

# Unidad 11. Java 8. Colecciones y la API Stream

## Contenido

1	Actividad inicial .....	2
2	Introducción .....	4
3	Programación Stream .....	4
4	El operador ::. Actividad de refuerzo .....	5
5	La clase Optional .....	11
5.1.1	Manejo de Optionals llenos y vacíos con map .....	13
5.1.2	Practica independiente clase Optional. ....	15
6	La API Stream para el manejo de colecciones en java.....	16
1.1	La API Stream. Clases .....	16
6.1	Colecciones y la API Stream. Actividad guiada transformación de arrays a colecciones .....	17
6.1.1	El método forEach .....	19
6.2	Peculiaridades de Map en java 8.....	21
6.3	Colecciones Inmutables y Collections .....	23
6.3.1	Collectors.toUnmodifiableMap() .....	23
6.4	Actividad independiente. Listas inmutables .....	27
6.5	La clase Collectors y el método de Stream collect() .....	28
6.6	Como funciona la API Stream .....	31
6.7	Tipos de Operaciones con Streams. Repaso aplicado a colecciones. Actividad guiada Operaciones no terminales .....	31
6.7.1	Practica independiente de operaciones no terminales .....	34
6.7.2	Actividad guiada generación de clases aleatorias para depuración. ....	34
6.7.3	Actividad independiente generación de clases aleatorias para depuración. ....	38
6.8	Operaciones no terminales .....	39
6.8.1	filter() .....	40
6.8.2	map() .....	41
6.8.3	De lista de String a Map de Empleados .....	44
6.8.4	Mapeando a un tipo básico mapToInt, mapToDouble, etc. Practica guiada mapeo tipos básicos .....	46
6.8.5	flatMap() .....	48
6.8.6	distinct() .....	51
6.8.7	sorted().....	51
6.8.8	limit() .....	53
6.8.9	peek() .....	53
6.9	Operaciones Terminales . Practica Guiada.....	54
1.1.1	anyMatch().....	54
1.1.2	allMatch() .....	54
1.1.3	noneMatch() .....	55
1.1.4	collect() .....	56
1.1.5	count() .....	56

1.1.6	findAny() y findFirst() .....	58
6.9.1	max() y min() .....	60
6.9.2	Manejando Optionals con el método Map .....	62
6.9.3	reduce() .....	63
6.9.4	Practica independiente de operaciones terminales .....	65
6.9.5	forEach() y toArray() .....	65
6.10	IntStream, Double Stream, LongStream .....	65
6.10.1	IntStream.of() .....	66
6.10.2	IntStream.iterate() .....	66
6.10.3	IntStream.generate() .....	66
6.10.4	IntStream range() .....	67
7	range(int startInclusive, int endExclusive) – Devuelve un Stream de tipo int que comienza con startInclusive( <i>incluido</i> ) y termina endExclusive (no incluido) con un incremento de una unidad. ....	67
8	rangeClosed(int startInclusive, int endInclusive) – Devuelve Stream de tipo int ordenado que comienza en startInclusive( <i>incluido</i> ) y termina endInclusive ( <i>no incluido</i> ) con un incremento de una unidad. ....	67
8.1.1	map() y mapToObject(). Practica guiada IntStream .....	68
8.1.2	Practica independiente IntStream .....	71
8.1.3	IntStream y funciones de agregación. Practica guiada .....	71
8.1.4	Practica independiente de funciones de agregación .....	73
8.2	Ampliación de la clase collectors. ....	74
8.2.1	Collectors.collectingAndThen() .....	74
8.2.2	Operaciones de particionamiento, agrupamiento y estadísticas con Collectors. ....	78
8.3	Concatenando Streams .....	86
8.4	Stream paralelos .....	87
9	El operador ... ..	87
10	Composición avanzada. Uso de Interfaces como parámetros .....	90
10.1	Interfaces como parámetros .....	90
10.2	Recorriendo los parámetros interfaz funcional del operador ... con un for. ....	92
10.3	Recorriendo los parámetros interfaz funcional del operador ... con un Stream. Practica de ampliacion .....	94
10.3.1	Refactorizando el ejemplo anterior. Practica de ampliación ...	96
11	Bibliografía y referencias web .....	97

## 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 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 resumen, se ha optimizado los procesadores con múltiples núcleos 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 El operador ::. Actividad de refuerzo

Para terminar de explicar el operador:: necesitamos realizar un ejemplo completo de uso. Se usa mucho en la actualidad en asincronía en Java, todas las aplicaciones web java lo pueden usar en substitución de callbacks. Me permite pasar una función como parámetro. Se está empezando a usar en aplicaciones móviles Android. Puedo pasar mi función a un método de otra clase, donde será ejecutado.

Lo mejor es mostrárselo con un ejemplo. Vamos a crear una clase de nombre Clase. Y vamos a mostrar todas las posibilidades de como pasar una expresión lambda, y con el operador :: una función estática y una no estática. En la misma clase, o pasándolo a otra clase. Este ejemplo incluye dos clases.

Clase.java y OtraClase.java. Montarlo antes de leer los apuntes.

## Funciones estáticas y lambdas

Definimos en Clase una función estática `funcionConParametroFunction`. Y vamos a llamarla desde Main. Recordar que Main es estática.

```
public static void funcionConParametroFunction(String param,
Function<String,String> funcion ) {

    System.out.println(funcion.apply(param));

}
```

Definimos también un función Estática en Clase, que pasaremos a la anterior como parámetro usando el operador `::`. `miFuncionEstatica` cumple con el interfaz `Function<String,String>`, recibe un `String` y devuelve un `String`.

```
public static String miFuncionEstatica(String param) {

    return param + " funcion ciudadana de primer nivel en Java";

}
```

Fijaos como pasamos a esta función estática un `Function`. En el primer caso como lambda, en el segundo una función estática `miFuncionEstatica` definida también en la clase Clase.

```
Clase.funcionConParametroFunction("Cadena", (s)-> s + " " +s );

Clase.funcionConParametroFunction("Funcion normal",
Clase::miFuncionEstatica);
```

## Funciones no estáticas

Vamos a ampliar lo anterior usando funciones no estáticas con el operador dos puntos como parámetro. Definimos una función no estática que recibe un `Function`.

```
public void funcionConParametroFunctionNoEstatica(String param,
Function<String,String> funcion ) {

    System.out.println(funcion.apply(param));

}
```

Definimos una función no estática que cumple con el interfaz funcional `Function<String,String>`, recibe un `String` y devuelve un `String`. La vamos a usar para pasarla como parámetro.

```
public String miFuncionNoEstatica(String param) {

    return param + " funcion ciudadana de primer nivel en Java";

}
```

En primer lugar vamos a hacerlo desde Main. Creo un objeto de Clase, llamado c1. Y llamo a la función `funcionConParametroFunctionNoEstatica` g, pasandole una función de mi objeto de tipo Clase c1, no estática, no función estática de Clase. Para ello uso la variable objeto c1 seguido de dos puntos y el nombre de la función. `c1::miFuncionNoEstatica`. Ya no usamos el `Clase::function`, esto vale para funciones estáticas, para no estáticas usamos el nombre del objeto y la función.

```
Clase c1 = new Clase();

c1.funcionConParametroFunctionNoEstatica("Funcion no estatica",
c1::miFuncionNoEstatica);
```

Vamos a hacerlo ahora dentro de mi clase Clase. Definimos en Clase otra función que llama a esta, pasando `funcionAMiClase`. Quiero que os fijéis en este punto que para pasar como parámetro la función ya no uso el nombre de la clase `Clase::`. Esto sólo sirve para funciones estáticas. Uso `this`, que es un puntero al objeto de tipo Clase c1, que he creado en Main, que es como se hace referencia a un objeto dentro de si mismo. Dentro del mismo objeto no puedo hacer `c1::`, no tengo la variable del objeto, con lo que para acceder o referenciarse a si mismo uso `this`.

```
public void pasandoFuncionAMiClase(String cadena){

    funcionConParametroFunctionNoEstatica(cadena, this::miFuncionNoEstatica);

}
```

```
Clase c1 = new Clase();

c1.pasandoFuncionAMiClase("No estatica en mi clase");
```

## Pasando funciones a otra clase

Tenemos otra clase en el ejemplo llamada `OtraClase.java`. Dentro de ella hemos definido una función que recibe como parámetro un `Function<String,String>` llamada no estática

```
public void noEstatica (Integer i, Function<Integer,Integer> funcion) {
```

```

        System.out.println(funcion.apply(i));
    }

```

Creo un objeto de tipo `OtraClase` en `Main` y le paso una función del objeto `c1` de tipo `Clase` `c1::miFuncionNoEstatica` a `otra.funcionDePruebaNoEstatica`. Y como veis puedo pasarle métodos de un objeto de una clase a una función de un objeto de otra clase distinta. Es la potencia de que las funciones sean ciudadanos de primera clase.

```

OtraClase otra = new OtraClase();

otra.funcionDePruebaNoEstatica("Funcion otra clase",
c1::miFuncionNoEstatica);

```

Para finalizar estudiad que estoy haciendo en este código.

```

c1.pasandoFuncionAOtraClase("No estatica en otra clase");

```

## Clase.java

```

import java.util.function.Function;

/**
 *
 * @author carlo
 */
public class Clase {

    public static void funcionConParametroFunction(String param,
Function<String,String> funcion ) {

        System.out.println(funcion.apply(param));
    }

    public void funcionConParametroFunctionNoEstatica(String param,
Function<String,String> funcion ) {

        System.out.println(funcion.apply(param));
    }

    public static String miFuncionEstatica(String param) {

```



```

        return param + " funcion ciudadana de primer nivel en Java";
    }

    public String miFuncionNoEstatica(String param) {

        return param + " funcion ciudadana de primer nivel en Java";
    }

    public void pasandoFuncionAMiClase(String cadena){

funcionConParametroFunctionNoEstatica(cadena, this::miFuncionNoEstatica);

    }

    public void pasandoFuncionAOtraClase(String cadena){

        OtraClase otra = new OtraClase();

        otra.funcionDePruebaNoEstatica(cadena, this::miFuncionNoEstatica);
    }


    public Clase () {

    }


    public static void main(String[] args) {

        Clase.funcionConParametroFunction("Cadena", (s)-> s + " " +s );

        Clase.funcionConParametroFunction("Funcion normal",
Clase::miFuncionEstatica);

        Clase c1 = new Clase();

        c1.funcionConParametroFunctionNoEstatica("Funcion no estatica",
c1::miFuncionNoEstatica);
    }

```

```

        c1.pasandoFuncionAMiClase("No estatica en mi clase");
        OtraClase otra = new OtraClase();

        otra.funcionDePruebaNoEstatica("Funcion otra clase",
c1::miFuncionNoEstatica);

        c1.pasandoFuncionAOtraClase("No estatica en otra clase");

    }
}

```

### OtraClase.java

```

import java.util.function.Function;

/**
 *
 * @author carlo
 */
public class OtraClase {

    public OtraClase() {

    }

    public void funcionDePruebaNoEstatica(String param,
Function<String,String> funcion ) {

        System.out.println(funcion.apply(param));

    }

    public void noEstatica (Integer i, Function<Integer,Integer> funcion) {

        System.out.println(funcion.apply(i));

    }

}

```

Se usa mucho este tipo de estructuras con el patron Strategy que intentaremos ver en este tema.

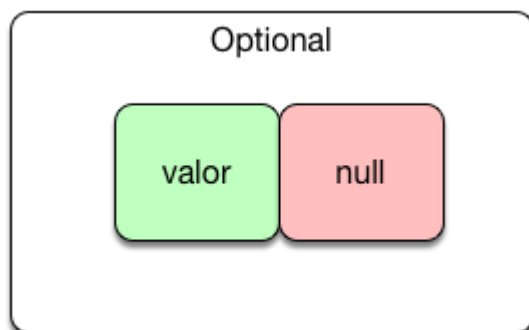
## 5 La clase Optional

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 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 contiene 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 **OptionalInt**, **OptionalDouble**, **OptionalLong**, etc. Tenéis un ejemplo en los métodos `max()` y `min()` que veremos posteriormente.

## 5.1 Manejo de Optionals llenos y vacíos con map

Otra opción con la clase **Optional** es usar el método `map`, con las diferentes opciones de **OrElse** que nos ofrece la combinación con el método de `map`. Lo podemos ver de manera sencilla con un ejemplo.

Aplicando el método `map()` de la clase **Optional** podemos extraer el empleado si el **Optional** está lleno. En caso de estar vacío se ejecutará el método `orElseGet` que recibe un supplier creando un **Empleado** vacío.

```

Optional<Empleado2> empl1 = Optional.of(new Empleado2(5, "Carlos Lopez"));
Optional<Empleado2> empl2 = Optional.empty();

Empleado2 empleado = empl1.map(emp-> emp).orElseGet(()-> new Empleado2());

```

También tenemos la opción en el caso de que el **Optional** este vacío de lanzar una excepción. En el siguiente ejemplo realizamos un `map` para obtener el

optional pero si viene vacío se lanzará una excepción con el método `orElseThrow()`, que recibe un **supplier** que genera la excepción a lanzar. Nótese que debemos en este caso envolver la instrucción con un try catch.

```
try {
    Empleado2 empleado2 = empl2.map(emp->emp).orElseThrow(() -> new
Exception("Empleado Optional vacío"));
} catch (Exception e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```

Como el Optional `empl2` está vacío se lanzará de manera efectiva la excepción en la ejecución del programa, dando como resultado de la ejecución en consola:

```
java.lang.Exception: Empleado Optional vacío
    at
Proyecto.Unidad11.optional.EjemploOptionalMap.lambda$3(EjemploOptionalMap.java:62)
    at java.base/java.util.Optional.orElseThrow(Optional.java:401)
    at
Proyecto.Unidad11.optional.EjemploOptionalMap.main(EjemploOptionalMap.java:62)
)
```

## EjemploOptionalMap.java

```
import java.util.Optional;

class Empleado2 {

    int id;
    String nombre;

    public Empleado2() {

    }

    public Empleado2 (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 EjemploOptionalMap {

    public static void main(String[] args) {

        Optional<Empleado2> empl1 = Optional.of(new Empleado2(5,"Carlos
Lopez"));
        Optional<Empleado2> empl2 = Optional.empty();

        Empleado2 empleado = empl1.map(emp-> emp).orElseGet(()-> new
Empleado2());

        try {
            Empleado2 empleado2 = empl2.map(emp->emp).orElseThrow(()-> new
Exception("Empleado Optional vacio"));
        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}

```

## 5.2 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.

## 6 La API Stream para el manejo de colecciones 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 características de los procesadores, para el manejo y manipulación de secuencias de objetos, Streams de objetos.

Vamos a poder manejar colecciones de objetos, como las clases que implementan los interfaces List, y Set, y por tanto el interfaz Collection e Iterable, como ArrayList, HashMap, HashSet, LinkedHashSet, LinkedList, TreeSet, EnumSet etc con la API Streams igual que hacíamos con los Arrays. La verdadera potencia de la API Streams se puede apreciar en el tratamiento de colecciones.

Básicamente vamos a ofrecer los mismos ejemplos que en el tema de arrays para repasar la API, y algunos nuevos, y además vamos a añadir una clase nueva de vital importancia, **Collectors**, que nos permitirá el tratamiento y transformación de Stream en colecciones para recoger los resultados. Es recomendable que conozcáis el interfaz Iterator y también Iterable para el tratamiento estándar de colecciones.

En el siguiente enlace a la documentación de Oracle podéis ver cuantas clases Java implementan Collection, son colecciones:

<https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html>

### 1.1 La API Stream. Clases

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 <b>operación de reducción mutable</b> 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 <b>DoubleStream</b> .
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.

## 6.1 Colecciones y la API Stream. Actividad guiada transformación de arrays a colecciones

Lo primero de todo vamos a ver es como podemos generar un Stream a partir de diferentes colecciones. Lo haremos con un ejemplo llamado StreamColecciones.java. Como podéis apreciar tanto para List, ArrayList, LinkedList, TreeSet, y HashSet, disponemos del método stream que nos devuelve directamente un Stream a partir de la colección, no necesitamos ninguna clase intermedia como en Arrays para generar el Stream. Igualmente, todas estas colecciones se pueden llenar usando otras, con el método addAll.

En el ejemplo genero una lista con un conjunto de nombres y el método de Arrays.asList.

A partir de ahí se rellenan el resto de estas colecciones usando la lista y el método `listLinked.addAll(listaNombres)`

```
List<String> listaNombre = Arrays.asList("borito", "Antonio", "Len", "Titus",
"Alejandro", "Aitor", "Sarika", "amanda", "Hans", "Shivika", "Sarah",
"Julius");
```

```

        System.out.println("Imprimimos la lista de nombres");

        listaNombre.stream().forEach(System.out::println);

        LinkedList<String> listLinked = new LinkedList<String>();

        listLinked.addAll(listaNombre);

```

La única colección que se comporta diferente es el `HashMap<k,v>`, porque tiene un par clave valor como tipo de entrada.

Por eso la rellenamos diferente, en este caso con un `IntStream`, para generar una clave en este caso numérica para cada nombre. Igualmente para generar un `Stream` desde un `Map` necesitamos llamar al método `entrySet`, que nos devuelve una colección de objetos tipo `Map.Entry<K,V>`, clave valor, y que imprimimos como podéis ver. De esta realizaremos más adelante algún ejemplo aparte. A partir del método `map.entrySet().stream()` podemos generar el stream.

```

HashMap<Integer,String> map = new HashMap<Integer, String>() ;

        IntStream.range(0,listaNombre.size()).forEach(i ->
map.put(i,listaNombre.get(i)));
        System.out.println("Imprimimos un Hash Map de key enteros,
valores string nombres");

        map.entrySet().stream().forEach((e)->System.out.println(e));

```

## StreamColecciones.java

```

import java.util.Arrays;
import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Set;
import java.util.TreeSet;
import java.util.stream.IntStream;

public class StreamColecciones {

    public static void main(String[] args) {

        List<String> listaNombre = Arrays.asList("borito", "Antonio",
"Len", "Titus", "Alejandro", "Aitor", "Sarika", "amanda", "Hans", "Shivika",
"Sarah", "Julius");

        System.out.println("Imprimimos la lista de nombres");

```

```

        listaNombre.stream().forEach(System.out::println);

        LinkedList<String> listLinked = new LinkedList<String>();
        listLinked.addAll(listaNombre);

        System.out.println("Imprimimos la lista linked de nombres");
        listLinked.stream().forEach(System.out::println);

        TreeSet<String> tree = new TreeSet<String>() ;
        tree.addAll(listaNombre);
        System.out.println("Imprimimos una arbol de nombres");
        tree.stream().forEach(System.out::println);

        HashSet<String> set = new HashSet<String>() ;
        set.addAll(listaNombre);
        set.stream().peek((e)->System.out.println(e));

        HashMap<Integer,String> map = new HashMap<Integer, String>()
;

        IntStream.range(0,listaNombre.size()).forEach(i ->
map.put(i,listaNombre.get(i)));
        System.out.println("Imprimimos un Hash Map de key enteros,
valores string nombres");
        map.entrySet().stream().forEach((e)->System.out.println(e));

    }
}

```

### 6.1.1 El método forEach

Todas las colecciones en java 8 aportan el método ya conocido `forEach` (`Consumer`<? super `T`> action) similar al de Stream, recibe un Consumer, para realizar una acción sobre cada objeto, donde T es el tipo de

**objeto de la colección. Se implementa y proviene del interfaz Iterable, podéis buscarlo en la documentación oficial de Oracle. Funciona exactamente igual que el forEach de Streams.**

**Vamos a verlo con un ejemplo, es bastante sencillo, no nos detendremos a explicarlo en detalle, se entiende por si mismo.**

```
List<String> listaNombre = Arrays.asList("borito", "Antonio", "Len", "Titus",
"Alejandro", "Aitor", "Sarika", "amanda", "Hans", "Shivika", "Sarah",
"Julius");

System.out.println("Imprimimos la lista de nombres");

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

LinkedList<String> listLinked = new LinkedList<String>();
listLinked.addAll(listaNombre);
System.out.println("Imprimimos la lista linked de nombres");
listLinked.forEach(System.out::println);

TreeSet<String> tree = new TreeSet<String>();
tree.addAll(listaNombre);
System.out.println("Imprimimos una arbol de nombres");
tree.forEach(System.out::println);

HashSet<String> set = new HashSet<String>();
set.addAll(listaNombre);
set.forEach((e)->System.out.println(e));

HashMap<Integer,String> map = new HashMap<Integer, String>();

IntStream.range(0,listaNombre.size()).forEach(i ->
map.put(i,listaNombre.get(i)));
System.out.println("Imprimimos un Hash Map de key enteros,
valores string nombres");
map.entrySet().forEach((e)->System.out.println(e));
```

## 6.2 Peculiaridades de Map en java 8

Map tiene dos métodos de los que se obtiene colecciones, `values()` te devuelve una colección con los valores del Map, y `keySet()` te devuelve el conjunto de claves del Map. Podemos obtener un Stream a partir de estos dos.

HashMap y TreeMap nos aporta tres métodos `compute`, para tratar las entradas de tipo Entry del HashMap con un Bifunction y que son propias del interfaz Map de Java.

default <u>V</u>	<u>compute</u> ( <u>K</u> key, <u>BiFunction</u> <? super <u>K</u> ,? super <u>V</u> ,? extends <u>V</u> > remappingFunction) Intenta realizar una operacion sobre una entrada en nuestro Map. Si el valor para la entrada no está en el Map, opera sobre null
default <u>V</u>	<u>computeIfAbsent</u> ( <u>K</u> key, <u>Function</u> <? super <u>K</u> ,? extends <u>V</u> > mappingFunction) Si la entrada no está en el Map, opera sobre ella y la introduce en el Map
default <u>V</u>	<u>computeIfPresent</u> ( <u>K</u> key, <u>BiFunction</u> <? super <u>K</u> ,? super <u>V</u> ,? extends <u>V</u> > remappingFunction) Si el valor para la clave esta presente y no es nulo, opera sobre el

Lo veremos en el siguiente ejemplo, `MapJava.java`.

Comprobar en la ejecución que el valor de 1 , existe y `compute()` añade al nombre Antonio Computado. Como veis recibe un Bifunction y que añade un nuevo valor a la entrada cuya clave es 1. Podríamos cambiar el nombre entero de la clave también, no añadir computado. El resultado del Bifunction se añade a la entrada 1 del Map.

```
map.compute(1, (k,v)-> v.concat(" Computado"));
```

El método `computeIfPresent` realiza lo mismo, pero sólo si la entrada está presente. Como 3 esta presente, se realiza.

```
map.computeIfPresent(3, (k,v)-> v.concat(" Computado si presente"));
```

Por último, `computeIfAbsent`, añade la entrada computada si no existe, nos añadiría 12, Gorgito a nuestro HashMap.

```
map.computeIfAbsent(12, v->"Gorgito");
```

Puedo igualmente con map obtener un Stream a partir del método `values()` que me devuelve una colección con los valores del Map, o con el método `keySet()` que me devuelve el conjunto de claves.

```
map.values().stream().forEach((e)->System.out.println(e));  
map.keySet().stream().forEach((e)->System.out.println(e));
```

```

import java.util.Arrays;
import java.util.HashMap;

import java.util.List;
import java.util.stream.IntStream;

public class MapJava {

    public static void main(String[] args) {

        List<String> listaNombre = Arrays.asList("borito", "Antonio",
"Len", "Titus", "Alejandro", "Aitor" ,"Sarika", "amanda", "Hans", "Shivika",
"Sarah", "Julius");

        HashMap<Integer,String> map = new HashMap<Integer, String>()
;

        IntStream.range(0,listaNombre.size()).forEach(i            ->
map.put(i,listaNombre.get(i)));

        map.entrySet().stream().forEach((e)->System.out.println(e));

        map.compute(1, (k,v)-> v.concat(" Computado"));

        map.computeIfPresent(3, (k,v)-> v.concat(" Computado si
presente"));

        map.entrySet().stream().forEach((e)->System.out.println(e));

        map.computeIfAbsent(12, v->"Gorgito");
        map.computeIfAbsent(14, v->"Jujito");

        map.entrySet().stream().forEach((e)->System.out.println(e));

        map.values().stream().forEach((e)->System.out.println(e));
        map.keySet().stream().forEach((e)->System.out.println(e));

    }

}

```

## 6.3 Colecciones Inmutables y Collections

Con la llegada de Java 10 y la programación funcional, java ofrece colecciones inmutables. Son similares a las colecciones normales con las que habéis trabajado con la única característica extra, de que no se pueden modificar, se mantienen constantes, siguiendo el patrón que vimos en el tema 3, objeto inmutable.

Ejemplos de estas colecciones son `unmodifiableList`, `unmodifiableMap`, `unmodifiableSet`, y todas sus variantes. Mediante métodos estáticos de `Collections`, podemos crear estas colecciones inmutables, llamando a los métodos `unmodifiable`. Como veis se crean a través de la factoria de `Collections`, no se crean las clases directamente con un `new`. Estas colecciones pertenecen al Java Core, es decir, a las librerías JRE que tenemos agregadas a nuestros proyectos.

```
List<String> stringList = Arrays.asList("a", "b", "c");
stringList = Collections.unmodifiableList(stringList);

Set<String> stringSet = new HashSet<>(Arrays.asList("a", "b", "c"));
stringSet = Collections.unmodifiableSet(stringSet);

Map<String, Integer> stringMap = new HashMap<String, Integer>();
stringMap.put("a", 1);
stringMap.put("b", 2);
stringMap.put("c", 3);
stringMap = Collections.unmodifiableMap(stringMap);
```

### 6.3.1 `Collectors.toUnmodifiableMap()`

En java 10 se introduce el concepto de inmutable o no modificable en colecciones java. Vamos a transformar una lista en un Map inmutable.

```
Map<Integer, String> map
=listaEmpleados.stream().collect(Collectors.toUnmodifiableMap(e->e.getId(),e-
>e.getNombre()));
    System.out.println(" Creando en Map inmutable");

try {
    map.put(20, "nombre");
```

```
    } catch (UnsupportedOperationException e) {  
        e.printStackTrace();  
    };
```

Si intentamos insertar en un mapa inmodificable nos arrojará la excepción `UnsupportedOperationException`, operación no permitida sobre el elemento

Las usaremos en ejemplos a lo largo del Tema. Lo tenéis en la documentación de Oracle.

<https://docs.oracle.com/javase/9/core/creating-immutable-lists-sets-and-maps.htm#JSCOR-GUID-DB0865D2-C052-40BC-A3DC-20FCB3088DC9>

### 6.3.1.1 `ImmutableList`, `ImmutableMap`, `ImmutableSet`, Google y Apache. **Actividad guiada librerías Inmutables**

Para vuestro conocimiento existen también las clases `ImmutableList`, `ImmutableMap`, `ImmutableSet`, en las librerías Google de guava `ImmutableCollections` y las librerías de `ImmutableCollections` de Apache. Estas librerías son usadas en los frameworks java de Google y de Apache, como por ejemplo Apache Tomcat, un servidor web Java.

<https://guava.dev/releases/19.0/api/docs/com/google/common/collect/ImmutableCollection.html>

<https://commons.apache.org/proper/commons-collections/apidocs/org/apache/commons/collections4/list/UnmodifiableList.html>

Para usarlas debéis añadir a vuestro proyecto estas librerías. Vamos a añadir un ejemplo usando `com.google.collections` para empezar a manejar el uso de nuevas librerías en nuestros proyectos.


Si estas compilando en un proyecto MAVEN, añadir al `pom.xml`, la siguiente librería en dependencias.

```
<dependencies>  
    <dependency>  
        <groupId>com.google.collections</groupId>  
        <artifactId>google-collections</artifactId>  
        <version>1.0</version>  
    </dependency>  
</dependencies>
```

Si lo estáis haciendo con compilación ant debéis descargar el jar de esta página:



<https://mvnrepository.com/artifact/com.google.collections/google-collections/1.0>



## Google Collections Library » 1.0

Google Collections Library is a suite of new collections and collection-related goodness for Java 5.0

License	Apache 2.0
Categories	Collections
Organization	Google
HomePage	<a href="http://code.google.com/p/google-collections/">http://code.google.com/p/google-collections/</a>
Date	(Dec 30, 2009)
Files	<a href="#">pom (2 KB)</a> <a href="#">jar (624 KB)</a> <a href="#">View All</a>
Repositories	<a href="#">Central</a> <a href="#">AdobePublic</a> <a href="#">Redhat</a> <a href="#">GA</a>
Used By	446 artifacts

Maven

Gradle

SBT

Ivy

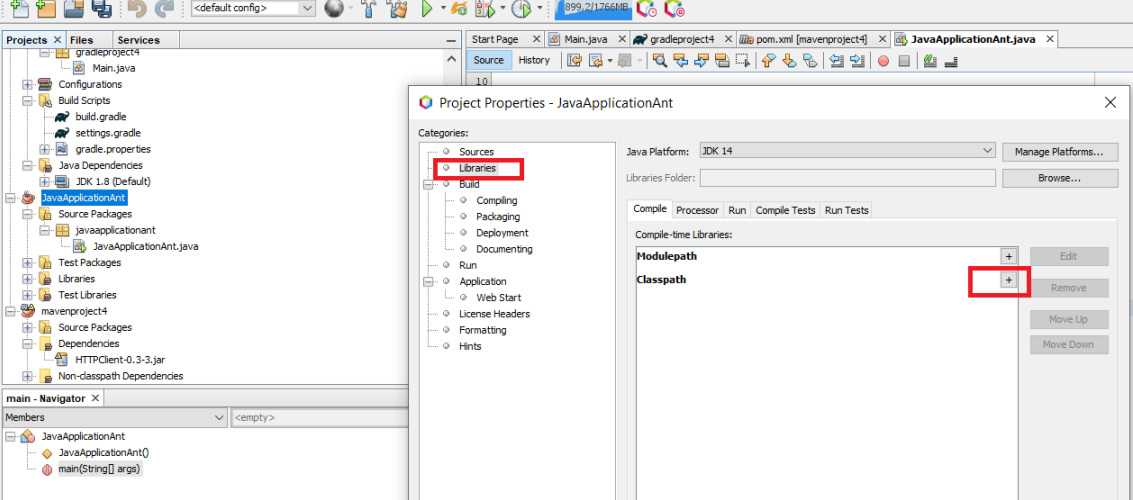
Grape

Leiningen

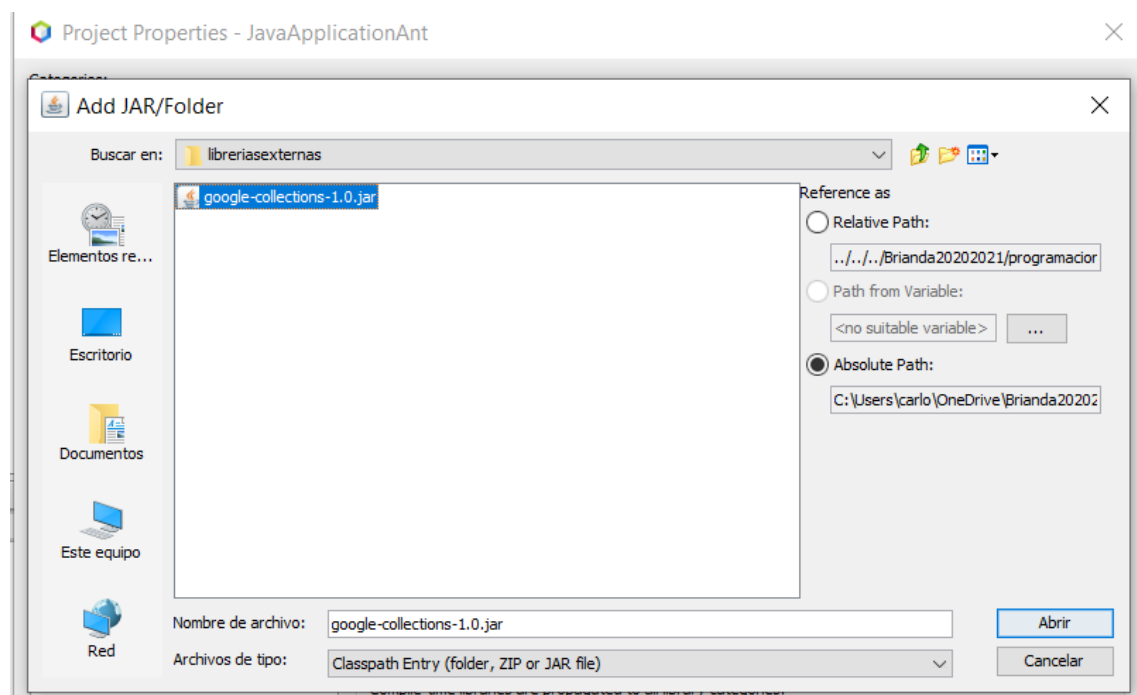
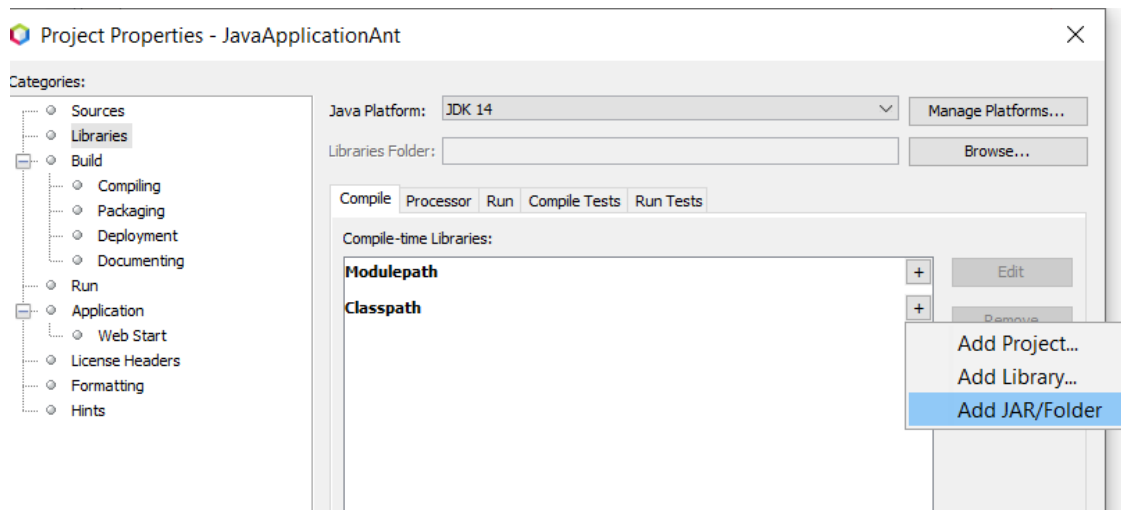
Buildr

```
<!-- https://mvnrepository.com/artifact/com.google.collections/google-collections -->
<dependency>
  <groupId>com.google.collections</groupId>
  <artifactId>google-collections</artifactId>
  <version>1.0</version>
</dependency>
```

E introducirlo en vuestro proyecto, pulsas en Project properties, seleccionar libraries y añadir en compile el jar descargado google-collections-1.0.jar.



Añadimos el jar descargado en el classpath



Con el jar añadido probar el siguiente ejemplo. Importamos las librerías de Google collections en este punto. Hacemos una copia de una lista normal y ya tenemos nuestra lista immutable con librerías Guava de Google.

```
import com.google.common.collect.*;
```

Y creamos una listaImmutable con el método estático copyOf de ImmutableList, a partir de la lista original

```
ImmutableList<String> listaImmutable =
ImmutableList.copyOf(lista);
```

```

import com.google.common.collect.*;

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

/**
 *
 * @author carlo
 */
public class JavaApplicationAnt {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        List<String> lista= Arrays.asList("hola","mundo", "como
estas");

        ImmutableList<String> listaInmutable =
ImmutableList.copyOf(lista);

        // TODO code application logic here

        listaInmutable.stream().forEach(System.out::println);

    }

}

```

## 6.4 Actividad independiente. Listas inmutables

1. A partir de este array {1,3,66,4,5,6,7,8,9} , crear una lista inmutable de la librería que el alumno elija y mostrarla por pantalla con un Stream.

## 6.5 La clase *Collectors* y el método de *Stream collect()*

Es una implementación del interfaz *Collector* que nos proporciona varias operaciones de reducción y agregación para acumular colecciones, resumir elementos a partir de diversos criterios. El método *collect* de la API *Stream* recibe de parámetro un *Collectors* que aplicará un método de reducción o agregación, como veremos a continuación. Podéis ver su signatura a continuación. `<R,A> R collect(Collector<? super T,A,R> collector)`. Recibe como parámetro un *Collector*, con *T* el tipo de objetos del *Stream*, *A* la operación de agregación, y *R* la operación de Reducción. Habitualmente agregamos añadiendo elementos a una colección, y reducimos, devolviendo esa colección, como un objeto.

Tenéis la documentación de Oracle en:

<https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html>  
<https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#collect-java.util.stream.Collector->

Se os proporciona un ejemplo inicial para empezar que transforma un *Stream* en una lista *List*, en un *Set* y en un *Map*. Empezamos a describirlo.

Primero con un *IntStream*.`mapToObj(i->new EmpleadoEnLista(i,nombres[i]))` generamos un *Stream* de objetos *EmpleadoEnLista*, lo recogemos con el método *collect* de *Stream* y aplicamos la operación de agregación *Collectors.toList()* que nos coloca o agrega todos los elementos empleados del *Stream* en una lista.

```
List<EmpleadoEnLista> listaEmpleados = IntStream.range(0, nombres.length).mapToObj(i->new EmpleadoEnLista(i,nombres[i])).collect(Collectors.toList());
```

Creamos una lista nueva de tipo *String* sólo con los nombres a partir de la lista *listaEmpleados*, mapeando sus nombres con *map()*,y con *Collector.toList()*, como parámetro de *collect*, agrupamos en una lista y devolvemos la lista.

```
List<String> list = listaEmpleados.stream().map(EmpleadoEnLista::getNombre).collect(Collectors.toList());
```

Creamos un *TreeSet* a partir de la lista de Empleados usando el método *toCollection* de *Collectors* y pasándole como parámetro un nuevo *TreeSet* *Collectors.toCollection(TreeSet::new)*;

```
listaEmpleados.stream().map(EmpleadoEnLista::getNombre).collect(Collectors.toCollection(TreeSet::new));
```

En este caso podríamos haber usado el método `.collect(Collectors.toSet());` consiguiendo el mismo efecto.

Y por último, creamos un nuevo Map usando `Collectors.toMap(K,V)`. Fijaos que debemos pasar dos valores, la clave y el objeto porque es de tipo Map.

```
Map<Integer, String> map
=listaEmpleados.stream().collect(Collectors.toMap(e->e.getId(),e-
>e.getNombre()));
```

Resumiendo, podemos con `collect()` y `Collectors` crear cualquier tipo de colección a partir de un Stream. Existe una versión Concurrent para estos métodos, `toConcurrentMap` o `toConcurrentList`. Son `ConcurrentMap` o `ConcurrentList` versiones de la listas y maps que son ThreadSafe, sólo un hilo de ejecución puede acceder a ella a la vez. Lo entenderéis en segundo. Se pueden realizar todavía más operaciones que veremos con `Collectors`.

EjemploCollectors.java

```
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.TreeSet;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

class EmpleadoEnLista {

    int id;
    String nombre;
    public EmpleadoEnLista (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 EjemploCollectors {

    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 void main(String[] args) {

        List<EmpleadoEnLista> listaEmpleados = IntStream.range(0,
nombres.length).mapToObj(i->new
EmpleadoEnLista(i,nombres[i])).collect(Collectors.toList());

        System.out.println("\n Acumulando en lista");
        // Acumulamos nombres en la lista
        List<String> list =
listaEmpleados.stream().map(EmpleadoEnLista::getNombre).collect(Collectors.to
List());

        list.forEach((e)->System.out.print(e+","));

        // Acumulamos nombres en el TreeSet
        System.out.println("\n Acumulando en treeSet");
        Set<String> set =
listaEmpleados.stream().map(EmpleadoEnLista::getNombre).collect(Collectors.to
Collection(TreeSet::new));

        set.forEach((e)->System.out.print(e+","));

        // Acumulamos nombres en el HashMap

        Map<Integer, String> map
=listaEmpleados.stream().collect(Collectors.toMap(e->e.getId(),e-
>e.getNombre()));
        System.out.println("\n Acumulando en el MAP");

        map.entrySet().forEach((e)->System.out.print(e+","));

        // Convert elements to strings and concatenate them, separated by
commas

        }

        map.values().stream().forEach((e)->System.out.println(e));
    }
}

```

```
}
```

Desgranaremos más en detalle más métodos de la clase `Collectors` más adelante en el tema, ahora vamos con el funcionamiento de la API Stream.

**Nota:** En conjuntos es obligatorio que la clase del objeto del Stream tenga un método `equals` para poder ordenar e insertar en el conjunto. También es válido un `Comparable`

## 6.6 Como funciona la API Stream

Ya sabemos del tema 6 que un Stream representa una secuencia de elementos y admite diferentes tipos de operaciones para realizar cálculos sobre esos elementos.

## 6.7 Tipos de Operaciones con Streams. Repaso aplicado a colecciones. **Actividad guiada Operaciones no terminales**

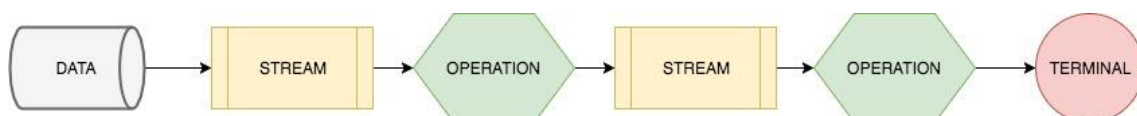
En este apartado vamos a repasar las operaciones con Streams, en este caso para colecciones, con nuevos ejemplos adaptados a colecciones y los realizaremos con diferentes colecciones.

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 secuencia 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.



```

        List<Integer> lista1 = Arrays.asList(1, 2, 3,4,5,6,7,8,9,10);
        List<Integer > lista2 =lista1
        .stream()
        .filter(n->n%2==0)
        .collect(    Collectors.collectingAndThen(Collectors.toList(),
Collections::<Integer> unmodifiableList));    //

```

Podeis verlo completo en el siguiente ejemplo

**StreamOperacionesEjemplo.java**

```

import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import java.util.Map;
import java.util.function.Function;
import java.util.stream.Collectors;

public class StreamOperacionesEjemplo {

    public static void main(String[] args) {

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

        List<Integer> lista = Arrays.asList(1, 2, 3,4,5,6,7,8,9,10);
        lista.stream().forEach( x->System.out.print(x+","));

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

        List<Integer> lista1 = Arrays.asList(1, 2, 3,4,5,6,7,8,9,10);
        List<Integer > lista2 =lista1
        .stream()
        .filter(n->n%2==0)
        .collect(    Collectors.collectingAndThen(Collectors.toList(),
Collections::<Integer> unmodifiableList));    //

        System.out.println("\nFiltramos por números pares y
multiplicamos cada numero par por dos y le sumamos 1");
        Map<String, Integer>map=
        lista2
        .stream()
        .filter(n->n%2==0)
        .map(n -> (Integer) 2 * n + 1)
        .collect(    Collectors.toMap(i->    String.valueOf(i),
Function.identity()));

        Double media =map.values()
        .stream()
        .filter(n->n%2==0)
        .map(n -> 2 * n + 1)
        .collect(Collectors.averagingInt(x->x));

        System.out.println("\nRecogemos la lista anterior y

```

```

    filtramos por números pares y multiplicamos cada numero par por dos y le sumamos
    1. Calculamos la media:" + media);

    }

}

```

### 6.7.1 Practica independiente de operaciones no terminales

Dado la siguiente lista:

```
{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.

### 6.7.2 Actividad guiada generación de clases aleatorias para depuración.

A lo largo de la explicación de las operaciones con Streams usaremos dos clases, **Usuarios** **GeneraUsuarios** y **GeneraAtributosAleatorios**, para poder generar una lista de usuarios que manejaremos para realizar las **operaciones con Streams**. Podéis crear un paquete modelo para estas clases y añadir imports a los ejemplos `import modelo.*;`

**Usuario.java**

```

import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.TreeSet;
import java.util.function.Function;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

public class Usuario {
    private int id;
    private String nombre;
    private String apellidos;
    private Integer Edad;
    private Double horasDeUso;
}

```

```

        private int numConexiones;

        public Usuario(int id, String nombre, String apellidos, Integer edad,
Double horasDeUso, int numConexiones) {

            this.id=id;
            this.nombre = nombre;
            this.apellidos = apellidos;
            Edad = edad;
            this.horasDeUso = horasDeUso;
            this.numConexiones = numConexiones;
        }

        public int getId() {
            // TODO Auto-generated method stub
            return id;
        }
        public String getNombre() {
            return nombre;
        }
        public String getApellidos() {
            return apellidos;
        }
        public Integer getEdad() {
            return Edad;
        }
        public Double getHorasDeUso() {
            return horasDeUso;
        }
        public int getNumConexiones() {
            return numConexiones;
        }
        public void setNombre(String nombre) {
            this.nombre = nombre;
        }
        public void setApellidos(String apellidos) {
            this.apellidos = apellidos;
        }
        public void setEdad(Integer edad) {
            Edad = edad;
        }
        public void setHorasDeUso(Double horasDeUso) {
            this.horasDeUso = horasDeUso;
        }
        public void setNumConexiones(int numConexiones) {
            this.numConexiones = numConexiones;
        }

        @Override
        public String toString() {
            return "Usuario [id=" + id + ", nombre=" + nombre + ",
apellidos=" + apellidos + ", Edad=" + Edad
                                + ", horasDeUso=" + horasDeUso + ",

```

```

numConexiones=" + numConexiones + "];
    }

    @Override
    public int compareTo(Object o) {
        // TODO Auto-generated method stub

        Usuario u2= (Usuario) o;
        return
this.getNumConexiones()>u2.getNumConexiones()?1:(this.getNumConexiones()==u2.
getNumConexiones()?0:-1);
    }

}

```

## GeneraUsuarios.java

```

package operacionesnoterminales;

import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.TreeSet;
import java.util.function.Function;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

public class GeneraUsuarios {

    public static List<Usuario> devueveUsuariosLista(int numUsuarios) {

        return IntStream
            .range(0, numUsuarios)
            .mapToObj(
                i-> new Usuario(i,

                    GeneraCamposAleatorios.getNombreAleatorio(),

                    GeneraCamposAleatorios.getApellidosAleatorio(),
                                GeneraCamposAleatorios.getEdad(),
                                GeneraCamposAleatorios.getHoras(),

                    GeneraCamposAleatorios.numConexiones())

            ).collect(Collectors.toList());

    }

    public static Set<Usuario> devueveUsuariosTree(int numUsuarios) {

```

```

        return IntStream
            .range(0, numUsuarios)
            .mapToObj(
                i-> new Usuario(i,

GeneraCamposAleatorios.getNombreAleatorio(),

GeneraCamposAleatorios.getApellidosAleatorio(),
                                GeneraCamposAleatorios.getEdad(),
                                GeneraCamposAleatorios.getHoras(),

GeneraCamposAleatorios.numConexiones())

            ).collect(Collectors.toCollection(TreeSet::new));

    }

public static Map<Integer,Usuario> devueveUsuariosMap(int numUsuarios) {

    return IntStream
        .range(0, numUsuarios)
        .mapToObj(
            i-> new Usuario(i,

GeneraCamposAleatorios.getNombreAleatorio(),

GeneraCamposAleatorios.getApellidosAleatorio(),
                                GeneraCamposAleatorios.getEdad(),
                                GeneraCamposAleatorios.getHoras(),

GeneraCamposAleatorios.numConexiones())

        ).collect(Collectors.toMap(u-
>u.getId(),Function.identity()));

    }

}

```

## GeneraCamposAleatorios.java

```

import java.util.Random;
import java.util.function.BiFunction;
import modelo.*;
public class GeneraCamposAleatorios {

    private 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"};

        private static final String apellidos[] = {"Garcia", "Gonzalez",
"Rodriguez", "Fernandez", "Lopez", "Martinez", "Sanchez", "Perez", "Gomez",
        "Martin", "Jimenez", "Ruiz", "Hernandez", "Diaz",
"Moreno", "Muñoz", "Alvarez", "Romero", "Alonso", "Gutierrez", "Navarro",
        "Torres", "Dominguez", "Vazquez", "Ramos", "Gil",
"Ramirez", "Serrano", "Blanco", "Molina", "Morales", "Suarez", "Ortega",
        "Delgado", "Castro", "Ortiz", "Rubio", "Marin",
"Sanz", "Nuñez", "Iglesias", "Medina", "Garrido", "Cortes", "Castillo",
"Santos"};

        private static BiFunction<Integer,Integer,Integer> numeroAleatorio =
(min,max) -> new Random().nextInt(max-min) + min ;
        private static BiFunction<Integer,Integer,Double> decimalAleatorio =
(min,max) -> new Random().nextDouble()*(max-min) + min ;

        public static String getNombreAleatorio() {

                return nombres[numeroAleatorio.apply(0, nombres.length-1)];
        }

        public static String getApellidosAleatorio() {

                return apellidos[numeroAleatorio.apply(0, apellidos.length-
1)] + " "
                                + apellidos[numeroAleatorio.apply(0,
apellidos.length-1)];
        }

        public static int getEdad() {

                return numeroAleatorio.apply(18, 100);
        }

        public static double getHoras() {

                return decimalAleatorio.apply(0, 200);
        }

        public static int numConexiones() {

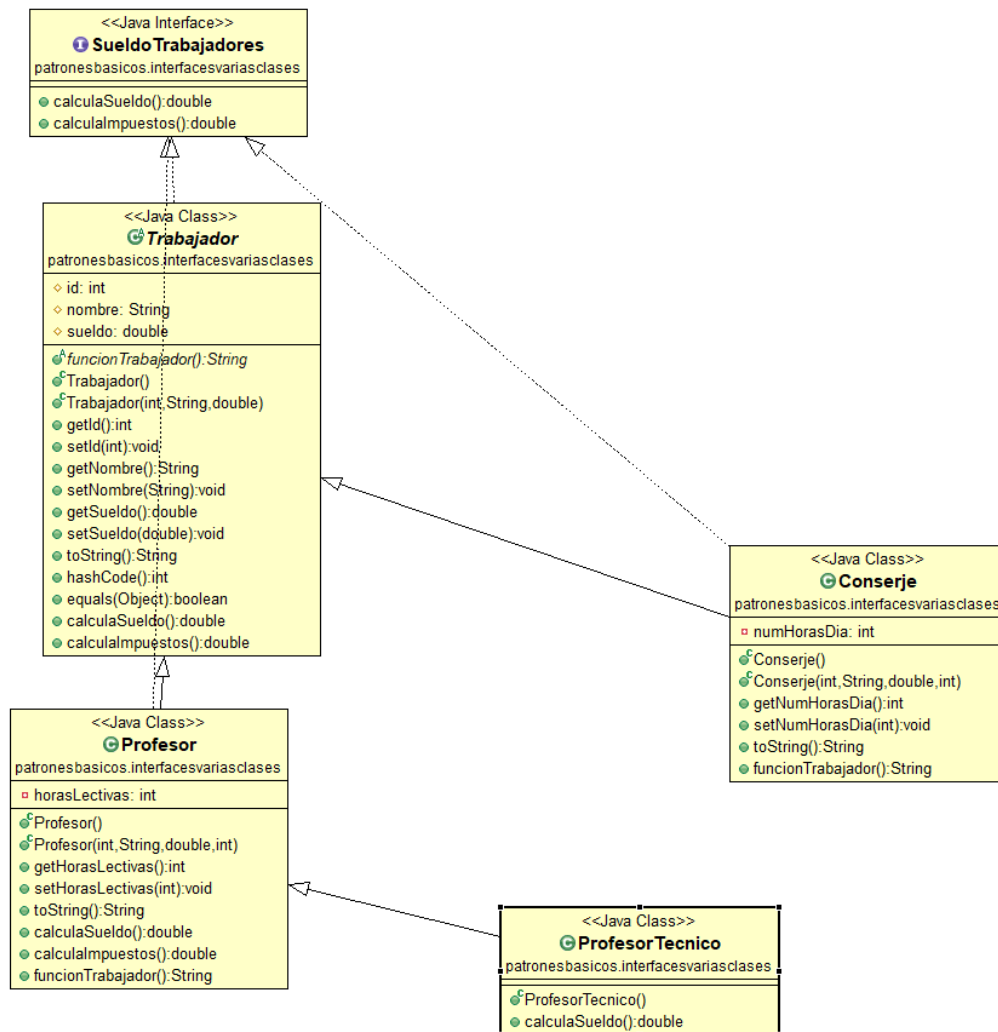
                return numeroAleatorio.apply(1, 500);
        }
}

```

### 6.7.3 Actividad independiente generación de clases aleatorias para depuración.

Para nuestro modelo de clases de Trabajadores de educación se pide

1. Añadir una clase `GeneraCamposAleatorios.java` que genere campos aleatorios para profesor y Conserje.
2. Añadir una clase `GeneraTrabajadores.java` con un método estático que genere una lista de Profesores y una colección de profesores.



## 6.8 Operaciones no terminales

Los ejemplos para este apartado están en el siguiente ejemplo:

`OperacionesNoTerminales.java`

```

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;
import modelo.*;

public class OperacionesNoTerminales {

    public static void main(String[] args) {

```

```

        List<Usuario> listaUsuarios =
GeneraUsuarios.devuelveUsuariosLista(100);

        listaUsuarios
        .stream()
        .map(u->u.getNombre())
        .map(String::toLowerCase)
        .filter(n -> n.startsWith("a") || n.startsWith("b") ||
n.startsWith("s") )
        .forEach(System.out::println);

        listaUsuarios
        .stream()
        .filter(u -> u.getEdad() > 30 )
        .forEach(System.out::println);

        listaUsuarios
        .stream()
        .map(u->u.getNombre() + " " + u.getApellidos())
        .flatMap((nombreAp) -> {
            List<String> NombreYApellidos = new
ArrayList<String>();
            NombreYApellidos.add(nombreAp);

            return (Stream<String>) NombreYApellidos.stream();
        })
        .forEach((nombreAp) -> System.out.print(nombreAp+","));

    }
}

```

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 secuencia 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:

### 6.8.1 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. Filtramos empleados cuya edad sea mayor que 30 en el siguiente ejemplo.

```
listaUsuarios
    .stream()
    .filter(u -> u.getEdad() > 30 )
    .forEach(System.out::println);
```

## 6.8.2 map()

El **método Java Stream map()** convierte (mapea) un elemento en otro 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.

En el **siguiente ejemplo transformamos un Stream de Usuarios en uno de tipo String**, que será el nombre de los usuarios . En el segundo map pasamos el nombre a minúsculas.

```
listaUsuarios
    .stream()
    .map(u->u.getNombre())
    .map(String::toLowerCase)
    .filter(n -> n.startsWith("a") || n.startsWith("b") ||
n.startsWith("s") )
    .forEach(System.out::println);
```

### 6.8.2.1 Mas ejemplos de map

Vamos a realizar unos cuantos ejemplos más de este método, que es de los más útiles cuando toca trabajar con Streams.

En el primer ejemplo transformamos una lista de nombres en una de Empleados.

Nuestra lista de nombres:

```
public static final List<String> lista = (List<String>)
Arrays.asList("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");
```

Con un map y un Collector.Tolist() obtenemos nuestra lista de Empleados, a partir de una lista de Strings. Como veis es más fácil que con arrays. Para crear el empleado usamos como id, el índice del nombre en la lista.

```
new Empleado(lista.indexOf(nombre),nombre)
```

```
List<Empleado> empleadosLista =
    lista.stream().map((nombre) -> (Empleado)
new Empleado(lista.indexOf(nombre),nombre)).collect(Collectors.toList());

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

MapEmpleadosList.java

```
import java.util.Arrays;
import java.util.HashMap;
import java.util.List;
import java.util.stream.Collectors;
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 MapEmpleadosList {

    public static final List<String> lista = (List<String>)
Arrays.asList("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 HashMap<Integer, String> mapaNombres = new
HashMap<Integer, String>();

    public MapEmpleadosList() {

    }

    public static void main(String[] args) {

        List<Empleado> empleadosLista =
            lista.stream().map((nombre) -> (Empleado)
new Empleado(lista.indexOf(nombre), nombre)).collect(Collectors.toList());

        empleadosLista.stream().forEach((objEmpleado) ->
System.out.println((Empleado) objEmpleado));
    }
}

```

### 6.8.3 De lista de String a Map de Empleados

Vamos a hacer lo mismo que en el caso anterior pero con un `HashMap` a partir de un array, para ver más transformaciones. Para ello usamos el `IntStream` del tema 6, con `range`. Nos crea un `Stream` de enteros de 0 al tamaño del array menos 1. Por cada entero hacemos un `forEach`, creamos una entrada en `MapEmpleados` de clave `i`, y de valor, un `Empleado`. Como veis el nombre del empleado lo sacamos del array de nombres.

```
HashMap<Integer, EmpleadoEnLista> mapEmpleados= new HashMap<Integer, EmpleadoEnLista>();
IntStream.range(0,
nombres.length).forEach((i)-> mapEmpleados.put(i, new EmpleadoEnLista(i,
nombres[i])));
System.out.println("Imprimimos hashmap");

mapEmpleados.entrySet().stream().forEachOrdered((e)->System.out.print(e+", "));
```

Después a partir de un `HashMap` creamos una lista, con `stream` y `Collectors.toList`, como hemos visto anteriormente. Sacamos cada empleado del `HashMap` con `getValue`. Es decir, mapeamos de un objeto clave valor `<Integer, EmpleadoEnLista>` a un `EmpleadoEnLista`. `.map((E) -> E.getValue())`

```
List<EmpleadoEnLista> listaEmpleados =
mapEmpleados.entrySet().stream().map((E) ->
E.getValue()).collect(Collectors.toList());
System.out.println("Imprimimos lista");
listaEmpleados.stream().forEachOrdered((e)->System.out.print(e+", "));
```

#### MapEmpleadosMapList.java

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

class EmpleadoEnLista {

    int id;
    String nombre;

    public EmpleadoEnLista (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 MapEmpleadosMapList {

    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 void main(String[] args) {

        HashMap <Integer,EmpleadoEnLista> mapEmpleados= new HashMap
<Integer,EmpleadoEnLista>();
        IntStream.range(0,
nombres.length).forEach((i)-> mapEmpleados.put(i, new EmpleadoEnLista(i,
nombres[i])));
        System.out.println("Imprimimos hashmap");
        mapEmpleados.entrySet().stream().forEachOrdered((e)-
>System.out.print(e+""));
    }
}

```

```

        List<EmpleadoEnLista> listaEmpleados =
mapEmpleados.entrySet().stream().map((E) ->
E.getValue()).collect(Collectors.toList());
        System.out.println("Imprimimos lista");
        listaEmpleados.stream().forEachOrdered((e)-
>System.out.print(e+","));

    }

}

```

### 6.8.3.1 Actividad independiente Mapeo a 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. Crear un HashMap a partir de una lista de nombres y apellidos usando collect() y mapeando a objetos.

### 6.8.4 Mapeando a un tipo básico mapToInt, mapToDouble, etc.

#### Practica guiada mapeo tipos básicos

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. Este proceso nos devolverá Streams de tipo **IntStream** o **DoubleStream**, los Stream básicos que vimos en el tema 6.

```

empleadosLista.stream().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.List;
import java.util.stream.Collectors;
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 List<String> lista =
(List<String>) Arrays.asList("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;

        List<Empleado> empleadosLista =
            lista.stream().map((nombre) ->
(Empleado) new
Empleado(lista.indexOf(nombre), nombre)).collect(Collectors.toList());

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

    }

}

```

### 6.8.5 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 que podéis encontrar en `EjemploOperacionesNoTerminales.java` presentado anteriormente vamos a convertir un Stream de usuarios en un Stream de tipo `String`, donde agrupamos los nombre y apellidos de cada usuario separados por comas.



```

listaUsuarios
    .stream()
    .map(u->u.getNombre() + " " + u.getApellidos())
    .flatMap((nombreAp) -> {
        List<String> NombreYApellidos = new
ArrayList<String>();
        NombreYApellidos.add(nombreAp);
        return (Stream<String>) NombreYApellidos.stream();
    })
    .forEach((nombreAp) -> System.out.print(nombreAp+","));

```

### Ejemplo de ejecución

MARIA LUISA Garrido Gomez, DAVID Iglesias Moreno, JUAN Rubio Dominguez, MARIA LUISA Ramirez Suarez, ELENA Ramos Hernandez, ANTONIA Iglesias Serrano,  
 Observar la operación.

**Primero mapeamos de usuario a nombre y apellidos separados por un espacio.** Generamos un nuevo Stream de tipo cadena con nombre y apellidos juntos.

```

.map(u->u.getNombre() + " " + u.getApellidos())

```

Luego en la function flatMap.

1. Creamos una **Lista con los nombres y apellidos del**

```

.flatMap((nombreAp) -> {
    List<String> NombreYApellidos = new
ArrayList<String>();

```

2. **Añadimos a la lista el nombreAp.**

```

NombreYApellidos.add(nombreAp);

```

3. Y devolvemos un **objeto de Tipo Stream.**

```

    return (Stream<String>) NombreYApellidos.stream();
})

```

4. Al final procesamos el Stream cada nombre y apellidos separados por comas. **Este tipo de operaciones son muy útiles en programas reales.**

```

.forEach((nombreAp) -> System.out.print(nombreAp+","));

```

Los siguientes Ejemplos los encontrareis en **OperacionesTerminales2.java**

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

import modelo.*;

public class OperacionesNoTerminales2 {

    public static int compararPorId(Usuario us1, Usuario us2) {

        return
us1.getId().compareTo(us2.getId()):1:(us1.getId().compareTo(us2.getId())?0:-1);
    }

    public static void main(String[] args) {
        List<Usuario> listaUsuarios =
GeneraUsuarios.devuelveUsuariosLista(100);

        listaUsuarios.add(listaUsuarios.get(99));

        System.out.println("\nPintamos la lista con un duplicado");
        listaUsuarios.stream().forEach(System.out::println);
        System.out.println("\nPintamos la lista ordenada sin duplicados");
        listaUsuarios.stream().
distinct().
forEach(System.out::println);

        System.out.println("\nOrdenamos por defecto, número de conexiones");

        listaUsuarios.stream().
sorted().forEach(System.out::println)

        System.out.println("\nOrdenamos por Nombre");

        Comparator<Usuario> ordenarPorNombre = (u1,u2)-
>u1.getNombre().compareTo(u2.getNombre());

        listaUsuarios.stream().
sorted(ordenarPorNombre).forEach(System.out::println);

        System.out.println("\nOrdenamos por Apellido");

        listaUsuarios.stream().
distinct().
sorted((u1,u2)-
>u1.getApellidos().compareTo(u2.getApellidos()))).forEach(System.out::println)
;

        System.out.println("\nOrdenamos por id");

        listaUsuarios.stream().
distinct().
sorted(OperacionesNoTerminales2::compararPorId).forEach(System.out::p
```

```

println);

        System.out.println("\nPintamos el stream de la lista de usuarios
limitada a 4 ordenada por Apellidos");
        listaUsuarios.stream()
            .limit(4)
            .sorted((u1,u2)->u1.getApellidos().compareTo(u2.getApellidos()))
            .forEach(x->System.out.print(x+", "));

        System.out.println("\nPintamos la lista , y la transformamos a un
Set");
        Set <Usuario> setUsuarios=listaUsuarios.stream()
            .peek(u->System.out.print(u.getApellidos()+" ,"))
            .collect(Collectors.toSet());

    }
}

```

### 6.8.6 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()**:

**Añadimos un elemento repetido y no lo pintamos con distinct, en el ejemplo.**

```

List<Usuario> listaUsuarios = GeneraUsuarios.devuelveUsuariosLista(100);

        listaUsuarios.add(listaUsuarios.get(99));

        System.out.println("\nPintamos la lista con un duplicado");
        listaUsuarios.stream().forEach(System.out::println);
        System.out.println("\nPintamos la lista ordenada sin duplicados");
        listaUsuarios.stream()
            .distinct()
            .forEach(System.out::println);

```

### 6.8.7 sorted()

Nos permite ordenar el stream pasando un **Comparator** como parámetro de tipo objeto del Stream. Lo podemos realizar usando el propio **Comparable** de la clase Usuario. Lo podemos realizar creando el **Comparator** primero, pasando una **expresión lambda** o definiendo un **Comparable** en la clase Usuario.

En `Usuario.java` definimos un `Comparable` comparando a los usuarios por el número de conexiones:

```
@Override
    public int compareTo(Object o) {
        // TODO Auto-generated method stub

        Usuario u2= (Usuario) o;
        return
this.getNumConexiones()>u2.getNumConexiones()?1:(this.getNumConexiones()==u2.
getNumConexiones()?0:-1);
    }
```

En la **ordenación por defecto** usamos la comparación que tiene la propia clase, no pasamos parámetro en el `sorted`. **Ordenamos por conexiones.**

```
System.out.println("\nOrdenamos por defecto, número de conexiones");

    listaUsuarios.stream().
        sorted().forEach(System.out::println);
```

**Creamos un comparador para nombre, para ordenar por nombre.** A veces queremos una **ordenación distinta** a la que nuestra clase tiene definida, en este caso `Conexiones`.

```
Comparator<Usuario> ordenarPorNombre = (u1,u2)->u1.getNombre().compareTo(u2.getNombre());

    listaUsuarios.stream().
        sorted(ordenarPorNombre).forEach(System.out::println);
```

**Pasamos la expresión lambda directamente para apellidos.**

```
System.out.println("\nOrdenamos por Apellido");

    listaUsuarios.stream().
        distinct().
        sorted((u1,u2)->u1.getApellidos().compareTo(u2.getApellidos())).forEach(System.out::println)
    ;
```

Pasamos una función que **sobreescriva el interfaz `Comparator` con el operador `::`**. **Muy importante.** El operador `::` nos permite pasar funciones como parámetro, recordarlo, es muy útil. **Hace a las funciones miembros de primera clase en Java.** En este tema **generaremos un ejemplo más completo posteriormente para terminar de aclarar este punto.** Podría ser no estática si estuviéramos en un método de la propia clase.

```
public static int compararPorId(Usuario us1, Usuario us2) {

    return
us1.getId()>us2.getId()?1:(us1.getId()==us2.getId()?0:-1);
}
```

Usamos la función declarada para ordenar por Id. Debe ser como Comparator, devolver 1, 0 o -1 de tipo entero.

```
listaUsuarios.stream().  
    distinct().  
    sorted(OperacionesNoTerminales2::compararPorId).forEach(System.out::println);
```

### 6.8.8 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

```
listaUsuarios.stream().  
    limit(4).  
    sorted((u1,u2)->u1.getApellidos().compareTo(u2.getApellidos()))).  
    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`

### 6.8.9 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 recogemos una lista en Stream, la pintamos con peek, y la transformamos a un Conjunto.

```
System.out.println("\nPintamos la lista , y la transformamos a un Set");
Set <Usuario> setUsuarios=listaUsuarios.stream()
    .peek(u->System.out.print(u.getApellidos()+" ")).collect(Collectors.toSet());
```

## 6.9 Operaciones Terminales . Practica Guiada

### 1.1.1 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 buscamos si hay algun empleado mayor de 45.

```
boolean empleadosMayor45= setUsuarios.stream().map((u)->
u.getEdad()).anyMatch((edad) -> { return edad>45; });

if (empleadosMayor45) {
    System.out.println("Hay Empleados mayores de 45 años");
}
```

### 1.1.2 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. En el siguiente ejemplo comprobamos si todos los empleados son mayores de 45 años.

```
boolean todosEmpleadosMayor45= setUsuarios.stream().map((u)->
u.getEdad()).allMatch((edad) -> { return edad>45; });

if (!todosEmpleadosMayor45) {
```

```

        System.out.println("No todos los empleados son
mayores de 45 años");
    }

```

### 1.1.3 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. En el siguiente ejemplo comprobamos si **ningun empleado es mayor de 45**.

```

        boolean ningunEmpleadosMayor45= setUsuarios.stream().map((u)->
u.getEdad()).noneMatch((edad) -> { return edad>45; });

        if (!ningunEmpleadosMayor45) {

            System.out.println("Ningun empleados es mayores de 45
años");

        }

```

En el siguiente ejemplo tenéis recogido las dos operaciones anteriores.

**EjemploOperacionesTerminales1.java**

```

package operacionesterminales;

import java.util.Arrays;
import java.util.List;
import java.util.Set;

import modelo.*;

public class EjemploOperacionesTerminales1 {

    public static void main(String[] args) {

        Set<Usuario> setUsuarios =
GeneraUsuarios.devuelveUsuariosSet(100);

        boolean empleadosMayor45= setUsuarios.stream().map((u)->

```

```

u.getEdad()).anyMatch((edad) -> { return edad>45; });

        if (empleadosMayor45) {
            System.out.println("Hay Empleados mayores de 45
años");
        }

        boolean todosEmpleadosMayor45= setUsuarios.stream().map((u)->
u.getEdad()).allMatch((edad) -> { return edad>45; });

        if (!todosEmpleadosMayor45) {
            System.out.println("No todos los empleados son
mayores de 45 años");
        }

        boolean ningunEmpleadosMayor45= setUsuarios.stream().map((u)-
> u.getEdad()).noneMatch((edad) -> { return edad>45; });

        if (!ningunEmpleadosMayor45) {
            System.out.println("Ningun empleados es mayores de 45
años");
        }
    }
}

```

### 1.1.4 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 hemos visto en detalle anteriormente. Seguiremos ampliando en este método junto a `Collectors` más adelante en los apuntes.

### 1.1.5 count()

El método `count` es una operación terminal que nos cuenta el número de elementos del `Stream`.



```

Set<Usuario> setUsuarios = GeneraUsuarios.devuelveUsuariosSet(100);

        long numUsuarios = setUsuarios
        .stream()
        .count();

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

```

**Usando el ejemplo de flatMap** que me convertía a un Stream de palabras, transformamos los apellidos en un Stream de Strings, separándolos por espacios con Split. De esta manera nos quedamos con cada apellido individual, los imprimimos con un peek y luego contamos el número total de apellidos con count.

```

long numApellidos = setUsuarios
    .stream()
    .map( (u) -> u.getApellidos())
    .flatMap((apellidoIndividual) -> {
        String[] apellidos = apellidoIndividual.split(" ");
        return (Stream<String>) Arrays.stream(apellidos);
    })
    .peek((apellidoIndividual)->System.out.print(apellidoIndividual+ ","))
    .count();

    System.out.println("\n El número de apellidos en la lista es: " +
numApellidos);

```

```

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Set;
import java.util.stream.Stream;

import modelo.*;
public class EjemploOperacionesTerminales2 {

        public static void main(String[] args) {

                Set<Usuario> setUsuarios =
GeneraUsuarios.devuelveUsuariosSet(100);

                long numUsuarios = setUsuarios
                .stream()
                .count();

```

```

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

        long numApellidos = setUsuarios
        .stream()
        .map( (u) -> u.getApellidos())
        .flatMap((apellidoIndividual) -> {
            String[] apellidos = apellidoIndividual.split(" ");
            return (Stream<String>) Arrays.stream(apellidos);
        })
        .peek((apellidoIndividual)->System.out.print(apellidoIndividual+
","))
        .count();

        System.out.println("\n El número de apellidos en la lista es: " +
numApellidos);
    }

```

### 1.1.6 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.

Esta marcado en amarillo, en los tres ejemplos, busco en el primero empleados de edad entre 25 y 45

```

Optional nombre = mapUsuarios.values().stream()
                    .filter((usuario) -> usuario.getEdad()>=25 &&
usuario.getEdad()<=45 )
                    .findAny();

```

En el segundo ejemplo buscamos el primer empleado con mas de 100 conexiones.

```

Optional nombre2 = mapUsuarios.values().stream()
                    .filter( (usuario)->
usuario.getNumConexiones()> 100 )
                    .findFirst();

```

```

        nombre2.ifPresent( s->
            System.out.println("Encontramos un usuario con
numero de conexiones mayor que 100 " + s));

```

En el tercer ejemplo empleados cuyo nombre empiecen por a.

```

Optional nombre3= mapUsuarios.values().stream()
    .filter( (usuario)->
usuario.getNombre().toLowerCase().startsWith("a"))
    .findAny();

```

#### **ejemploTerminalesBusqueda.java**

```

import java.util.Map;
import java.util.Optional;
import java.util.Set;
import java.util.stream.Stream;

import modelo.*;
public class EjemploTerminalesBusqueda {

    public static void main(String[] args) {

        Map<Integer,Usuario> mapUsuarios =
GeneraUsuarios.devuelveUsuariosMap(100);

        Optional nombre = mapUsuarios.values().stream()
            .filter((usuario) ->usuario.getEdad()>=25 &&
usuario.getEdad()<=45 )
            .findAny();

        if (nombre.isPresent()) {

            System.out.println("Encontramos usuarios cuya edad
esta entre 25 y 45" + nombre.get());
        } else {

            System.out.println("Encontramos usuarios cuya edad
esta entre 25 y 45" );
        }

        Optional nombre2 = mapUsuarios.values().stream()
            .filter( (usuario)->
usuario.getNumConexiones()> 100 )
            .findFirst();

        nombre2.ifPresent( s->
            System.out.println("Encontramos un usuario con
numero de conexiones mayor que 100 " + s));

```

```

Optional nombre3= mapUsuarios.values().stream()
                    .filter( (usuario)->
usuario.getNombre().toLowerCase().startsWith("a"))
                    .findAny();

    if (nombre3.isPresent()) {
        System.out.println("encontramos un usuario cuyo
nombre empieza por a" + nombre.get());
    } else {
        System.out.println("no encontramos un usuario cuyo
nombre empiezan por a");
    }
}
}

```

### 6.9.1 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()`. Si no se pasa ninguna usará el `Comparable` de los objetos que maneja el Stream. El método `max()` es igual pero devuelve el valor máximo. Recordar que la mayoría de las clases ya tienen implementado el comparador. Al igual que `findAny` y `findFirst`, devuelven un tipo `Optional` con el objeto tipo del Stream. En el tema 6 trabajábamos sobre números, ya ordenados, en este tema trabajamos sobre objetos, debemos incluir en el `max` y en el `min` un `Comparator`, un criterio de ordenación.

En el primer ejemplo obtenemos el usuario con mínimo número de conexiones en el sistema, la comparación por defecto dentro de usuario.

```

Optional<Usuario> usuario = listaUsuarios.stream().min((u1,u2)->
u1.compareTo(u2));

```

En el segundo ejemplo obtenemos el usuario mayor ordenado alfabéticamente, el comienzo de sus apellidos más cercano a la Z, comparando cadenas con el `compareTo` de `String`. No usamos una variable `Optional`, usamos el `ifPresent` de `Optional` para imprimir si lo ha encontrado.

```

listaUsuarios.
                    stream().
                    max((u1,u2)->
u1.getApellidos().compareTo(u2.getApellidos()))).
    ifPresent((us)->System.out.println ("El ultimo usuario en

```

```
orden alfabetico por apellidos "+ us));
```

En el tercer ejemplo usamos la función `usuarioMasHoras` pasada como parámetro al Comparador, con el operador dos puntos.

```
public static int usuarioMasHoras(Usuario u1, Usuario u2) {  
  
    return u1.getHorasDeUso()>u2.getHorasDeUso()?1:(  
u1.getHorasDeUso()>u2.getHorasDeUso()?0:-1);  
}
```

```
listaUsuarios.  
    stream().  
    min( OperacionesTerminalesMaxMin::usuarioMasHoras).  
  
    ifPresent((us)->System.out.println ("El usuario con menos horas en el  
sistema "+ us));
```

#### OperacionesTerminalesMaxMin.java

```
import java.util.Arrays;  
import java.util.List;  
import java.util.Optional;  
import java.util.OptionalInt;  
import modelo.*;  
public class OperacionesTerminalesMaxMin {  
  
    public static int usuarioMasHoras(Usuario u1, Usuario u2) {  
  
        return u1.getHorasDeUso()>u2.getHorasDeUso()?1:(  
u1.getHorasDeUso()>u2.getHorasDeUso()?0:-1);  
    }  
  
    public static void main(String[] args) {  
        List<Usuario> listaUsuarios =  
GeneraUsuarios.devuelveUsuariosLista(100);  
  
        System.out.println("\nPintamos la lista");  
        listaUsuarios.forEach(u->System.out.print(u+", "));  
  
        Optional<Usuario> usuario =  
listaUsuarios.stream().min((u1,u2)-> u1.compareTo(u2));  
  
        System.out.println ("\nUsuario con el numero Minimo de  
conexiones: "+ usuario.get());  
  
        listaUsuarios.  
            stream().  
            max((u1,u2)->  
u1.getApellidos().compareTo(u2.getApellidos()))).  
            ifPresent((us)->System.out.println ("El ultimo usuario en  
orden alfabetico por apellidos "+ us));
```

```

        listaUsuarios.
            stream().
                min( OperacionesTerminalesMaxMin::usuarioMasHoras).

        ifPresent((us)->System.out.println ("El usuario con menos horas en el
sistema "+ us));

    }

}

```

## 6.9.2 Manejando Optionals con el método Map

Otra manera de manejar los resultados de tipo Optional que nos pueden devolver algunas operaciones de la API Stream es combinarlo con la operación map.

En el siguiente ejemplo vamos a buscar un usuario filtrado por edad, después el findAny() nos devolverá un tipo Optional. En caso de que el findAny() devuelva un optional lleno, lo mapemos y extraemos el usuario buscado. El orElse() se ejecutará en caso contrario, cuando el Optional venga vacío. Esto quiere decir que el filtrado no ha tenido éxito. En este caso estamos creando un usuario vacío.

```

Usuario usuario = mapUsuarios.values().stream()
    .filter((user) ->user.getEdad()>=25 &&
user.getEdad()<=45 )
    .findAny().map(usuario -> usuario).orElse(new
Usuario());

```

Otra opción para manejar esta situación de búsquedas o filtrados que no han tenido éxito es lanzar una excepción con el método orElseThrow. En el siguiente ejemplo en lugar de crear un usuario vacío, elegimos lanzar un excepción de tipo UsuarioNotFoundException.

```

    try {
        Usuario usuarioBusqueda = mapUsuarios.values().stream()
            .filter((user) -> user.getEdad() >= 25 &&
                user.getEdad() <= 45 )
            .findAny().map(usuarioMap -> usuarioMap)
            .orElseThrow(() -> new
                UsuarioNotFoundException("Error en la búsqueda de usuario" ));
    } catch (UsuarioNotFoundException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}

```

### 6.9.3 reduce()

Es uno de los métodos más complejos junto a flatMap. Va a reducir el Stream a un solo objeto aplicando algún tipo de operación de agregación. Recibe como parámetro un BynaryOperator<T>, que es una implementación 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 vacía la primera vez.

("", "Nombre Usuario 1") -> "Nombre Usuario 1" es el resultado sobre el primer elemento

("Nombre Usuario 1", "Nombre Usuario 2") -> "Nombre Usuario 1,NombreUsuario 2"

("Nombre Usuario 1,NombreUsuario 2", "Nombre Usuario 3") -> "Nombre Usuario 1,NombreUsuario 2, Nombre Usuario 3"

.... Y así sucesivamente hasta que finalice el Stream y ponga a todos los nombres de usuarios separados por comas

En el ejemplo en la variable de tipo Optional concatenación se almacena el resultado del reduce. Ya sabéis 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.

En el primer ejemplo:

Con el map nos quedamos con los nombres de los usuarios y aplicamos el reduce.

```
Optional concatenacionNombres = setUsuarios.stream()
    .map((us)->us.getNombre())
    .reduce((nombresCombinados, nombres)->
nombresCombinados+", "+nombres );
```

Imprimimos si tenemos resultados:

```
concatenacionNombres.ifPresent(System.out::println);
```

El segundo ejemplo es igual pero paralelo, más rápido, usamos todos nuestros procesadores, y no guardamos en variable Optional, usamos el ifPresent directamente.

```
setUsuarios.parallelStream()
    .map((us)->us.getNombre())
    .reduce((nombresCombinados, nombres)->
nombresCombinados+", "+nombres ).
    ifPresent(System.out::println);
```

#### EjemploReduce.java

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

import modelo.GeneraUsuarios;
import modelo.Usuario;

public class EjemploReduce {

    public static void main(String[] args) {

        Set<Usuario> setUsuarios =
GeneraUsuarios.devuelveUsuariosSet(100);

        Optional concatenacionNombres = setUsuarios.stream()
            .map((us)->us.getNombre())
            .reduce((nombresCombinados, nombres)->
nombresCombinados+", "+nombres );
```



```

concatenacionNombres.ifPresent(System.out::println);

setUsuarios.parallelStream()
    .map((us)->us.getNombre())
    .reduce((nombresCombinados, nombres)->
nombresCombinados+", "+nombres ).
    ifPresent(System.out::println);

}

}

```

La manera funcional sería;

```

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

```

#### 6.9.4 Practica independiente de operaciones terminales

Dada la siguiente lista:

```
{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

#### 6.9.5 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().

### 6.10 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`.

### 6.10.1 `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
```

### 6.10.2 `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
```

### 6.10.3 `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);
```

## 6.10.4 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.

**`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.

**`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

    }

}

```

### 6.10.5 map() y mapToObject(). Practica guiada IntStream

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 ofrecer otros dos ejemplos de **MapToObject**, uno con Empleados, otro con Usuarios.

En este ejemplo usamos un **IntStream** para recoger los nombres el array, y crea un empleado, con el número **num** que genera el **IntStream**. Como entrada tenemos un **Stream** de tipo básico **int**, como salida un **Stream** de tipo **Empleado**, después de realizar el **mapToObj**. Finalmente con un **Collectors.ToList()** agregamos el **Stream** a una lista.

```

List<Empleado> lista=
                                IntStream.rangeClosed(0,nombres.length-
1).mapToObj((num) -> (Empleado) new
Empleado(num,nombres[num])).collect(Collectors.toList());

```

## MaptoObjEmpleados.java

```

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
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) {

        List<Empleado> lista=
            IntStream.rangeClosed(0,nombres.length-
1).mapToObj((num) -> (Empleado) new
Empleado(num,nombres[num])).collect(Collectors.toList());

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

    }
}

```

```
}
```

Otro ejemplo de `mapToObject`, para generar `Usuarios` lo tenéis en `GeneraUsuarios.java`. Lo hemos ofrecido anteriormente. **Volvemos a mapear de `IntStream`, un tipo básico `int`, a `Usuario`**. En este caso estamos recogiendo el `Stream` con `collect` y `Collectors.toList()` en un `Set`, conjunto.

```
return IntStream
    .range(0, numUsuarios)
    .mapToObj(i -> new Usuario(i,
        GeneraCamposAleatorios.getNombreAleatorio(),
        GeneraCamposAleatorios.getApellidosAleatorio(),
        GeneraCamposAleatorios.getEdad(),
        GeneraCamposAleatorios.getHoras(),
        GeneraCamposAleatorios.numConexiones())
    ).collect(Collectors.toCollection(TreeSet::new));
```

**Nota:** Como resumen, `IntStream` nos sirve para realizar una labor similar a un bucle `for`.

### 6.10.6 Practica independiente `IntStream`

Dado el siguiente array de apellidos:

```
public static final String apellidos[] = {"García", "González",
    "Rodríguez", "Fernández", "López", "Martínez", "Sánchez", "Pérez", "Gómez",
    "Martín", "Jiménez", "Ruiz", "Hernández", "Díaz",
    "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.

### 6.10.7 `IntStream` y funciones de agregación. Practica guiada

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. Te 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 a usarlos sobre esta clase aunque el Stream se haya cerrado, 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);
```

### 6.10.8 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.

## 6.11 Ampliación de la clase collectors.

### 6.11.1 Collectors.collectingAndThen()

*CollectingAndThen* es una **operación especial** que tras una **acumulación** nos **permitiría realizar otra**

En el ejemplo después de pasar a lista el Stream le hacemos una copia inmutable con la librería de Google.

```
List<String> listInmutable =
    listaEmpleados
        .stream()
        .collect(Collectors
            .collectingAndThen(
                Collectors.toList(),
                Collections::<String> unmodifiableList));
```

Se **proporciona en versión Google** también. Debéis **eliminarlo del ejemplo** si no habéis **introducido las librerías**.

```
List<String> listInmutable2 =
    listCadenas
        .stream()
        .collect(Collectors
            .collectingAndThen(
                Collectors.toList(),
                ImmutableList::copyOf));
```

#### 6.11.1.1 Collectors.joining()

El **metodo joining** se usa para **unir todos los elementos de una colección**. En este **caso vamos a unirlos en una cadena**

```
String resultadoJoining = listCadenas.stream().collect(Collectors.joining());

System.out.println("\n Resultado joining: " +resultadoJoining);
```

Podemos **usar separadores sufijos y prefijos en el joining**. En el siguiente ejemplo añadimos comas como separador.

```
String resultadoJoining1Separador=
listCadenas.stream().collect(Collectors.joining(", "));
```

```
System.out.println("\n Resultado joining con separador "
+resultadoJoining1Separador);
```

### 6.11.1.2 Collectors.filtering(Predicate, Collector)

Vamos a poder filtrar con un predicate los elementos que acumulamos, usando filtering. Se usa igual que el método filter de Streams, en el primer parámetro. Como segundo parámetro recibe un Collector para realizar la operación de agrupación sobre los elementos tras filtrado, en este caso los elementos filtrados se agrupan en un conjunto

```
Set<EmpleadoEnCollectors> setFiltrado =
listaEmpleados.stream().collect(Collectors.filtering(e-
>e.getNombre().startsWith("A"), Collectors.toSet()));

setFiltrado.forEach((e)->System.out.print(e+", "));
```

#### EjemploCollectors2.java

```
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.TreeSet;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

import com.google.common.collect.ImmutableList;

class EmpleadoEnCollectors {

    int id;
    String nombre;
    public EmpleadoEnCollectors (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 EjemploCollectors2 {

    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 void main(String[] args) {

        List<EmpleadoEnCollectors> listaEmpleados =
IntStream.range(0, nombres.length).mapToObj(i->new
EmpleadoEnCollectors(i,nombres[i])).collect(Collectors.toList());

        System.out.println("\n Acumulando en lista");
        // Acumulamos nombres en la lista
        List<String> listCadenas =
listaEmpleados.stream().map(EmpleadoEnCollectors::getNombre).collect(Collectors.toList());

        listCadenas.forEach((e)->System.out.print(e+","));

        // Acumulamos nombres en el HashMap
        Map<Integer, String> map
=listaEmpleados.stream().collect(Collectors.toUnmodifiableMap(e->e.getId(),e-
>e.getNombre()));
        System.out.println("\n Creando en Map inmutable");
        try {
            map.put(20, "nombre");

        } catch (UnsupportedOperationException e) {

            e.printStackTrace();
        }
    }
}

```

```

        map.entrySet().forEach((e)->System.out.print(e+","));

        System.out.println("\n Creando lista immutable");

        List<String> listImmutable =
            listCadenas
                .stream()
                .collect(Collectors
                    .collectingAndThen(
                        Collectors.toList(),
                        Collections::<String> unmodifiableList));

        listImmutable.forEach((e)->System.out.print(e+","));

        System.out.println("\n Creando lista immutable google");

        List<String> listImmutableGoogle =
            listCadenas
                .stream()
                .collect(Collectors
                    .collectingAndThen(
                        Collectors.toList(),
                        ImmutableList::copyOf));

        listImmutableGoogle.forEach((e)->System.out.print(e+","));

        String resultadoJoining =
listCadenas.stream().collect(Collectors.joining());

        System.out.println("\n Resultado joining: " +resultadoJoining);

        String resultadoJoining1Separador=
listCadenas.stream().collect(Collectors.joining(", "));

        System.out.println("\n Resultado joining con separador "
+resultadoJoining1Separador);

        System.out.println("\n Filtramos empleados que empiecen por a y
los metemos en un conjunto ");

        Set<EmpleadoEnCollectors> setFiltrado =
listaEmpleados.stream().collect(Collectors.filtering(e-
>e.getNombre().startsWith("A"), Collectors.toSet()));

        setFiltrado.forEach((e)->System.out.print(e+","));

    }

```

```
}
```

### 6.11.2 Operaciones de particionamiento, agrupamiento y estadísticas con Collectors.

En este apartado vamos a ver operaciones estadísticas similares a las que veíamos en `IntStream`

#### 6.11.2.1 `counting()`

Cuenta los elementos que provienen del Stream.

```
Long contandoPares = listaEmpleados.stream().filter(e -> e.getId() % 2 == 0).collect(Collectors.counting());
System.out.println("\n Contando id pares: " + contandoPares);
```

#### 6.11.2.2 `minBy(Comparator)` y `maxBy(Comparator)`

Nos obtiene el máximo y el mínimo del Stream, dado un `Comparator`. En el ejemplo en el primer `Comparator`, comparamos por el nombre del empleado. En el segundo comparamos por el id.

```
Optional<EmpleadoEnCollectorsStat> empleadoNombreMayor =
listaEmpleados.stream().collect(Collectors.maxBy((e1, e2)-
->e1.getNombre().compareTo(e2.getNombre())));
System.out.println("\n Retorna empleado Maximo nombre");
empleadoNombreMayor.ifPresent( System.out::print);

Optional<EmpleadoEnCollectorsStat> empleadoIdMenor =
listaEmpleados.stream().collect(Collectors.minBy((e1, e2)->e1.getId() -
e2.getId()));
System.out.println("\n Retorna empleado id Minimo");

empleadoIdMenor.ifPresent( System.out::print);
```

#### 6.11.2.3 `summingInt(ToIntFunction)` y `averagingInt(ToIntFunction)`

Son dos funciones estadísticas que suman o calculan la media de todos los valores del Stream. El parámetro `ToIntFunction` es una función que nos permitirá transformar el elemento recibido a tipo `int` antes de aplicar la operación. Tenemos una versión de estos métodos para `double` y `Long` entre otras.

En el siguiente ejemplo recibimos unas cadenas numéricas y transformamos pasando como parámetro a `averagingDouble` el método `Double::parseDouble` a double los tres elementos cadena antes de calcular la media.

```
Double media = Stream.of("12", "23", "1").collect(Collectors.averagingDouble(Double::parseDouble));

System.out.println("\n Aplicando media a un Stream:" + media);
```

Recordar que cuando usamos como parámetro un interfaz funcional, podemos pasar una expresión lambda, o una función java como parámetro. En este tercer caso construyo mi propia función para pasar como parámetro, en mi clase `EjemploCollectors3.java`

```
public static int transformaAEntero(String cadenaNumerica) {
    return Integer.parseInt(cadenaNumerica);
}

int mediaEntero = Stream.of("12", "23", "1").collect(Collectors.summingInt(EjemploCollectors3::transformaAEntero));

System.out.println("\n Aplicando suma a un Stream:" + mediaEntero);
```

En el tercer caso sumamos los id's de todos los empleados. En este caso uso una expresión lambda para implementar el parámetro `FunctionToInt`.

```
int SumaIds=
    listaEmpleados.stream().map(e->e.getId()).collect(Collectors.summingInt((x)->(int) x));

System.out.println("\n Aplicando la suma de todos los id's de empleados del stream a un Stream:" + media);
```

### EjemploCollectors3.java

```
import java.util.Collections;
import java.util.HashMap;
import java.util.IntSummaryStatistics;
import java.util.List;
import java.util.Map;
import java.util.Optional;
import java.util.Set;
import java.util.TreeSet;
import java.util.function.Function;
import java.util.function.ToIntFunction;
import java.util.stream.Collectors;
import java.util.stream.IntStream;
import java.util.stream.Stream;

import com.google.common.collect.ImmutableList;
```

```

class EmpleadoEnCollectorsStat {

    int id;
    String nombre;
    public EmpleadoEnCollectorsStat (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 static Long TransformaALongIdEmpleado(int id) {

        return Long.valueOf(id);

    }

    public String toString () {

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

}

public class EjemploCollectors3 {

    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 transformaAEntero(String cadenaNumerica) {

        return Integer.parseInt(cadenaNumerica);

    }

}

```



```

        public static void main(String[] args) {

            List<EmpleadoEnCollectorsStat> listaEmpleados =
IntStream.range(0, nombres.length).mapToObj(i->new
EmpleadoEnCollectorsStat(i,nombres[i])).collect(Collectors.toList());

            System.out.println("\n Acumulando en lista");
            // Acumulamos nombres en la lista
            List<String> listCadenas =
listaEmpleados.stream().map(EmpleadoEnCollectorsStat::getNombre).collect(Coll
ectors.toList());

            listCadenas.forEach((e)->System.out.print(e+", "));

            System.out.println("\n Contando id pares");
            Long contandoPares = listaEmpleados.stream().filter(e ->
e.getId() % 2 == 0).collect(Collectors.counting());
            System.out.println("\n Contando id pares: " + contandoPares);

            Optional<EmpleadoEnCollectorsStat> empleadoNombreMayor =
listaEmpleados.stream().collect(Collectors.maxBy((e1, e2)-
>e1.getNombre().compareTo(e2.getNombre())));
            System.out.println("\n Retorna empleado Maximo nombre");
            empleadoNombreMayor.ifPresent( System.out::print);

            Optional<EmpleadoEnCollectorsStat> empleadoIdMenor =
listaEmpleados.stream().collect(Collectors.minBy((e1, e2)->e1.getId() -
e2.getId()));
            System.out.println("\n Retorna empleado id Minimo");
            empleadoIdMenor.ifPresent( System.out::print);

            Double media = Stream.of("12", "23",
"1").collect(Collectors.averagingDouble(Double::parseDouble));

            System.out.println("\n Aplicando media a un Stream:" + media);

            int mediaEntero = Stream.of("12", "23",
"1").collect(Collectors.summingInt(EjemploCollectors3::transformaAEntero));

            System.out.println("\n Aplicando suma a un Stream:" +
mediaEntero);

            int SumaIds=
                listaEmpleados.stream().map(e-
>e.getId()).collect(Collectors.summingInt((x)->(int) x));
            System.out.println("\n Aplicando la suma de todos los id's de
empleados del stream a un Stream:" + media);

```

```

    }
}

```

### 1.1.6.1 Java Collectors groupingBy(Function)

**Agrupar elementos** dada una función o Interfaz Function pasado como parámetro. Crea un HashMap cuyo key es el resultado de la función y cuyo valor es una lista de los elementos que agrupamos. En el siguiente ejemplo vamos a agrupar los nombres por letra de comienzo o inicial.

**Agrupamos en el primer ejemplo con clave, inicial del nombre del empleado,** y lista todos los empleados que empiecen por esa inicial. La clave del Map se obtiene con una expresión lambda, `((e)->e.getNombre().substring(0,1))` que nos devuelve la primera letra del nombre. Ejecutarlo para ver como funciona.

```

        Map<String,List<EmpleadoEnCollectorsAgrup>> mapAgrupadoInicial =
            listaEmpleados.stream().collect(Collectors.groupingBy((e)-
            >e.getNombre().substring(0,1) ));

        System.out.println("\n    Map    agrupando    por    inicial"    +
mapAgrupadoInicial);

```

### 1.1.6.2 Java Collectors partitioningBy(Predicate)

Igual que el anterior, pero solo sirve para crear dos grupos al particionar, true y false. Para separar unos de otros usamos un Predicate que pasamos como parámetro al PartitioningBy. Aunque use una expresión lambda también podría definir una función que recibiera un elemento del Stream y devolviera true or false para pasar como parámetro.

Agrupamos por id's pares, usando lambda

```

Map<Boolean, List<EmpleadoEnCollectorsAgrup>> mapAgrupadoIdpar =

```

```

listaEmpleados.stream().collect(Collectors.partitioningBy((e)-
>e.getId()%2==0));

                                System.out.println("\n Map agrupando por id par o
impar, false impar, true par" + mapAgrupadoIdpar);

```

**El mismo ejemplo pero con una función definida para el Predicate. Ejecutar el ejemplo**

```

public static boolean compruebaPar(EmpleadoEnCollectorsAgrup empl) {

    return empl.getId()%2==0;

}

                                Map<Boolean,                               List<EmpleadoEnCollectorsAgrup>>
mapAgrupadoIdpar2 =

listaEmpleados.stream().collect(Collectors.partitioningBy(EjemploCollectors4:
:compruebaPar));

                                System.out.println("\n                               Map
agrupando por id par o impar, false impar, true par" + mapAgrupadoIdpar);

```

## Nota importante:

Cuando tenemos un parámetro de tipo interfaz funcional, podemos pasar:

1. Una lambda
2. Una función estática.

Opcionalmente, sólo cuando el tipo de objeto del Stream es el mismo que el de la función que vamos a usar podemos usar una función no estática con el parámetro ::

En este ejemplo que aparece en EjemploCollectors3.java fijos que `EmpleadoEnCollectorsStat::getNombre`, `getNombre` no es estático pero lo puedo usar así porque por el tipo del Stream, que es de objetos `EmpleadoEnCollectorsStat`.

`EmpleadoEnCollectorsStat::getNombre` es equivalente a `e->e.getNombre()`

```

List<String>                                listCadenas                                =
listaEmpleados.stream().map(EmpleadoEnCollectorsStat::getNombre).collect(Coll
ectors.toList());

```

Los ejemplos anteriores los tenéis en **EjemploCollectors4.java**

```
import java.util.Collections;
import java.util.HashMap;
import java.util.IntSummaryStatistics;
import java.util.List;
import java.util.Map;
import java.util.Optional;
import java.util.Set;
import java.util.TreeSet;
import java.util.function.Function;
import java.util.function.ToIntFunction;
import java.util.stream.Collectors;
import java.util.stream.IntStream;
import java.util.stream.Stream;

import com.google.common.collect.ImmutableList;

class EmpleadoEnCollectorsAgrup {

    int id;
    String nombre;
    public EmpleadoEnCollectorsAgrup (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 static Long TransformaALongIdEmpleado(int id) {

        return Long.valueOf(id);
    }

    public String toString () {

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

```

    }

}

public class EjemploCollectors4 {

    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 boolean compruebaPar(EmpleadoEnCollectorsAgrup empl)
    {

        return empl.getId()%2==0;

    }

    public static void main(String[] args) {

        List<EmpleadoEnCollectorsAgrup> listaEmpleados =
IntStream.range(0, nombres.length).mapToObj(i->new
EmpleadoEnCollectorsAgrup(i,nombres[i])).collect(Collectors.toList());

        System.out.println("\n Acumulando en lista");
        // Acumulamos nombres en la lista
        List<String> listCadenas =
listaEmpleados.stream().map(EmpleadoEnCollectorsAgrup::getNombre).collect(Col
lectors.toList());

        listCadenas.forEach((e)->System.out.print(e+", "));

        Map<String,List<EmpleadoEnCollectorsAgrup>>
mapAgrupadoInicial =

        listaEmpleados.stream().collect(Collectors.groupingBy((e)-
>e.getNombre().substring(0,1) ));

        System.out.println("\n Map agrupando por inicial" +
mapAgrupadoInicial);
    }
}

```

```

        Map<Boolean, List<EmpleadoEnCollectorsAgrup>> mapAgrupadoIdpar =
listaEmpleados.stream().collect(Collectors.partitioningBy((e)-
>e.getId()%2==0));

        System.out.println("\n Map agrupando por id par o
impar, false impar, true par" + mapAgrupadoIdpar);

        Map<Boolean, List<EmpleadoEnCollectorsAgrup>>
mapAgrupadoIdpar2 =
listaEmpleados.stream().collect(Collectors.partitioningBy(EjemploCollectors4:
:compruebaPar));

        System.out.println("\n Map
agrupando por id par o impar, false impar, true par" + mapAgrupadoIdpar);
    }
}

```

## 6.12 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) {

        List<String> listaNombres =Arrays.asList("Al", "Ankit",
"Brent", "Tomas", "Alejandro", "Aitor" ,"Sarika", "amanda", "Hans",
"Shivika", "Sarah", "Julius");

        List<String> listaNombres2 = Arrays.asList("lodi", "samuel"
,"iker" , "Jeronimo", "Gracielo", "Rene","Reus");

        Stream<String> stream1 =listaNombres.stream();

```

```

        Stream<String> stream2 = listaNombres2.stream();

        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+","));

    }
}

```

## 6.13 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 de Stream se puede realizar en cada procesador, lo único que hay que realizar es llamar al método `parallelStream()`, que transforma nuestra ejecución en paralela. En el ejemplo anterior hemos llamado al método `parallel()` para hacer nuestra ejecución paralela. Es lo que único que hay que hacer. Hay un método `isParallel()`, para comprobar si el stream es paralelo. Podéis probar a incluirlo en ejemplos anteriores. La ventaja de la ejecución paralela es que podemos usar todos los núcleos de nuestro procesador para ejecutar las operaciones con Streams. Si tengo cuatro núcleos en mi procesador, será cuatro veces más rápida

```

listaNombres.parallelStream()
    .reduce((strCombinado, str)-> strCombinado+str)
    .ifPresent(System.out::println);

```

## 7 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 cadenas. 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 )
```

Recordar que al usar el operador ... podemos pasar de cero a N parámetros, podríamos llamar al método con `funcionConPuntosSuspensivosEnteros("Luis")`;, sin pasar parámetros a números, **cuidado**.

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));

    }

}

```

## 8 Composición avanzada. Uso de Interfaces como parámetros

### 8.1 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));

    }

}

```

## 8.2 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 resultado 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 añadir 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));
    }
}

```

### 8.3 Recorriendo los parámetros interfaz funcional del operador ... con un Stream. **Practica de ampliacion**

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 genérico. El primero parámetro es un elemento del Tipo T,  $\tau$  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 genérico 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.  $x \rightarrow x*x$ ,  $x \rightarrow x-7$ ,  $x \rightarrow x/3$ ,  $x \rightarrow x*5$ , 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);
```

Podría 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);

    }

}
```

```
}
```

### 8.3.1 Refactorizando el ejemplo anterior. Practica de ampliación

Para finalizar vamos a ofrecer una manera más pura de resolver el ejercicio anterior convirtiendo a la función `ComposicionAvanzada` en una función pura que nos devuelve una función.

Este reduce es más difícil porque componemos cada función de dentro de interfaces, primero con `identity`, la función identidad,  $x \rightarrow x$ , y luego con `andThen`. Lo que estamos haciendo es a cada `Function` dentro de interfaces aplicarle el `andThen` con el anterior, hasta que termina el Stream.

En este caso la función devuelve el interfaz compuesto `Function<T,T>`

```
        return Stream.of(interfaces).reduce(Function.identity(),
Function::andThen);

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

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

public class EjemploDeComposicionAvanzadaPuntosSuspensivosRefactorizado<T> {

    public EjemploDeComposicionAvanzadaPuntosSuspensivosRefactorizado() {

    }

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

        return Stream.of(interfaces).reduce(Function.identity(),
Function::andThen);

    }

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



```

        int i =9;

        EjemploDeComposicionAvanzadaPuntosSuspensivosRefactorizado<Integer>
ejemploAvanzada = new
EjemploDeComposicionAvanzadaPuntosSuspensivosRefactorizado();

        Function<Integer,Integer> funcion =
ejemploAvanzada.ComposicionAvanzada( x -> x*x, x->x-7, x->x/3, x->x*5, x-
>2*x, y->5*y/2);

        System.out.println("El resultado de aplicar los interfaces
funcionales a la variable " + i + " es " + funcion.apply(i));

    }

}

```

## 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 Jiménez Pérez, Francisco Manuel Pérez Montes,  
Paraninfo, 1ª edición, 2021