:

```
IntStream.rangeClosed(0,24).mapToObj((num) -> (Empleado) new
Empleado(num,nombres[num])).toArray();
```

Ejemplo anterior:
Eliminamos el array números, y cambiamos el array números en el Stream por IntStream para generar la secuencia de números. El resto es igual.

```
public static final Integer numeros[]=
{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24};

…

         Object empleados[] =
        Arrays.stream(numeros).map((num) -> (Empleado) new
Empleado(num,nombres[num])).toArray();
```

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

```
        class Empleado {

                int id;
                String nombre;
                public Empleado (int id, String nombre) {

                        this.id=id;
                        this.nombre=nombre;

                }

                public int getId() {
                        return id;
                }

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

                public String getNombre() {
                        return nombre;
                }

                public void setNombre(String nombre) {
                        this.nombre = nombre;
                }


                public String toString () {

                return "{id:"+ this.getId() + ", nombre:" + this.getNombre()
+ "}";
                }

        }

        public class MaptoObjEmpleados {


          public static final String nombres[] = {"ANTONIO", "MANUEL",
"JOSE", "FRANCISCO", "DAVID", "JUAN", "JOSE ANTONIO", "JAVIER", "DANIEL",
"JOSE LUIS", "FRANCISCO JAVIER",
                    "CARLOS", "JESUS", "ALEJANDRO", "FRANCISCA", "LUCIA",
"MARIA ISABEL", "MARIA JOSE", "ANTONIA", "DOLORES", "SARA",
                    "PAULA", "ELENA", "MARIA LUISA", "RAQUEL"};



        public MaptoObjEmpleados() {



        }


        public static void main(String[] args) {


                Object empleados[] =
```

```
         IntStream.rangeClosed(0,24).mapToObj((num) -> (Empleado) new
Empleado(num,nombres[num])).toArray();


                  Arrays.stream(empleados).forEach((objEmpleado) ->
System.out.println(((Empleado) objEmpleado)));

                }




}
```

4.7.6 Practica Independiente de mapeo de objetos.

Dado el siguiente array de apellidos:

```
        public static final String apellidos[] = {"García", "González",
"Rodríguez", "Fernández", "López", "Martínez", "Sánchez", "Pérez", "Gómez",
                        "Martin", "Jiménez", "Ruiz", "Hernández", "Diaz",
"Moreno", "Muñoz", "Álvarez", "Romero", "Alonso", "Gutiérrez", "Navarro",
                        "Torres", "Domínguez", "Vázquez", "Ramos", "Gil",
"Ramírez", "Serrano", "Blanco", "Molina", "Morales", "Suarez", "Ortega",
                        "Delgado", "Castro", "Ortiz", "Rubio", "Marín",
"Sanz", "Núñez", "Iglesias", "Medina", "Garrido", "Cortes", "Castillo",
"Santos"};
```

1. Añadir el atributo Apellidos en el modelo anterior a la clase Empleado
2. Modificar el ejemplo para añadir también los apellidos en la creación del array de objetos.

4.7.7 IntStream y funciones de agregación. Practica guiada de funcionas agregación

Podemos **usar el IntStream en combinación con reduce para calcular operaciones matemáticas de agregación** como una **suma o multiplicación de números, medias**, y otras **operaciones estadísticas.**
sum() te **devuelve la suma del IntStream**
average() te **devuelve la media del Stream**. Devuelve **un OptionalDouble**. Funciona exactamente igual que **Optional**.

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

```
public class IntStreamAgregacion {



        public static void main(String[] args) {

                int suma = IntStream.range(1, 10).sum();

                System.out.println("La suma del IntStream es:" +suma);


                OptionalDouble media = IntStream.range(1, 10).average();
                System.out.println("La media del IntStream es:"
+media.getAsDouble());

        }

}
```

De **IntStream** como de otros Stream y lo veremos mas adelante con **el objeto Collect**, podemos **obtener un objeto** de **estadísticas** de la clase `IntSummaryStatistics.` La clase **IntSummaryStatistics guarda información estadística del Stream**. Los **métodos de agregación como max, min, average, podemos volver usarlos** sobre esta clase **aunque el Stream se haya cerra**do, llamando **al método summaryStatistics() de Stream** y usando el objeto posteriormente.

```
IntSummaryStatistics  estadisticas= IntStream.range(1,
10).summaryStatistics();
```

IntStreamAgregacion.java

```
import java.util.IntSummaryStatistics;
import java.util.OptionalDouble;
import java.util.OptionalInt;
import java.util.stream.IntStream;

public class IntStreamAgregacion {



        public static void main(String[] args) {

                int suma = IntStream.range(1, 10).sum();

                System.out.println("La suma del IntStream es:" +suma);


                OptionalDouble media = IntStream.range(1, 10).average();
                System.out.println("La media del IntStream es:"
+media.getAsDouble());


                IntSummaryStatistics  estadisticas= IntStream.range(1,
```

```
10).summaryStatistics();

                System.out.println("Datos para el Stream. Media:" +
estadisticas.getAverage() + " Suma:" +
                estadisticas.getSum() +" Mínimo: " + estadisticas.getMin() +
" Máximo:" + estadisticas.getMax()+
                " total elementos " + estadisticas.getCount());

IntSummaryStatistics  estadisticas2= IntStream.range(4,
12).summaryStatistics();

estadisticas.combine(estadisticas2);

                System.out.println("Datos para el Stream. Media:" +
estadisticas.getAverage() + " Suma:" +
                                estadisticas.getSum() +" Mínimo: " +
estadisticas.getMin() + " Máximo:" + estadisticas.getMax()+
                                " total elementos " +
estadisticas.getCount());

        }

}
```

Podemos **combinar varios objetos estadísticos con Combine**, como **podéis ver en la parte final del ejemplo.**

```
estadisticas.combine(estadisticas2);
```

4.7.8 Practica independiente de funciones de agregación

Para el ejemplo de intStream y mapeo a empleados calcular y mostrar por pantalla.

1. Suma de id's
2. Máximo id y mínimo.
3. Media de id's de los empleados.

4.7.9 IntStream y recursividad

Podemos **usar el IntStream en combinación con reduce para realizar cálculos recursivos** como el **factorial**. Intentarlo. Se deja abierto para los alumnos. Haced cálculos como **factorial, fibonnaci, usando IntStream** y **reduce**.

4.8 Concatenando Streams

Con **el método estático de la clase Stream, concat**, podemos **concatenar dos**

Streams del mismo tipo. Es **muy sencillo**, lo podemos ver sobre el ejemplo

Obtenemos un Stream, stream3, concatenando stream1 y stream2.

```
Stream<String> stream3 = Stream.concat(stream1, stream2);
```

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

public class ConcatenarStreams {

	public static void main(String[] args) {



		String ArrayNombres[] ={"Al", "Ankit", "Brent", "Tomas",
"Alejandro", "Aitor" ,"Sarika", "amanda", "Hans", "Shivika", "Sarah",
"Julius"};

		String ArrayNombres2[] ={"lodi", "samuel" ,"iker" ,
"Jeronimo", "Gracielo", "Rene","Reus"};

		Stream<String> stream1 =Arrays.stream(ArrayNombres);

		Stream<String> stream2 =Arrays.stream(ArrayNombres2);

		Stream<String> stream3 = Stream.concat(stream1, stream2);
		System.out.println("Concatemos los dos Stream arrays y
obtenemos uno con los elementos de los dos");
		stream3.forEach(x->System.out.print(x+","));


	}

}
```

4.9 Stream paralelos

Podemos **convertir todos los ejemplos anteriores a su versión paralela**. La API Stream **nos permite ejecutar paralelamente nuestros programas sobre Streams**. Esto significa que para **resolver la operación de Streams en lugar de usar uno sólo núcleo de vuestro procesador** puede **usar varios núcleos, y un trozo del procesado del Stream** se puede **realizar en cada procesador**, lo único que **hay que realizar es llamar al método parallel()**, que transforma **nuestra ejecución en paralela**. En el ejemplo anterior hemos **llamado al método parallel() para hacer nuestra ejecución paralela**.. Hay un **método isParallel(),** para comprobar si el stream es paralelo. Podéis **probar a incluirlo en ejemplos anteriores.**

```
Arrays.stream(ArrayNombres)
	.parallel()
	.reduce((strCombinado, str)-> strCombinado+str)
	.ifPresent(System.out::println);
```

5 El operador …

El operador … esta disponible desde la versión 5 de java y es equivalente a la lista de parámetros vararg que aparece en nuestra función main. ¿Qué quiere decir los …? Indican un número ilimitado de parámetros de un tipo determinado. Lo vamos a ver en los siguientes ejemplos, pero básicamente si en mi función param(String … strings) defino los parámetros con puntos suspensivos , significa que estoy pasando un número ilimitado de cádenas. Para llamar a esta función puedo pasar una cadena param("Hola"), dos cadenas param("hola", "adios"), tres cadenas param("hola", "adios", "hola"), un número de parámetros indefinidos en resumen.
Dentro de la función mi parámetro strings será considerado como un array de cadenas, de Strings. Vamos a verlo en detalle en el siguiente ejemplo
Nota: en caso de usar el operador … y parámetros normales en la misma función debemos colocar el parámetro del operador … en última posición.

```
public int funcionConPuntosSuspensivosEnteros (String nombre, int ...numeros )
```

Vamos a diseccionar el ejemplo paso a paso. Se ha introducido dos versiones de cada función, una declarativa y otra funcional.

La primera función `funcionConPuntosSuspensivosCadena`, recibe como parámetro `String ...strings` y los recorre con un bucle for.

```
for (String parametro: strings) {

                System.out.println(parametro);

        }
```

Cuando llamamos a esta función pasamos 4 cadenas como parámetros. El compilador se encarga de transformar esas cuatro cadenas en un array de Strings.

```
opPuntos.funcionConPuntosSuspensivosCadena("hola","cadena","adios","cadena2")
;
```

La versión lambda de la misma función usa un Stream para realizar el mismo trabajo:
`funcionConPuntosSuspensivosCadenaVersionLambda`, recibe el mismo parámetro `String ...strings` y lo procesa con un Stream.

```
Stream.of(strings).forEach(System.out::println);
```

Para parámetros de tipo int la función `funcionConPuntosSuspensivosEnteros(String Nombre, int … números)`, recibe una cadena y luego un número indeterminado de enteros. Calcula la suma de los números con un bucle for y la devuelve. Importante los parámetros con el operador puntos suspensivo siempre al final de la declaración de la función cuando aparecen con otros parámetros.

```
for (int parametro: numeros) {
```

```
                suma= parametro+ suma;

        }

        return suma;
```

Vuestra tarea será estudiar la versión funcional public Optional `funcionConPuntosSuspensivosEnterosVersionLambda (String nombre, Integer ...numeros )` y deducir que hace.

OperadorPuntosSuspensivos.java

```
import java.util.Optional;

import java.util.stream.Stream;

public class OperadorPuntosSuspensivos {

 public OperadorPuntosSuspensivos() {



 }


 public void funcionConPuntosSuspensivosCadena(String ...strings  ) {

         for (String parametro: strings) {

                 System.out.println(parametro);

         }

 }

public void funcionConPuntosSuspensivosCadenaVersionLambda(String ...strings
) {


        System.out.println("Version lambda");
         Stream.of(strings).forEach(System.out::println);

 }



 public int funcionConPuntosSuspensivosEnteros (String nombre, int ...numeros
) {

         int suma=0;
         for (int parametro: numeros) {

                 suma= parametro+ suma;
```

```
         }


         return suma;


 }


 public Optional funcionConPuntosSuspensivosEnterosVersionLambda (String
nombre, Integer ...numeros ) {

         int suma=0;


         System.out.println(nombre);
         return Stream.of(numeros).reduce((sumaAcumulada,numero)->
sumaAcumulada+numero);




 }


        public static void main(String[] args) {


                OperadorPuntosSuspensivos opPuntos = new
OperadorPuntosSuspensivos();


        opPuntos.funcionConPuntosSuspensivosCadena("hola","cadena","adios","c
adena2");


        opPuntos.funcionConPuntosSuspensivosCadenaVersionLambda("hola","caden
a","adios","cadena2lambda", "lambda exp");

                int suma =
opPuntos.funcionConPuntosSuspensivosEnteros("Version normal", 1,2,3,4,7,9,0);

                System.out.println("devuelve la suma de los números pasados
como parámetro: "+ suma);

                Optional<Integer> optSuma=
opPuntos.funcionConPuntosSuspensivosEnterosVersionLambda("Version lambda",
4,5,6,1,2,3,4,7,9,0);

                optSuma.ifPresent(sumaLambda-> System.out.println("devuelve
la suma de los números pasados como parámetro: "+ sumaLambda));

        }

}
```

6 Composición avanzada. Uso de Interfaces como parámetros

6.1 Interfaces como parámetros

Para finalizar el tema, vamos **a estudiar como usar los interfaces funcionales como parámetros** y como **componerlos para aplicarlos todos como un único método**. En este caso vamos a **definir un método en nuestra clase** que reciba **parámetros de tipo interfaz Function**.

En el método `ComposicionAvanzadaConParametros` recibimos tres parámetros de tipo **interfaz Function, interface1, interface2 e interface 3.** Los combinamos con **andThen** y los devolvemos, ya que esta función **es una función de orden superior** que devuelve **un interface Function** combinación de los tres.

```
public Function <Integer,Integer> ComposicionAvanzadaConParametros
(Function<Integer,Integer> interface1,Function<Integer,Integer>
interface2,Function<Integer,Integer> interface3) {

                return  interface1.andThen(interface2).andThen(interface3);
        }
```

Cuando llamamos al método en main, en nuestro programa principal le pasamos como parámetro tres expresiones lambda, que implementan a los tres interfaces funcionales.

```
ejemploAvanzada.ComposicionAvanzadaConParametros( x -> x*x, x->x-7, x->x/3);
```

Finalmente ejecutamos el interface resultado con Apply, y obtenemos el resultado de aplicar tres funciones combinadas.

```
   System.out.println("El resultado de combinar los tres interfaces es:" +
funcionResultado.apply(numero1));
```

```
import java.util.Scanner;
import java.util.function.Function;

public class ComposicionAvanzadaConParametros{

public Function <Integer,Integer> ComposicionAvanzadaConParametros
(Function<Integer,Integer> interface1,Function<Integer,Integer>
interface2,Function<Integer,Integer> interface3) {
```

```
                return  interface1.andThen(interface2).andThen(interface3);
        }



                public static void main(String[] args) {

                        Integer numero1;
                        Scanner miScanner = new Scanner(System.in);


                    System.out.println("Escriba un número entero");
                    numero1 = miScanner.nextInt();



                        ComposicionAvanzadaConParametros ejemploAvanzada =
new ComposicionAvanzadaConParametros();


                        Function <Integer,Integer> funcionResultado =

        ejemploAvanzada.ComposicionAvanzadaConParametros( x -> x*x, x->x-7,
x->x/3);

                   System.out.println("El resultado de combinar los tres
interfaces es:" + funcionResultado.apply(numero1));

        }


}
```

6.2 *Recorriendo los parámetros interfaz funcional del operador … con un for.*

En el siguiente ejemplo vamos a **aprovechar la potencia del operador …** para **recorrer el conjunto de parámetros interfaces funcionales** y aplicarlos todos, componerlos y obtener un **intefaz funcional resultado composición** de todos los **interfaces funcionales**. Nos va a permitir pasar **una cantidad ilimitada de interfaces funcionales** para ser **aplicados sobre un parámetro** del tipo asignado.

Fijaos en la cabecera de la función **recibe n parámetros de tipo interfaz Function** de tipo **Integer**. Function<Integer,Integer>. Como re**sultado esta función devuelve otro interfaz funcional**, un interfaz **Function<Integer,Integer>**, resultado de **componer todos los anteriores** pasados como **parámetros**. Es una **función de nivel superior**, como parámetros recibe funciones (Interfaces funcionales) y devuelve como resultado un interfaz funcional Function<Integer,Integer>.

```
public Function <Integer,Integer> composicionAvanzadaConParametrosPuntos
```

```
(Function<Integer,Integer> ... paramInterfaces ) {
```

Para componer los **interfaces funcionales usamos un bucle for y andThen**. **Inicializo resultado con la función identidad x->x**. Esta función devuelve el mismo resultado que le pasas, si x=1, devuelve un 1. Es decir, no tiene ningún efecto. Se usa la función identidad para inicializar interfaces que **tengan un efecto neutro**. Es lo mismo que **inicializar una variable entero i=1**, si luego vamos a multiplicar, o una variable contador=0, si luego vamos a sumar. **Sumar 0 no tiene ningún efecto. Componer con la función identidad** no tiene **ningún efecto.**

```
Function <Integer,Integer > resultado = x->x;

                for (Function <Integer,Integer> interfaz: paramInterfaces) {

                        resultado = resultado.andThen(interfaz);

                }
```

Devolvemos resultado. `return resultado;` Es el **resultado de componer todos los interfaces** o **expresiones lambda** pasados como parámetro. Cuando llamamos al método obtenemos un interfaz como parámetro de resultado. Ese interfaz **Function aplicará todas las expresiones lambda** pasadas como **parámetro.**

```
Function <Integer,Integer> funcionResultado =

        ejemploAvanzada.composicionAvanzadaConParametrosPuntos( x -> x*x, x-
>x-7, x->x*3);
```

```
funcionResultado.apply(numero1));
```

Imaginad que numero1=10. Primero se hará la función x->x*x , que nos devuelve el cuadrado. **10*10 =100**. Luego la segunda lambda. **100-7=93**. Y al final **la última lambda 93*3=289**. Así podría **estar añadiendo expresiones lambda** (interfaces funcionales) **indefinidamente.** De este modo puedo aplicar todas las funciones sobre un número, o un objeto que pase como parámetro, **una detrás de otra**. La potencia de calculo del lenguaje java ha sido aumentada gracia a la introducción de la programación funcional.
Probad a añadid vuestras propias expresiones lambda a la lista de parámetros y observar el resultad.

```
import java.util.function.Function;

import java.util.Scanner;


public class ComposicionAvanzadaConParametrosPuntosSuspensivos{

        public Function <Integer,Integer>
composicionAvanzadaConParametrosPuntos (Function<Integer,Integer> ...
paramInterfaces ) {
```

```
                Function <Integer,Integer > resultado = x->x;

                for (Function <Integer,Integer> interfaz: paramInterfaces) {

                        resultado = resultado.andThen(interfaz);

                }

                return resultado;
        }



                public static void main(String[] args) {

                        Integer numero1;
                        Scanner miScanner = new Scanner(System.in);


                    System.out.println("Escriba un número entero");
                    numero1 = miScanner.nextInt();



                    ComposicionAvanzadaConParametrosPuntosSuspensivos
ejemploAvanzada = new ComposicionAvanzadaConParametrosPuntosSuspensivos();


                        Function <Integer,Integer> funcionResultado =

        ejemploAvanzada.composicionAvanzadaConParametrosPuntos( x -> x*x, x-
>x-7, x->x*3);

                  System.out.println("El resultado de combinar los
interfaces es:" + funcionResultado.apply(numero1));

        }


}
```

6.3 Recorriendo los parámetros interfaz funcional del operador … con un Stream. Practica guiada composición de interfaces pasados como parámetros.

En el siguiente ejemplo vamos a **aprovechar la potencia de la API Stream** para **recorrer el conjunto de parámetros interfaces funcionales** y aplicarlos todos. Nos va a permitir pasar **una cantidad ilimitada de interfaces funcionales** para ser **aplicados sobre un parámetro** del tipo asignado.

Fijaos en la cabecera de **la función recibe n parámetros de tipo Function**

génerico. El **primero parámetro es un elemento del Tipo T**, `T i`, sobre el que **vamos a aplicar todos los interfaces funcionales** pasados como parámetro.

```
Optional<Function<T, T>> opt =
public void ComposicionAvanzada (T i, Function<T,T> ... interfaces)
```

Vamos a **recorrerlos y a componerlos** todos con un **Stream, reduce y andThen**

```
Optional<Function<T, T>> opt =
Stream.of(interfaces).reduce((funcComb, func)-> funcComb.andThen(func));
```

Declaramos la clase con su **tipo génerico Integer.**

```
EjemploDeComposicionAvanzadaPuntosSuspensivos<Integer> ejemploAvanzada = new
EjemploDeComposicionAvanzadaPuntosSuspensivos();
```

El resultado almacenado en el tipo Optional opt, es un interfaz funcional compuesto por todos los interfaces que hemos pasado como parámetro**, se van a aplicar todas estas expresiones lambda en orden.** `x -> x*x, x->x-7, x->x/3, x->x*5,` tras realizar la **llamada a la función**. Podemos **seguir pasando más expresiones lambda** a la siguiente llamada **indefinidamente.** Nos proporciona una **potencia de cálculo enorme**.

```
ejemploAvanzada.ComposicionAvanzada(10, x -> x*x, x->x-7, x->x/3, x->x*5);
```

Pódria seguir escribiendo expresiones lambda una detrás de otra indefinidamente.

```
ejemploAvanzada.ComposicionAvanzada(10, x -> x*x, x->x-7, x->x/3, x->x*5 , x-
>x-1, x->x/5)
```

```
import java.util.Optional;
import java.util.function.Function;
import java.util.stream.Stream;

public class EjemploDeComposicionAvanzadaPuntosSuspensivos<T> {


        public EjemploDeComposicionAvanzadaPuntosSuspensivos() {



        }


        public void ComposicionAvanzada (T i, Function<T,T> ... interfaces) {

                T valor = interfaces[0].apply(i);

                System.out.println(valor);
            Optional<Function<T, T>> opt =
Stream.of(interfaces).reduce((funcComb, func)-> funcComb.andThen(func));
            opt.ifPresent(x -> System.out.println("El resultado de aplicar
los interfaces funcionales a la variable " + i + " es " +x.apply(i) ));

            Stream.of(interfaces).
            reduce((funcComb, func)-> funcComb.andThen(func)).
            ifPresent(x -> System.out.println("El resultado de aplicar los
interfaces funcionales a la variable " + i + " es " +x.apply(i) ));
```

```
        }



        public static void main(String[] args) {

                EjemploDeComposicionAvanzadaPuntosSuspensivos<Integer>
ejemploAvanzada = new EjemploDeComposicionAvanzadaPuntosSuspensivos();

                ejemploAvanzada.ComposicionAvanzada(10, x -> x*x, x->x-7, x-
>x/3, x->x*5);


        }

}
```

7 Recursividad en Arrays

Vamos a **ver un par de ejemplos de como tratar los arrays recursivamente**. Se usa mucho en métodos de **ordenación y busqueda de arrays**. Vamos a intentar explicar la recursividad en arrays con estas búsquedas y reordenaciones. Son ejemplos **de como reordenar un array desordenado**, y son **un ejercicio muy bueno para aplicar la algoritmia** y **la recursividad**. Quien los implementa y los programa correctamente, mejora su nivel de programación.

7.1 Búsqueda Binaria. Practica guiada recursividad.

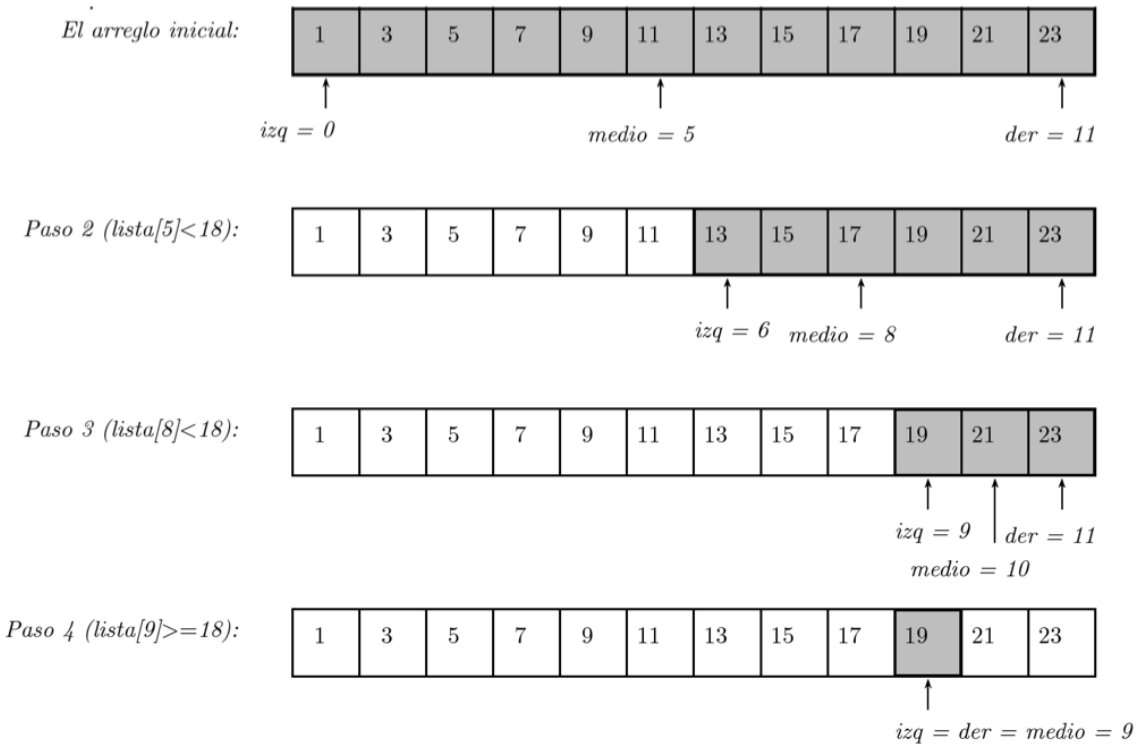
Dado un array ordenado array[] de n elementos, vamos a **implementar el algoritmo de búsqueda binaria de manera recursiva**. Lo que **se hace en este algoritmo es ir al punto medio del array**. Si el **elemento a buscar es mayor** que el **elemento que hay en el punto medio** se **busca en la parte derecha** del array. Si es **menor**, se **busca en la parte izquierda** del array. Si es igual, hemos encontrado el elemento. La solución de búsqueda binaria es dividir nuestro problema en un **problema más pequeño**. Empezamos buscando en el array entero, y en función del valor que haya a la mitad del array**, buscamos en la submitad izquierda o derecha** del array. Lo podéis ver en la siguiente figura que es bastante clarificadora.

Este algoritmo se usa porque es más eficiente en media que si hiciéramos una **búsqueda lineal**. En una **búsqueda lineal el tiempo medio es N/2**. En una **búsqueda binaria es $\log_2 N$**. Para arrays de tamaño grande se nota mucho la

diferencia. Por ejemplo para 1000, el tiempo medio de la **búsqueda lineal sería 500**. EL tiempo medio para la búsqueda binaria sería $\log_2 1000$ = **9.9**. Es decir, para un array de 1000 elementos el algoritmo de **búsqueda binaria es 25 veces más rápido**. Es **muy importante realizar algoritmos eficientes**, para mejorar nuestros programas.

Funcionamiento del algoritmo

En la siguiente imagen tenéis una ejecución del algoritmo conforme al ejemplo que os propociono con código java, `BusquedaBinariaRecursiva.java`. Paso a paso vamos a ir viendo el funcionamiento del mismo.



Paso 1. Estamos buscando el 19 en el array completo de la posición 0 a la 11. el punto medio es la mitad del array, 5 (0+ (11-0/2)= 0+5), **como el valor del array de punto medio es 11, y 19 es mayor**, buscamos **por la derecha**. Dividimos **el problema por la mitad** y buscamos en un número de elementos menor. El algoritmo realiza izq= medio +1.= 6, medio= izq + (der-izq/2) = 5+((11-5)/2)=5 + 3= 8.

Paso 2. Ahora izq es 6 y der 11. Estamos buscando **en la parte derecha el punto medio es 8**. El **valor 17 es menor que 19**. Volvemos a hacer la misma operación vamos a hacer **nuestra búsqueda más pequeña**, buscamos entre la posicion 9 y 11, **por la derecha**. El algoritmo realiza izq= medio +1.= 9, medio= izq + (der-izq/2)= 9 + ((11-9)/2) = 9 +1 =10.
Paso 3. Buscamos entre la **posición 9 y 11 del array**. El medio es 10, Y su **valor 21** que es mayor que el **elemento buscado 19**. Por tanto, tenemos que **seguir buscando a la izquierda en el array**. El algoritmo realiza der= medio - 1.= 9, medio= izq + (der-izq/2)= 9 + ((9-9)/2) = 9 +0=9.
Paso 4. Ahora **izq, der y medio señalan a la posición 9** del array. Tenemos **dos**

opciones. Que el contenido de la **posición 9, no sea 19.** Es o quiere decir que el elemento no está en el array, **devolvemos -1**. O que, si lo sea, y **devolvemos la posición 9**.

Resolución del algoritmo.

Lo resolvemos **de manera recursiva**
Nuestra llamada inicial,

```
= busqueda.busquedaBinaria(array1, 0, array1.length-1,19);
```

Buscamos entre la posición 0, y 11, la longitud del array 12 menos 1.
Lo primero es posicionar el valor medio del array

```
medio = izq + (der - izq) / 2;
```

Tenemos dos casos base en esta recursión:

1. Encontrando el elemento.

```
if (array[medio] == x)
                    return medio;
```

2. No encontrando el elemento.

Si der<izq, querra decir que hemos llegado al final del algoritmo y no hemos encontrado nada. La variable der debe ser mayor que izq.

```
    // Si no está devolvemos un -1
          return -1;
```

Tenemos dos posibles llamadas recursivas.

3. Si el valor de la posición medio es menor que la posición buscada buscamos por la izquierda con una llamada recursiva.

```
        if (array[medio] > x)
                    return busquedaBinaria(array, izq, medio - 1, x);
```

4. Si el valor de la posición medio es mayor que la posición buscada buscamos por la izquierda con una llamada recursiva.

```
          return busquedaBinaria(array, medio + 1, der, x);
```

BusquedaBinariaRecursiva.java

```
public class BusquedaBinariaRecursiva {
        public int busquedaBinaria (int array[], int izq, int der, int x)
        {
```

```
            int medio=0;
        if (der >= izq) {
                                medio = izq + (der - izq) / 2;

            // Si el elemento está presente en la posición medio

            if (array[medio] == x)
                return medio;


            //el elemento está a la izquierda del array
            if (array[medio] > x)
                 return busquedaBinaria(array, izq, medio - 1, x);

           //El elemento esta a la derecha del array

            else
              return busquedaBinaria(array, medio + 1, der, x);
        }

        // Si no está devolvemos un -1
        return -1;
    }

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

              int[] array1 = {1,3,5,7,9,11,13,15,17,19,21,23};

              BusquedaBinariaRecursiva busqueda= new
BusquedaBinariaRecursiva();
              int resultado = busqueda.busquedaBinaria(array1, 0,
array1.length-1,19);

              if  (resultado==-1) {


                       System.out.println("Numero 19 no encontrado en el
array");
              } else {


                       System.out.println("La posición que ocupa en el
array es: " +resultado);
              }

    }



}
```

7.2 Algoritmo de reordenación por inserción Directa. Practica independiente recursividad.

Dado el siguiente array:

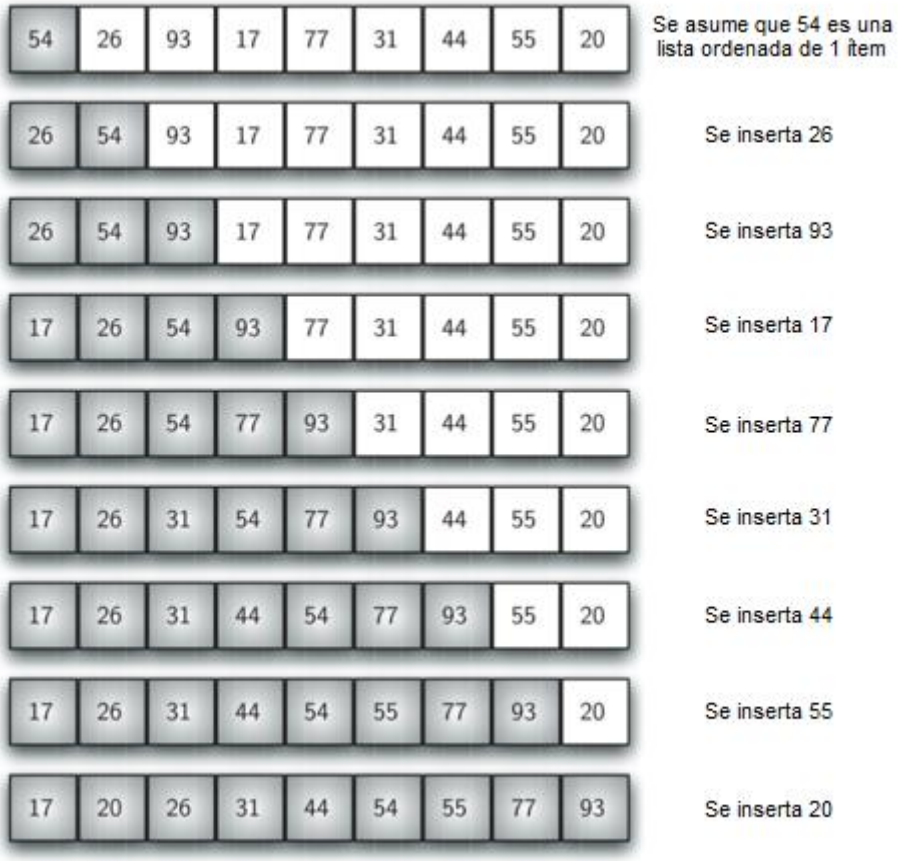
Dado el siguiente array de apellidos:

```
        public static final String apellidos[] = {"García", "González",
"Rodríguez", "Fernández", "López", "Martínez", "Sánchez", "Pérez", "Gómez",
                        "Martin", "Jiménez", "Ruiz", "Hernández", "Diaz",
"Moreno", "Muñoz", "Álvarez", "Romero", "Alonso", "Gutiérrez", "Navarro",
                        "Torres", "Domínguez", "Vázquez", "Ramos", "Gil",
"Ramírez", "Serrano", "Blanco", "Molina", "Morales", "Suarez", "Ortega",
                        "Delgado", "Castro", "Ortiz", "Rubio", "Marín",
"Sanz", "Núñez", "Iglesias", "Medina", "Garrido", "Cortes", "Castillo",
"Santos"};
```

1. Crear una clase ReordenaciónInserccion
2. Crear el método estático ordenacionPorInserccionIterativo que aplique el algoritmo explicado debajo
3. Crear el método estático ordenacionPorParticion para aplicar el algoritmo recursivo de ordenación de arrays por partición o QuickSort.

Algoritmo de ordenación por inserción

Vamos **a explicar brevemente este algoritmo**. Es un **algoritmo para ordenación** de arrays desordenados. La idea en este algoritmo es la de dividir el problema en problemas más pequeños como siempre.
Paso 1. Asumimos **que el array es sólo el primer elemento del array completo**, que **tiene una única posición y ordenamos**. Insertamos **54 en orden.**
Paso 2. Ordenar 54 el fácil, ya está ordenado. Seguimos considerando que el array tiene dos posiciones. Insertamos 26 en orden. **De el array considerado 54 26** tras la inserción directa obtenemos de resultado **26 54**.
Paso 3. Ordenamos **un array de 3 posiciones** esta vez, **insertamos 93 en orden**.
Paso 4. En el paso 4, insertamos 17l y ordenamos. Podéis ver el resultado en la imagen.
Y **así sucesivamente hasta que consideramos todo el array**, llegando a la **última posición del array**. Es un **algoritmo bastante sencillo** que deberéis hacer **vosotros mismos recursivamente**.



Como veis en **el último paso el array está ordenado**. Otra vez hay que indicar que este algoritmo **es más eficiente que una reordenación lineal**.

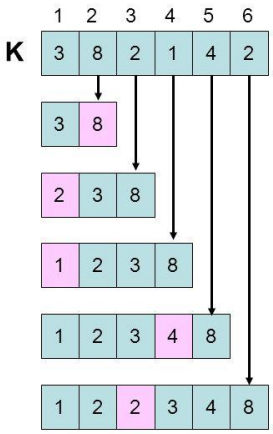
La **versión iterativa del algoritmo** sería la siguiente. Intentad **implementar ambas versiones, iterativa y recursiva**.

Ordenamiento por inserción directa

Variables
- K arreglo de datos a ordenar
- V variable auxiliar
- i, j índices para el arreglo
- N número de elementos

InserciónDirecta

```
Inicio
   Para i=2 hasta N incremento 1
     v = K(i)   //elemento a acomodar
      j = i
      Mientras (j > 1) y (K(j-1) > v)
           K(j) = K(j-1)  //mueve elementos
           j = j-1
     K(j) = v   // inserta el elemento actual
Fin
```



Método de ordenación por partición, o quicksort (método rápido).

Se trata de un algoritmo que inventó Hoare. Se toma aribitrariamente un elemento del array (denominado **pivote**, x). Se inspecciona el array de izquierda a derecha hasta encontrar una celda A[i]>x (puede ser que i sea mayor que la posición de x), y entonces se inspecciona el array de derecha a izquierda hasta encontrar una casilla A[j]<x. A continuación, se intercambian las dos celdas y se continua este proceso de inspección e intercambio hasta que los recorridos de ambas direcciones se encuentren en algún punto, y se coloca el pivote ahí, intercambiándolo con la casilla que está en dicha posición.

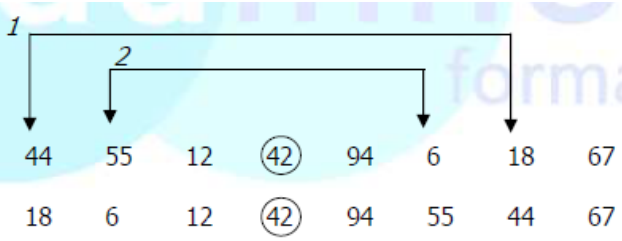


Figura 11.8. Algoritmo quicksort con pivote en el número 42. Se han marcado los pasos 1 y 2 en cursiva. En la parte inferior se observa el resultado de la primera iteración

Escogiendo una posición distinta del pivote se llega a un resultado distinto al final de la primera iteración, aunque después de completar el algoritmo el resultado será el mismo:

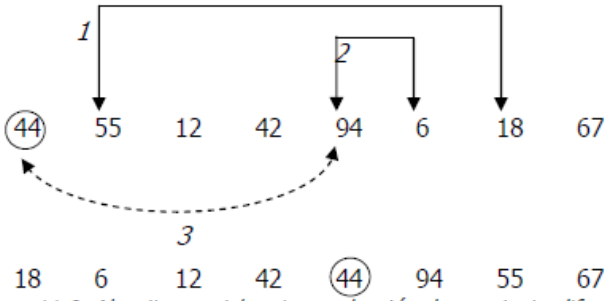


Figura 11.9. Algoritmo quicksort con elección de un pivote diferente

```
   QUICKSORT (A, a, b)
1. SI a<b
2.       k ← PARTICIÓN (A, a, b)
3.       QUICKSORT (A, a, k-1)
4.       QUICKSORT (A, k+1, b)
```

Algoritmo 11.3. Quicksort

Donde la función de partición es:

```
1.    i ← a                  //Posición inicial
2.    i ← b             //Posición final
3.    {ELECCIÓN ALEATORIA DE UNA POSICIÓN X COMO PIVOTE}
                         //Por ejemplo, el
                         //punto medio: x = ⌊(a+b)/2⌋;
                         //ó un extremo: x = a
4.    MIENTRAS i ≤ j HACER
5.          MIENTRAS A[i] < A[x] HACER
6.                i ← i + 1
7.          MIENTRAS A[j] > A[x] HACER
8.                j ← j-1
9.          SI i ≤ j ENTONCES
10.               INTERCAMBIAR A[i] CON A[j]
11.   INTERCAMBIAR A[j] CON A[i]    //Mover  el  pivote  a  la  posición
      adecuada
      DEVOLVER j                    //Se devuelve la posción del pivote
12.
```

8 Recursividad con la API Stream y expresiones lambda. Actividad de ampliación

A lo **largo de este último punto del tema vamos a desarrollar como usar la API Stream** y las **expresiones lambda para crear soluciones seudorecursivas y recursivas** a **problemas computacionales.** Es el **punto clave en programación funcional** que intentaré **dilucidar con ejemplos sencillas. Empezaremos usando la API Stream** y el **método reduce** para dar **una solución seudorecursiva**, ya que **el método reduce se llama a si mismo continuamente** para cada **elemento del Stream.** Posteriormente, realizaremos **recursividad utilizando el concepto de funciones de orden superior**, en resumen,

mediante funciones recursivas y expresiones lambda que se llaman a si mismas.

8.1 Seudorecursividad usando reduce e IntStream

En este punto **vamos a iterar sobre un IntStream** para poder **calcular el factorial del número n** pasado como **parámetro**. Esta **solución no es la óptima**. Para poder usar una variable **dentro de una expresión lambda el ámbito de esta variable** debe de **ser de clase o de programa principal**. Esto quiere decir que lo primero que tengo que hacer es **declarar un atributo para la clase**, llamado **private int resultadofinal**. Esta solución por tanto **rompe con el concepto de función sin estado**. Aunque la **variable sólo la estamos usando para ese método**, pero no es lo más correcto. Sin embargo, los **programadores lo hacen habitualmente** porque son **muy cómodos los Streams para programar**.

Lo segundo es declarar un IntStream que vaya de 1 a n, con rangeClosed, `IntStream.rangeClosed(1, n).`

Y por último, y ya vimos que la función del **método reduce** es **reducir el Stream** a un solo objeto, en este caso **un número** `reduce((resulParcial,i) -> resulParcial*i);`
Para **cada elemento del Stream se realiza un reduce de resulParcial**, donde guardamos los resultados parciales, y un elemento del Stream. Por ejemplo, **la ejecución para factorial(3)** funcionaría como sigue:
Sería range(1,3)

El primer **elemento del Stream el 1,**
reduce(1,1) -> 1*1=”

El segundo elemento del Stream el 2
reduce(1,2)-> 1*2=2

El **tercer elemento del Stream el 3**
reduce(2,3)-> 2*3 =6

Y así **obtendríamos nuestro resultado final** 6. El método **reduce se ejecuta para cada elemento del Stream**, en este caso **hay 3**, 1,2 y 3. Y en **el primer parámetro del reduce, se guardan los resultados parciales** y se **vuelve a usar**. Esto ya lo **hemos visto pero con cadenas.**

```
OptionalInt resultado = IntStream.rangeClosed(1, n).reduce((resulParcial,i) -
> resulParcial*i);
```

FactorialConReduce.java

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

```
public class FactorialConReduce {

        private int resultadofinal;

        public int factorial(int n) {


                OptionalInt resultado = IntStream.rangeClosed(1,
n).reduce((resulParcial,i) -> resulParcial*i);

                resultado.ifPresent((fact)-> resultadofinal=fact);



                resultadofinal = resultado.orElse(1);


                return resultadofinal;
        }

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


                 FactorialConReduce claseFact = new FactorialConReduce();
                        Integer numero1;
                        Scanner miScanner = new Scanner(System.in);


                    System.out.println("Escriba un número entero");
                    numero1 = miScanner.nextInt();

                  System.out.println("El factoria para el número " +numero1
+ " es " + claseFact.factorial(numero1));


         }

}
```

8.2 Factorial recursivo con función de orden superior.

Es la manera correcta de **realizar el programa en programación funcional**, con una **función de orden superior y sin estado**. Lo realizaremos con **recursión directa.**

Voy a desarrollarlo igualmente con un ejemplo. Tenemos una función factorial, que recibe un **UnaryOperator** un tipo de función que **recibe un parámetro**, en este caso **Long y devuelve un Long igualmente.** Ya la hemos desarrollado en el curso.

```
public UnaryOperator<Long> factorial(UnaryOperator<Long> res,
                Long numero) {
```

Para hacer **la recursión en vez de números** tanto en **el caso base** como en el resto de los casos debemos devolver **un interfaz funcional**, una función ya no trabajamos con parámetros simples, devolvemos una expresión lambda.

Lo primero **nuestro caso base es** o debe ser una función que devuelva un uno Long.
`(n)->1L`
Si realizamos **una llamada al método apply** de nuestro caso base **nos devolverá un 1** que es lo que devuelve esa expresión lambda.
Nuestro caso base como en la recursión normal:

```
if (numero==1) {

                 return ((n)->1L);
               }
```

Para el resto de casos tenemos que realizar una llamada que **divida el problema** en uno **más pequeño,** como en la recursión normal:
Dentro de **nuestra expresion lambda resultado llamamos a la función factorial**, y **llamamos al método apply** para que nos devuelva un resultado, pero si os fijais hemos hecho el problema más pequeño, **llamando a la misma función, pero con i-1.** Igualmente el apply(i-1), es más pequeño. Llamamos a apply porque dentro de la Expresión lambda debemos **devolver un Long, un número, no otra expresión lambda**. En el **return si devolvemos una expresión lambda**, un **UnaryOperator** return (i->i*factorial(res,i-1).apply(i-1));
`factorial(res,i-1).apply(i-1));`

```
return (i->i*factorial( res,i-1).apply(i-1));
```

Vamos a simular una ejecución:
Paso 1. factorial(res,3)
En la primera llamada devolvemos con el return la expresión lambda 3->3*factorial(res, 2)`.apply(2)`
Paso 2. factorial (res,2) devolvemos 2*factorial(res, 1).apply(1)

Paso 3. En factorial(res,1) hemos llegado al devolvemos ((n)->1).

Paso 4. Volvemos a factorial 2 y se aplicará 2*((n)->1).apply() . El apply del UnaryOperator n->1, devuelve un 1. Se resuelve factorial 2 con 2*1 =2.

Paso 5. Y finalmente volvemos a factorial(3) que devuelve 3*factorial(res,2),.apply(2). Ya hemos visto en el paso anterior que factorial(res,2).apply(2) devolvía 2. Con lo que el **resultado final es 3*2=6.**
Importante:
Como ahora **trabajamos con funciones como parámetros**, debemos **pasar una función para empezar a Factorial**. Debe ser **una neutra o inocua**, como **si estuviéramos multiplicando por 1**, en este caso **la función identidad, x->x.**

```
classFact.factorial(x->x ,numero1).apply( numero1);
```

```
import java.util.OptionalInt;
import java.util.Scanner;
import java.util.function.UnaryOperator;
import java.util.stream.IntStream;

public class RecursividadFuncionOrdenSuperior {


public UnaryOperator<Long> factorial(UnaryOperator<Long> res,
                Long numero) {

                if (numero==1) {

                 return ((n)->1L);
                } else {


                        return (i->i*factorial( res,i-1).apply(i-1));

                }


        }

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


                 RecursividadFuncionOrdenSuperior classFact = new
RecursividadFuncionOrdenSuperior();
                 Long numero1;

                 Scanner myScanner = new Scanner(System.in);

                    System.out.println("Teclea un número entero");
                    numero1 = myScanner.nextLong();


                    // I cast to double to get a wider range of results for
the factorial function

                    Long resultado = classFact.factorial(x->x,
numero1).apply( numero1);

                  System.out.println("El factorial del numero: " +numero1 +
" es " +resultado);


         }

}
```

8.3 *Factorial recursivo con función de orden superior con el método identity. Versión mas completa*

Esta versión es igual al anterior. **Igual que antes esta parte de los apuntes no entra en el curso** es para quien quiera indagar. Sólo cambia que usamos el **método identity del UnaryOperator**. En subrayado los cambios, **investigad porque funciona igualmente**.

```
import java.util.OptionalInt;
import java.util.Scanner;
import java.util.function.UnaryOperator;
import java.util.stream.IntStream;

public class recursionHighOrderFunctionIdentidade {


public UnaryOperator<Double> factorial(UnaryOperator<Double> res,
                Double numero) {

                if (numero==1) {

                 return (res);
                } else {


                        return (i->i*factorial( res,i-1).apply(i-1));

                }


        }

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


                  recursionHighOrderFunctionIdentidade classFact = new
recursionHighOrderFunctionIdentidade();
                  Integer numero1;

                  Scanner myScanner = new Scanner(System.in);

                     System.out.println("Introduce un número entero");
                     numero1 = myScanner.nextInt();
                     Double numeroUnoDoble = Double.valueOf(numero1);



                     Double resultado =
classFact.factorial(UnaryOperator.identity(), numeroUnoDoble).apply((Double)
numeroUnoDoble);

                   System.out.println("El resultado del factorial del numero
" +numero1 + " es " + resultado);
```

```
        }
}
```

9 Bibliografía y referencias web

Referencias web

Tutoriales de Java Jacob Jenkov

http://tutorials.jenkov.com/

Tutoriales de Java arquitectura Java

https://www.arquitecturajava.com/

Repositorio Github Venkat Subramanian

https://github.com/venkats

Tutoriales Java geeksforgeeks

https://www.geeksforgeeks.org/

Tutoriales Java Baeldung

https://www.baeldung.com/

Bibliografía

Functional Programming in Java: Harnessing the Power Of Java 8 Lambda Expressions, Venkat Subramanian, The Pragmatic Programmers, 2014

Programación, Alfonso Jímenez Pérez, Francisco Manuel Pérez Montes, Paraninfo, 1ª edición, 2021