

# U.D.13: STREAM

Basado en el libro de  
Paraninfo

## **INTRODUCCIÓN**

Las colecciones aportan versatilidad y potencia al procesamiento y la manipulación de datos complejos. Sin embargo, para recorrerlas disponemos de los iteradores, cuyo manejo puede resultar incómodo.

A partir de Java 8, se ha introducido una serie de herramientas que permiten efectuar operaciones globales con los elementos de una colección, sin necesidad de recorrerlas nodo a nodo, aprovechando el procesamiento paralelo (ejecución simultánea de dos partes del código), de una forma transparente al programador.

## **INTRODUCCIÓN**

También pueden encadenarse, una a continuación de otra, formando tuberías, para dar un resultado final, sin necesidad de acceder a resultados intermedio. Aquí vamos a introducir los conceptos más importantes, como los Stream o las tuberías, con sus aplicaciones más frecuentes.

## INTERFACES FUNCIONALES Y EXPRESIONES LAMBDA

En la Unidad 9, donde estudiamos las interfaces, distinguíamos entre métodos por defecto, estáticos y abstractos. De todos ellos, en la definición de la clase solamente hay que implementar los últimos.

Se llaman ***interfaces funcionales*** a aquellas que tienen un solo método abstracto. Son especialmente importantes porque tienen una sintaxis alternativa que permite una implementación más sencilla.

## INTERFACES FUNCIONALES Y EXPRESIONES LAMBDA

Esto ha hecho que, de un tiempo a esta parte, proliferen las interfaces funcionales para tareas específicas que surgen con frecuencia en el trabajo del programador. Quizá la más conocida es la interfaz **Comparator**, que ya hemos usado repetidas veces y que nos va a servir como ejemplo.

A la hora de implementar una clase comparadora, podemos seguir varios caminos. Lo vamos a ilustrar manejando la lista de clientes de la unidad anterior. Supongamos que, en determinados momentos, queremos hacer una ordenación o una búsqueda por nombres, para lo cual necesitamos un comparador basado en el atributo **nombre**.

# INTERFACES FUNCIONALES Y EXPRESIONES LAMBDA

## Primera forma

Creamos explícitamente una clase **ComparaNombres**, que implemente la interfaz **Comparator**, para comparar objetos **Cliente** basándose en el atributo nombre.

```
class ComparaNombres implements Comparator<Cliente> {  
    public int compare(Cliente c1, Cliente c2) {  
        return c1.nombre.compareTo(c2.nombre);  
    }  
}
```



# INTERFACES FUNCIONALES Y EXPRESIONES LAMBDA

## Primera forma

A continuación, creamos un objeto **ComparaNombres** y lo pasamos a la función donde se va a usar.

```
Comparator<Cliente> comp = new ComparaNombres( );
```

```
Collections.sort(lista, comp); /* la lista queda ordenada por nombres */
```

Incluso podríamos prescindir de la variable **comp**, escribiendo una sola sentencia,

```
Collections.sort(lista, new ComparaNombres( ));
```

# **INTERFACES FUNCIONALES Y EXPRESIONES LAMBDA**

## **Segunda forma**

Si vamos a usar el comparador una sola vez, no merece la pena implementar la clase comparadora explícitamente. Basta crear un objeto con una clase anónima.

```
Comparator<Cliente> comp = new Comparator<>( ) {  
    public int compare(Cliente c1, Cliente c2) {  
        return c1.nombre.compareTo(c2.nombre);  
    }  
}  
Collections.sort(lista, comp);
```



# INTERFACES FUNCIONALES Y EXPRESIONES LAMBDA

## Segunda forma

O bien,

```
Collections.sort(lista, new Comparator<>( ) {  
    public int compare(Cliente c1, Cliente c2) {  
        return c1.nombre.compareTo(c2.nombre);  
    }  
});
```

donde el constructor de **Comparator** usa el operador diamante, ya que Java infiere el tipo **Cliente** de la lista que se pasa como primer parámetro.

# INTERFACES FUNCIONALES Y EXPRESIONES LAMBDA

## Tercera forma (expresiones lambda)

La sentencia anterior es la forma más corta de escribir el código para hacer la ordenación de la lista de clientes, pero en ella hay información redundante. Podríamos preguntarnos por qué es necesario especificar el nombre del método **compare( )** cuando sabemos que la interfaz **Comparator** solo tiene ese método abstracto.

Esa es la idea que subyace en la sintaxis de las expresiones **lambda**.

# INTERFACES FUNCIONALES Y EXPRESIONES LAMBDA

## Tercera forma (expresiones lambda)

Para implementar una interfaz funcional con una expresión lambda, basta escribir la lista de parámetros y el cuerpo de la función abstracta separados por una flecha (->). En nuestro ejemplo, implementar el comparador de nombres de clientes consiste en implementar el método **compare( )** que, en forma de expresión lambda, quedaría así:

```
Comparator<Cliente> comp =  
(Cliente a, Cliente b) -> {return a.nombre.compareTo(b.nombre);} ;
```

## INTERFACES FUNCIONALES Y EXPRESIONES LAMBDA

### Tercera forma (expresiones lambda)

Todo lo que está a la derecha del operador de asignación es la expresión lambda del método **compare( )** de la interfaz **Comparator**, implementado para comparar nombres. El nombre del método no aparece, ya que Java lo infiere del lado izquierdo, donde aparece el de la interfaz **Comparator**, cuyo único método abstracto es **compare( )**.

Por tanto, Java sabe que en el lado derecho estamos implementando **compare( )**.

# INTERFACES FUNCIONALES Y EXPRESIONES LAMBDA

## Tercera forma (expresiones lambda)

En realidad, también infiere el tipo de los parámetros de entrada (**Cliente** en nuestro caso), que igualmente se puede omitir del lado derecho.

```
Comparator<Cliente> comp =  
(a, b) -> {return a.nombre.compareTo(b.nombre); } ;
```

## INTERFACES FUNCIONALES Y EXPRESIONES LAMBDA

### Tercera forma (expresiones lambda)

En general, entre las llaves podemos escribir tantas sentencias como sean necesarias. En casos como el anterior, en el que el cuerpo de la función es una sola sentencia, tampoco es necesaria la orden **return**. También podemos prescindir de la variable **comp** y colocar la expresión lambda directamente en la lista de parámetros de **sort( )**.

```
Collections.sort(lista,  
(a, b) -> {return a.nombre.compareTo(b.nombre); } ) ;
```

## INTERFACES FUNCIONALES Y EXPRESIONES LAMBDA

### Tercera forma (expresiones lambda)

Java sabe que el segundo parámetro se **sort( )** es un objeto **Comparator** e interpreta que el código que le pasamos corresponde al método **compare( )** implementado para objetos **Cliente**, que es el tipo genérico asociado a la lista que se pasa como primer parámetro.

La sintaxis general de una expresión lambda es:

(tipo1 param1, tipo2 param, ...) -> {Cuerpo de la expresión lambda };



# **INTERFACES FUNCIONALES Y EXPRESIONES LAMBDA**

## **Tercera forma (expresiones lambda)**

Es decir,

- Una lista de parámetros formales, entre paréntesis, separados por comas. Los tipos de los parámetros se pueden omitir si Java los puede inferir del contexto. Cuando hay un solo parámetro de entrada, también se pueden omitir los paréntesis.
- Una flecha -> (guion alto “-” seguido de “>”).

# INTERFACES FUNCIONALES Y EXPRESIONES LAMBDA

## Tercera forma (expresiones lambda)

- El cuerpo de la función entre llaves, que puede consistir en una sentencia o un bloque de sentencias. Si es una única sentencia y no devuelve ningún valor, las llaves se pueden omitir. Si es una única sentencia y devuelve un valor, la orden **return** se puede omitir, ya que Java devuelve automáticamente el resultado de la sentencia.

## INTERFACES FUNCIONALES Y EXPRESIONES LAMBDA

**Actividad Resuelta 13.1.** Definir una interfaz funcional cuya función abstracta permita generar un saludo dirigido al objeto que se le pasa como parámetro. Implementar un saludo para nombres (clase **String**) y otra para clientes (clase **Cliente**). Aplicarlas a varios casos particulares.

**Actividad Resuelta 13.2.** Utilizando la interfaz **Saludo** de la actividad anterior, implementa un método estático que aplique un saludo a un grupo de personas que se le pasa como parámetro en una tabla. Devolverá los saludos en una lista de cadenas. Aplicarlos a una tabla de clientes.

## **INTERFACES FUNCIONALES Y EXPRESIONES LAMBDA**

**Actividad Resuelta 13.3.** Implementar un método estático al que se pasa como parámetro una tabla de tipo genérico y un comparador para dicho tipo. El método devuelve el valor máximo de los elementos de la tabla según el criterio de orden del comparador. Aplicarlo a una tabla de clientes para buscar el de más edad.

## ALGUNAS INTERFACES FUNCIONALES DE LA API

En vista de la simplicidad y la versatilidad de las interfaces funcionales, se ha definido un cierto número de ellas que, como **Comparator**, corresponden a operaciones fundamentales, frecuentes en las tareas del programador. A continuación, vamos a ver las más importantes, que además serán necesarias con los objetos de tipo **Stream** que estudiaremos más adelante.

- **Predicate<T>** : se emplea para comprobar una condición en un valor del tipo genérico **T**. Su método abstracto es:  
**boolean test (T valor)** : devuelve **true** si la condición se verifica para **valor** y **false** en caso contrario.

## ALGUNAS INTERFACES FUNCIONALES DE LA API

Por ejemplo, para comprobar si un **Integer** es positivo, podemos definir el predicado.

```
Predicate<Integer> esPositivo = x -> x > 0;
```

Entonces,

```
esPositivo.test(5)
```

devolverá **true**.

## ALGUNAS INTERFACES FUNCIONALES DE LA API

El método **test( )** es el único abstracto de la interfaz **Predicate**, pero junto a él hay otros tres métodos por defecto:

1. **Predicate<T> negate( )**: devuelve un nuevo predicado que es la negación del predicado invocante. En nuestro caso,

`esPositivo.negate( )`

nos devuelve un predicado que comprueba si un Integer no es positivo (es menor o igual que 0).



## ALGUNAS INTERFACES FUNCIONALES DE LA API

Predicate<Integer> esNoPositivo = esPositivo.negate( );

La expresión

esNoPositivo.test(5)

devolverá **false**. En una sentencia única:

esPositivo.negate( ).test(5)

que dará el mismo resultado, **false**.

## ALGUNAS INTERFACES FUNCIONALES DE LA API

2. **Predicate<T> and(Predicate<? Super T> otro)**: devuelve un predicado que es la conjunción del predicado invocante y del que se le pasa como parámetro, de modo que **test( )** devolverá **true** cuando los dos predicados sean ciertos para el valor que se le pase como parámetro.

El tipo genérico de **otro** debe ser igual o una superclase de **T** para garantizar que no va a contener ni evaluar más atributos que los de la clase **T**.

Veámoslo con un ejemplo.

## **ALGUNAS INTERFACES FUNCIONALES DE LA API**

Para ello vamos a definir un segundo predicado,

```
Predicate<Integer> esPar = n -> n % 2 == 0;
```

que comprueba si un entero es par.

Si queremos saber si el entero 6 es par y positivo, escribimos

```
Predicate<Integer> esPositivoYPar = esPar.and(esPositivo);
```

## ALGUNAS INTERFACES FUNCIONALES DE LA API

Entonces, la expresión

```
esPositivoYPar.test(6);
```

devolverá **true**, ya que 6 es par y positivo a la vez.

También podemos poner

```
esPar.and(esPositivo).test(6)
```

## ALGUNAS INTERFACES FUNCIONALES DE LA API

En cambio,

```
esPar.and(esPositivo).test(-6);
```

y

```
esPar.and(esPositivo).test(7);
```

devuelven **false**, ya que – 6 es par, pero no positivo y 7 es positivo, pero impar.

## ALGUNAS INTERFACES FUNCIONALES DE LA API

3. **Predicate<T> or(Predicate<? Super T> otro):** devuelve un predicado cuyo método **test( )** devolverá **true** cuando al menos uno de los dos predicados (invocante y otro), sea **true** para el valor que se le pase como parámetro.

```
Predicate<Integer> esPositivoOPar = esPositivo.or(esPar);
```

```
esPositivoOPar.test(6) //true, par y positivo
```

```
esPositivoOPar.test(5) //true, es positivo
```

```
esPositivoOPar.test(-2) //true, es par
```

```
esPositivoOPar.test(6) //false, no es par ni positivo
```

## ALGUNAS INTERFACES FUNCIONALES DE LA API

- **Function<T, V>** : coincide con la funcionalidad de las funciones matemáticas. Su único método abstracto es:

**V apply(T x)** : acepta un parámetro de tipo **T** con el que hace una serie de operaciones que dan como resultado un valor de tipo **V**, que es devuelto por la función. Por ejemplo, si queremos definir una función que calcula el cuadrado de un valor real (de tipo Double),

```
Function<Double, Double> cuadrado = x -> x * x;
```

```
System.out.println(cuadrado.apply(2.0)); /* mostrará 4.0 por consola*/
```



## ALGUNAS INTERFACES FUNCIONALES DE LA API

Además, **Function** añade tres funciones por defecto, que sirven para componer funciones, y que no las vamos a ver en este curso.

- **Consumer<T>** : sirve para realizar una acción a partir de un argumento de entrada. Su método abstracto es:  
**void accept(T t)** : recibe un valor de tipo **T**, con el que hace operaciones sin devolver nada.

## ALGUNAS INTERFACES FUNCIONALES DE LA API

Por ejemplo, si queremos mostrar por pantalla un saludo a distintos clientes,

```
Consumer<Cliente> saludoClie = c -> System.out.println("Hola, " + c.nombre);
```

El método **accept( )**, recibe como argumento un objeto **Cliente** y, a partir de él, creará un saludo con su nombre.

```
Cliente clie = new Cliente("123", "Jorge", 20);  
saludoClie.accept(clie); // se mostrará "Hola, Jorge"
```

## ALGUNAS INTERFACES FUNCIONALES DE LA API

A veces queremos que un objeto **Consumer** actúe sobre un conjunto de instancias de una determinada clase. Para ello se usa el método **forEach( )**, de la interfaz **Iterable<T>**,

```
default void forEach(Consumer<? Super T> accion)
```

Este método podrá ser llamado por cualquier objeto que implemente **Iterable**, como, por ejemplo, las colecciones **ArrayList**, **LinkedList**, **HashSet**, **TreeSet** o **LinkedHashSet**. El método lo recorrerá y realizará la acción “para cada” (for each, en inglés) uno de sus elementos.

## ALGUNAS INTERFACES FUNCIONALES DE LA API

Por ejemplo, si queremos saludar a todos los clientes de **listaClientes**,

```
List<Cliente> listaClientes = new ArrayList<>( );  
listaClientes.add(new Clientes("111", "Marta", "12/02/2000"));  
listaClientes.add(new Clientes("115", "Jorge", "16/03/1999"));  
listaClientes.add(new Clientes("112", "Carlos", "01/10/2002"));  
listaClientes.add(new Clientes("211", "Ana", "07/12/2001"));  
listaClientes.forEach(saludoClie);
```

La API proporciona otras interfaces funcionales importantes que iremos viendo.

## **ALGUNAS INTERFACES FUNCIONALES DE LA API**

Una particularidad de las clases anónimas y de las expresiones lambda (en realidad, de todas las clases llamadas locales, cuyo estudio no lo abordaremos en este curso) es que dentro de ellas se pueden usar variables locales del ámbito donde está definida la expresión, es decir, dentro del mismo bloque de sentencias.

Por ejemplo, en el siguiente código, se puede usar la variable `x` en la expresión lambda, pero no la `y`:

## ALGUNAS INTERFACES FUNCIONALES DE LA API

```
int y = 5;
{
    int x = 6;
    Function<Integer, Integer> f = a -> a + x; // Correcto
    Function<Integer, Integer> g = a -> a + y; // ¡Error!
}
```

Sin embargo, la variable local que se incluya en una expresión lambda (en nuestro caso, la **x**) debe ser una constante, bien declarada con el modificador **final**, o bien “efectivamente inmutable”, que significa que, aunque no se haya declarado **final**, actúa como si lo fuera.

## ALGUNAS INTERFACES FUNCIONALES DE LA API

Es decir que, una vez declarada e inicializada, no cambia su valor dentro de su ámbito de existencia, ya sea antes de la expresión lambda, dentro de ella o después de ella.

```
int x = 6;
```

```
x++; // ¡Error!
```

```
Function<Integer, Integer> f = a -> a + x++; // ¡Error!
```

```
x=10; // ¡Error!
```



## **ALGUNAS INTERFACES FUNCIONALES DE LA API**

**Actividad Resuelta 13.4.** Implementar un método estático al que se pasa como parámetro una tabla de tipo genérico y un predicado. El método devuelve otra tabla con los elementos de la tabla original que verifiquen la condición del predicado. Aplicarlo a una tabla de 50 enteros entre 1 y 100, que devuelva los múltiplos de 3.

## ALGUNAS INTERFACES FUNCIONALES DE LA API

**Actividad Resuelta 13.5.** Implementar el método estático

`static <T, V> V[] transformar(T[] original, V[] transf, Function<T, V> f)`

al que se pasan dos tablas de tipo **T** y **V** respectivamente, y devuelve la segunda tabla con los elementos de la primera transformados mediante la función que va en el tercer parámetro. Escribir un programa donde se usa este método para obtener una tabla con las raíces cuadradas de los elementos de otra.

## ALGUNAS INTERFACES FUNCIONALES DE LA API

**Actividad Resuelta 13.6.** Implementar el método estático

```
static <T> paraCada(T[ ] tabla, Consumer<T> c)
```

Similar a **forEach** (que no existe para tablas). Este método ejecuta en cada elemento de la tabla la acción implementada en el objeto **Consumer**.

Usarlo para mostrar por pantalla los nombres y edades de los **Cientes** de una tabla.

## ALGUNAS INTERFACES FUNCIONALES DE LA API

### Referencias a métodos:

A partir de la versión 8 de Java, es posible trabajar con referencias a métodos ya definidos en alguna clase. Cuando hemos implementado la interfaz **Function**, hemos pasado la función **apply( )** como expresión lambda, es decir, como método de una clase anónima.

Pero cuando la función ya está implementada en un método de alguna clase, como ocurre con **Math.sqrt( )**, tenemos una forma aún más corta de escribirla: como referencia al método.

## ALGUNAS INTERFACES FUNCIONALES DE LA API

### Referencias a métodos:

Una referencia a **Math.sqrt( )** se escribe

`Math::sqrt`

y se puede colocar en lugar de la expresión lambda,

`x -> Math.sqrt(x)`

## ALGUNAS INTERFACES FUNCIONALES DE LA API

### Referencias a métodos:

Entonces, para calcular raíces cuadradas de valores *Double*, podemos implementar

```
Function<Double, Double> raiz = Math::sqrt;
```

Para calcular una raíz cuadrada, pondríamos

```
Double x = raiz.apply(9.); // devolvería 3.0
```

## ALGUNAS INTERFACES FUNCIONALES DE LA API

### Referencias a métodos:

Las referencias a métodos se escriben poniendo el nombre de la clase, seguido de “: :” y el nombre del método (sin paréntesis ni lista de argumentos) cuando este es estático.

Si no es estático, en vez del nombre de la clase pondremos una referencia a un objeto de la clase donde está definido el método. En nuestro caso, hemos escrito una referencia al método estático **sqrt( )**, definido en la clase **Math** de la API.

## **ALGUNAS INTERFACES FUNCIONALES DE LA API**

### **Referencias a métodos:**

A primera vista puede parecer extraña la idea de una referencia a una función. Pero, cuando el sistema va a ejecutar un programa, antes carga su contenido en la memoria, de donde luego va leyendo y ejecutando sentencia a sentencia.

Por tanto, un método que forma parte de una aplicación que se va a ejecutar ocupa un cierto bloque de memoria. Cuando pasamos como parámetro la referencia de un método, lo que estamos pasando es la referencia del bloque de memoria donde está su código.



## ALGUNAS INTERFACES FUNCIONALES DE LA API

### Referencias a métodos:

Como podemos ver, el método referenciado (en este caso **sqrt( )**) no tiene por qué tener el mismo nombre del método “esperado” (**apply( )**). Basta con que los parámetros de entrada y el tipo devuelto sean compatibles.

A la hora de asignar a una variable de tipo **Function** (o cualquier otra interfaz funcional) una referencia a un método, este puede estar implementado en una clase cualquiera.

## ALGUNAS INTERFACES FUNCIONALES DE LA API

### Referencias a métodos:

Por ejemplo, definamos los métodos **cuadrado( )** y **cubo( )** en la clase **Calculos**.

```
Class Calculos{  
    Integer cuadrado(Integer a){  
        return a*a;  
    }  
    static Integer cubo(Integer x){  
        return x*x*x;  
    }  
}
```

## ALGUNAS INTERFACES FUNCIONALES DE LA API

### Referencias a métodos:

Cualquiera de ellos puede ser asignado a una variable de tipo **Function**, ya que su estructura de parámetros de entrada y tipo devuelto es compatible con el método **apply( )** definido en la interfaz. Se accede al método estático por medio del nombre de la clase y al no estático a través de un objeto creado previamente.

```
Function<Integer, Integer> f1 = Calculos::cubo;  
Calculos calc = new Calculos( );  
Function<Integer, Integer> f2 = calc::cuadrado;
```

## ALGUNAS INTERFACES FUNCIONALES DE LA API

### Referencias a métodos:

No obstante, si se trata de un método no estático de la propia clase a la que pertenece el valor al que se aplica, se puede invocar con el nombre de la clase, sin necesidad de crear un nuevo objeto. Por ejemplo, si implementamos la clase **Entero**.

# ALGUNAS INTERFACES FUNCIONALES DE LA API

## Referencias a métodos:

```
public class Entero{

    Integer valor;
    public Entero(Integer valor) {
        this.valor = valor;
    }
    Entero siguiente( ) {
        return new Entero(valor +1);

    }

    @Override
    public String toString( ) {
        return "Entero{" + "valor=" + valor + "}";
    }

}
```

## ALGUNAS INTERFACES FUNCIONALES DE LA API

### Referencias a métodos:

A partir de ella podemos definir la función **siguienteEntero**, que nos devuelve un objeto con valor incrementado en 1.

```
Function<Entero, Entero> siguienteEntero = Entero::siguiente;  
System.out.println(siguienteEntero.apply(new Entero(3))); // 4
```

Vemos que la referencia al método no estático **siguiente** se hace a través del nombre de la clase **Entero**.

## ALGUNAS INTERFACES FUNCIONALES DE LA API

### Referencias a métodos:

Esta circunstancia no se daba en el ejemplo anterior, donde los datos eran de tipo **Integer** y los métodos pertenecían a la clase **Calculos**. Sin embargo, la encontraremos frecuentemente en los **Stream**.

## ALGUNAS INTERFACES FUNCIONALES DE LA API

### Referencias a métodos:

Veamos un ejemplo un poco más elaborado de utilización de referencias a métodos. Vamos a implementar un método estático que aplica una transformación **m** a todos los elementos de una tabla, que también se el pasa como parámetro.

```
Static <T> void aplicar(T[ ] tabla, Function<T,T> m) {  
    for (int i = 0; i < tabla.length; i++) {  
        tabla[i] = m.apply(tabla[i]);  
    }  
}
```



## ALGUNAS INTERFACES FUNCIONALES DE LA API

### Referencias a métodos:

Para probarlo, le pasaremos una tabla de enteros que deberá elevar al cuadrado con nuestro método **cuadrado( )** definido en la clase **Calculos**.

```
Integer[ ] t = {1, 2, 3, 4, 5}  
aplicar(t, f2); // o bien: aplicar(t, calc::cuadrado);  
System.out.println(Arrays.toString(t));
```

Obtendríamos por pantalla: **[1 4 9 16 25]**

## ALGUNAS INTERFACES FUNCIONALES DE LA API

### Referencias a métodos:

Obsérvese que los nombres de los métodos son **cuadrado( )** o **cubo( )**, no **apply( )**. Igual que pasa con las expresiones lambda, Java infiere del tipo del parámetro de entrada *m* (la interfaz *Function*), que ambos métodos deben identificarse con **apply( )**.

Naturalmente, para que esto sea posible, los parámetros de entrada y el tipo devuelto de los métodos referenciados (**cuadrado( )** o **cubo( )**), tienen que ser compatibles con la definición de **apply( )**.

## ALGUNAS INTERFACES FUNCIONALES DE LA API

### Referencias a métodos:

Todo esto es extensible a cualquier interfaz funcional, ya sea de la API o creada por nosotros mismos.

También se pueden usar referencias a constructores. En este caso, la sintaxis es un poco especial. Como el constructor tiene el mismo nombre que la clase, cabría esperar algo así como **Cliente::Cliente**, pero en realidad es **Cliente::new**.

## ALGUNAS INTERFACES FUNCIONALES DE LA API

### Referencias a métodos:

Como ejemplo, podríamos implementar la interfaz Function para construir objetos de la clase Saludo.

```
class Saludo {  
    String nombre;  
    Saludo(String nombre) {  
        this.nombre = nombre;  
    }  
    public String toString( ) {  
        return "Hola, " + nombre;  
    }  
}
```

## ALGUNAS INTERFACES FUNCIONALES DE LA API

### Referencias a métodos:

El método **apply( )** de la interfaz **Function** recibirá una cadena con el nombre, y deberá construir y devolver un objeto **Saludo** con ese nombre.

```
Function<String, Saludo> construyeSaludo = Saludo::new;  
Saludo s = construyeSaludo.apply("Claudia");  
System.out.println(s); // ¡Hola Claudia!
```

A la hora de ejecutar **apply( )**, Java busca el constructor en la clase **Saludo** y lo ejecuta pasando el valor <<Claudia>> como parámetro.

## ALGUNAS INTERFACES FUNCIONALES DE LA API

### Referencias a métodos:

**Actividad Resuelta 13.7.** Añadir a la clase **Calculos** el método.

```
static Double raiz3(Double x)
```

que calcula la raíz cúbica de **x**.

Con el método **transformar( )**, implementado en la actividad resuelta 13.5, obtener una tabla con las raíces cúbicas de los elementos de una tabla de números reales que se le pasa como parámetro.

## ALGUNAS INTERFACES FUNCIONALES DE LA API

### Referencias a métodos:

**Actividad Resuelta 13.8.** Definir la interfaz **Funcion2**, donde se declara el método abstracto.

U operar(T a, V b) ,

que admite dos parámetros, de tipo **T** y **V** respectivamente, y devuelve un resultado de tipo **U**.

## ALGUNAS INTERFACES FUNCIONALES DE LA API

### Referencias a métodos:

Implementar el método estático

```
static <T, V, U> U[ ] operarTablas(T[ ] op1, V[ ] op2, U[ ] resultado,  
Funcion2<T, V, U> f)
```

al que se pasan dos tablas, **op1** y **op2**, y devuelve otra tabla cuyos elementos son el resultado de operar los elementos correspondientes de **op1** y **op2** utilizando el método implementado en **f**.



## ALGUNAS INTERFACES FUNCIONALES DE LA API

### Referencias a métodos:

Añadir a **Calculos** el método **producto( )**, que devuelve el producto de los valores reales que se le pasan como parámetros. Usar el método **operar( )** para multiplicar los valores de dos tablas de tipo **Double**.

## INTERFAZ STREAM

Los objetos de las clases que implementan la interfaz **Stream**, son sucesiones de objetos sobre los que se puede realizar una serie de operaciones, que pueden ir encadenadas hasta dar un resultado final. Dichas operaciones realizadas con un **Stream** pueden ser de dos tipos:

- **Intermedias:** dan como resultado un nuevo **Stream**, al que se le pueden seguir aplicando nuevas operaciones.
- **Terminales:** dan un resultado final, numérico o de otro tipo, pero no un **Stream**.

## INTERFAZ STREAM

La idea es crear, a partir de una colección o una tabla, o bien explícitamente, un **Stream** al que se aplican operaciones intermedias encadenadas (es lo que se conoce como una tubería o pipeline), obteniendo un resultado final por medio de una operación terminal.

La ventaja de crear el **Stream** es que dispone de muchas más operaciones para procesar sus datos que las colecciones o las tablas.

## INTERFAZ STREAM

Los **Stream** son objetos que implementan la interfaz **Stream**. Por tanto, la clase **Stream** no existe y los objetos **Stream** no se pueden crear con un constructor, sino llamando a alguna de las funciones implementadas para ello.

Se dice que las operaciones sobre **Stream** son agregadas y se inspiran en las operaciones globales de las colecciones, ya que operan sobre la totalidad del **Stream**. Mucha de ellas hacen uso de interfaces funcionales de la API, de las que hemos visto algunas ya. De hecho, los **Stream** se han diseñado para trabajar con expresiones lambda.

## **INTERFAZ STREAM**

**Aclaración:** Se llaman operaciones agregadas a aquellas que operan sobre la totalidad de un **Stream**, permitiendo la ejecución en paralelo, transparente al programador, para aumentar la velocidad del proceso.

## INTERFAZ STREAM

### Formas de crear un Stream:

Hay diversas formas de obtener un **Stream** inicial, es decir, que no proceda de otro **Stream**. Nosotros vamos a ver cuatro.

- **A partir de una colección:** llamando al método **stream( )**, definido en las clases de tipo **Collection**.

**Stream<T> nombreStream = nombreColeccion.stream( );**

## **INTERFAZ STREAM**

### **Formas de crear un Stream:**

- **A partir de una tabla de tipo T[ ]:** llamando al método **of( )**, de la interfaz **Stream**, con la tabla como parámetro.

**Stream<T> nombreStream = Stream.of(tabla);**

- **A partir de una tabla de tipo T[ ]:** usando el método **stream( )**, de la clase **Arrays**, con la tabla como parámetro.

**Stream<T> nombreStream = Arrays.stream(tabla);**

## **INTERFAZ STREAM**

### **Formas de crear un Stream:**

- **Inicializándolo directamente:** también con el método **of( )**, de **Stream**, pero pasándole como lista de parámetros los valores de tipo **T**, que lo inicializan.

**Stream<T> nombreStream = Stream.of(val1, val2, ... );**



## **INTERFAZ STREAM**

### **Formas de crear un Stream:**

Supongamos que queremos trabajar con los elementos de una lista. Para verlo con un caso práctico, vamos a empezar creando una lista de cadenas:

```
List<String> lista = new ArrayList<>( );  
lista.add("dado");  
lista.add("arte");  
lista.add("bola");  
lista.add("asa");
```

## INTERFAZ STREAM

### Formas de crear un Stream:

```
lista.add("buzo");  
lista.add("coche");  
lista.add("barco");  
lista.add("duna");
```

A partir de ella, creamos un **Stream** de cadenas por el primer método:

```
Stream<String> streamCad = lista.stream( );
```

## **INTERFAZ STREAM**

### **Formas de crear un Stream:**

**streamCad** contiene una copia de todos los datos de la lista, no una referencia a los originales. Por tanto, los cambios que se hagan en el Stream no se van a reflejar en la lista original, que permanecerá intacta.

Una de las cosas que podemos hacer con los elementos de un Stream es filtrarlos. Para ello se usa el método

**Stream<T> filter(Predicate<? Super T> pred)**

## INTERFAZ STREAM

### Formas de crear un Stream:

Invocado desde el **Stream** original, se le pasa un predicado que se aplicará a todos los elementos del **Stream**. Solo aquellos que devuelvan **true** formarán parte del nuevo **Stream** devuelto por el método.

Naturalmente, **filter( )** es un método intermedio, ya que devuelve un nuevo **Stream**, susceptible de llamar a nuevos métodos para producir nuevas transformaciones.

## INTERFAZ STREAM

### Formas de crear un Stream:

Por ejemplo, si queremos obtener, a partir de **streamCad**, un nuevo **Stream** con los elementos que empiezan por “a”, crearemos el predicado

```
Predicate<String> empiezaPorA = s -> s.startsWith("a");
```

## INTERFAZ STREAM

### Formas de crear un Stream:

donde se ha invocado al método **startsWith( )** de la clase **String**. Este predicado se le pasa como argumento al método **filter( )**, invocado por **streamCad**, y devuelve un nuevo **Stream** con los elementos filtrados.

```
Stream<String> streamA = streamCad.filter(empiezaPorA);
```

## INTERFAZ STREAM

### Formas de crear un Stream:

Ahora **streamA** contiene aquellos elementos del **Stream** original que empiezan por “a”. En realidad, lo más común es que el filtro solo se tenga que aplicar una vez. Por tanto, generalmente no merece la pena crear una variable para el predicado. Lo normal es pasarlo como argumento directamente, en forma de expresión lambda, al método **filter( )**.

```
Stream<String> streamA = streamCad.filter(s -> s.startsWith("a"));
```

## INTERFAZ STREAM

### Formas de crear un Stream:

Si queremos ver los resultados obtenidos hasta ahora, tendremos que aplicar una nueva operación, ya que no existe una función **toString( )** para **Stream**. Es decir, no podemos escribir

```
System.out.println(streamA);
```

Para que todos los elementos de un **Stream** se muestren por pantalla, deberemos hacer que para cada uno de ellos se ejecute el método

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



## INTERFAZ STREAM

### Formas de crear un Stream:

Siempre que queramos que se ejecute una determinada acción "para cada" elemento de un **Stream**, usaremos el método

**void forEach(Consumer<? Super T> accion)**

donde **T** es el tipo genérico del **Stream** que invoca el método. El parámetro **accion** es un **Consumer** que lleva encapsulado el método **accept( )**, que se tiene que ejecutar para todos y cada uno de los elementos del **Stream**.

## INTERFAZ STREAM

### Formas de crear un Stream:

Como puede verse, **forEach( )** no devuelve otro **Stream** (de hecho, no devuelve nada), por lo cual es un método terminal. Si queremos mostrar por pantalla todos los elementos de **streamA**, llamamos a **forEach( )** pasándole como argumento un **Consumer** que muestre cadenas por pantalla

```
Consumer<String> mostrar = s -> System.out.println(s);  
streamA.forEach(mostrar);
```

## **INTERFAZ STREAM**

### **Formas de crear un Stream:**

o más brevemente

```
streamA.forEach(s -> System.out.println(s)); /* se mostrará "arte" y "asa"*/
```

o incluso, usando referencias a métodos

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

## INTERFAZ STREAM

### Formas de crear un Stream:

Una cosa **muy importante** que debemos tener en cuenta con los **Stream** es que no son reusables, es decir, cada operación intermedia sobre un **Stream** nos devuelve un **Stream** transformado, pero el **Stream** original se pierde. Por ejemplo, si después de obtener **streamA** con los elementos filtrados a partir de **streamCad** intentamos volverá utilizar este último para filtrar los elementos que empiezan por “b”,

```
streamCad.filter(s -> s.startsWith("b")).forEach(System.out::println);
```

## INTERFAZ STREAM

### Formas de crear un Stream:

Salta la excepción **java.lang.IllegalStateException**, con la descripción

**stream has already been operated upon or closed**

Es decir, ya se ha operado antes sobre **stream** y no se puede volver a usar. Podemos aplicar un nuevo método al **Stream** devuelto, formando una tubería (como veremos en el siguiente apartado), pero no podemos volver a usar el **Stream** original.

## INTERFAZ STREAM

### Formas de crear un Stream:

Todo esto deberá tenerse en cuenta a la hora de probar las distintas funciones que estamos viendo, ya que un **Stream** usado con una función no puede ser reutilizado para probar otra. Si queremos hacerlo, deberemos volver a crearlo desde el principio a partir de la colección o de la tabla original.

## INTERFAZ STREAM

### Tuberías o pipelines:

Si de lo que se trataba era de mostrar por pantalla los elementos que empiezan por “a”, podríamos haber prescindido de la variable intermedia **streamA** y haber encadenado las dos operaciones para formar lo que se llama una tubería, que no es más que un **Stream** fuente (creado a partir de una colección, de una tabla o por otro medio) al que se aplica una serie de operaciones intermedias encadenadas y se acaba con una operación terminal.

## **INTERFAZ STREAM**

### **Tuberías o pipelines:**

En el ejemplo anterior, podíamos haber escrito

```
lista.stream( ).filter(s -> s.startsWith("a")).forEach(System.out::println);
```

Las tuberías, a menudo, son largas y no caben en una sola línea del editor. Además, la lectura puede ser incómoda.



## INTERFAZ STREAM

### Tuberías o pipelines:

Por eso es costumbre poner cada operación en una línea.

```
lista.stream( )  
    .filter(s -> s.startsWith("a"))  
    .forEach(System.out::println);
```

El **Stream** del ejemplo lo obtuvimos a partir de una lista. También podemos obtener un **Stream** a partir de una tabla con el método estático **of( )** de la interfaz **Stream**.

## **INTERFAZ STREAM**

### **Tuberías o pipelines:**

Para ver un ejemplo, vamos a crear una tabla de clientes.

```
Cliente[ ] tClie = {  
    new Cliente("111", "Marta", "12/02/2000"),  
    new Cliente("115", "Jorge", "16/03/1999"),  
    new Cliente("112", "Carlos", "01/10/2002"),  
    new Cliente("211", "Ana", "07/12/2001") };
```

## **INTERFAZ STREAM**

### **Tuberías o pipelines:**

y, a partir de ella, obtendremos un **Stream** por cualquiera de los métodos aludidos

```
Stream<Cliente> streamClie = Stream.of(tClie);
```

o bien

```
Stream<Cliente> streamClie = Arrays.stream(tClie);
```

## INTERFAZ STREAM

### Tuberías o pipelines:

Como los **Stream** no son reutilizables, tiene poco sentido crear la variable **streamClic**. Lo habitual es escribir las tuberías completas, incluyendo la lista o la tabla iniciales cada vez. Así lo haremos con las nuevas operaciones de agregación que vamos a estudiar.

Una muy importante es ordenar los elementos de un Stream por medio del método

```
Stream<T> sorted( );
```

## **INTERFAZ STREAM**

### **Tuberías o pipelines:**

que devuelve un nuevo Stream con los elementos ordenados según su orden natural.

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

mostrará los clientes ordenados por DNI.

## INTERFAZ STREAM

### Tuberías o pipelines:

El método **sorted( )** está sobrecargado y puede admitir como parámetro un comparador para especificar el criterio de ordenación de los elementos. Por ejemplo, si queremos que los clientes se ordenen por nombre, definimos el comparador:

```
Comparator<Cliente> comp = (x, y) -> x.nombre.compareTo(y.nombre);
```

## **INTERFAZ STREAM**

### **Tuberías o pipelines:**

con lo cual

```
Arrays.stream(tClie)  
    .sorted(comp)  
    .forEach(System.out::println);
```

## **INTERFAZ STREAM**

### **Tuberías o pipelines:**

o bien, prescindiendo de la variable **comp**

```
Arrays.stream(tClie)  
    .sorted((x, y) -> x.nombre.compareTo(y.nombre))  
    .forEach(System.out::println);
```

muestra los clientes ordenados por nombres.



## INTERFAZ STREAM

### Tuberías o pipelines:

A partir de un **Stream** podemos obtener otro cuyos elementos se corresponden uno a uno con los del **Stream** original, pero con una determinada transformación. Por ejemplo, puede interesarnos un **Stream** con los DNI de los clientes, en el mismo orden en que aparecen en el **Stream** original. Esa tarea la lleva a cabo el método

**Stream<V> map(Function<? Super T, ? extends V> mapper)**

## INTERFAZ STREAM

### Tuberías o pipelines:

A pesar de lo aparatoso de la expresión, es fácil de usar. El método recibe como parámetro una función que transforma los elementos del **Stream** original del tipo **T** y devuelve un **Stream** con los elementos transformados, de tipo **V**. En el ejemplo propuesto, necesitamos una función (en realidad, el método abstracto **apply( )**, que ya vimos) que reciba un objeto **Cliente** y devuelve su DNI. La expresión lambda correspondiente será:

```
Function<Cliente, String> aDni = c -> c.dni;
```

## INTERFAZ STREAM

### Tuberías o pipelines:

que transforma un objeto **c** del tipo **Cliente** en su DNI, de tipo **String**. Por tanto, prescindiendo de la variable **aDni**, podemos escribir

```
Arrays.stream(tClie)
    .map(c -> c.dni)
    .forEach(System.out::println);
```

que mostrará los DNI de todos los elementos de la tabla de clientes.

## INTERFAZ STREAM

### Tuberías o pipelines:

El método terminal

**long count( )**

nos devuelve el número de elementos de un **Stream**. Por ejemplo,

```
long n = Arrays.stream(tClie)
    .filter(c -> c.fechaNacimiento.isAfter(LocalDate.of(2000, 12, 31)))
    .count( )
```

## INTERFAZ STREAM

### Tuberías o pipelines:

devuelve 2, el número de clientes nacidos después de 2000.

Vamos a crear ahora un **Stream** de enteros inicializándolo de forma explícita.

```
Stream<Integer> streamEnteros = Stream.of(4, 3, 7, 1, 0, 8, 9, 3, 5, 4, 2,  
1, 4, 6, 8, 1, 0, 2, 3);
```

## INTERFAZ STREAM

### Tuberías o pipelines:

Una de las cosas que podemos hacer es eliminar los elementos repetidos. Para ello existe el método

**Stream<T> distinct( )**

que devuelve un nuevo **Stream** sin repeticiones,

```
Stream.of(4, 3, 7, 1, 0, 8, 9, 3, 5, 4, 2, 1, 4, 6, 8, 1, 0, 2, 3)  
    .distinct( )  
    .forEach(x -> System.out.println(x + " "));
```

## INTERFAZ STREAM

### Tuberías o pipelines:

mostrará por pantalla

**4 3 7 1 0 8 9 5 2 6**

A menudo querremos obtener un valor como resultado de cálculos con los elementos de un **Stream**. Para ello disponemos de los métodos de reducción, como **sum( )**, **average( )** o **reduce( )**.

## **INTERFAZ STREAM**

### **Tuberías o pipelines:**

Por ejemplo, si queremos obtener la suma de las edades de los clientes de **tClie**.

```
int sumaEdades = Arrays.stream(tClie)
    .mapToInt(c -> c.edad( )) /* devuelve Stream de objetos Integer*/
    .sum( );
System.out.println(sumaEdades);
```



## INTERFAZ STREAM

### Tuberías o pipelines:

o el promedio de las edades, por medio de un **Stream** especial para enteros,

```
double mediaEdades = Arrays.stream(tClie)
    .mapToInt(Cliente::edad) /* devuelve un IntStream, que es un Stream especial de enteros*/
    .average( );
    .getAsDouble( ); /* necesario, porque average( ) devuelve un objeto
                        OptionalDouble, no Double*/
System.out.println(mediaEdades);
```

## INTERFAZ STREAM

### Tuberías o pipelines:

Además de **sum( )** y **average( )**, **IntStream** dispone de otras operaciones, como **max( )** (valor máximo), **min( )** (valor mínimo) o **skip(long n)**, que devuelve un nuevo **Stream** resultante de descartar los **n** primeros elementos.

El método **reduce( )** es más general. Permite hacer operaciones que impliquen algún tipo de acumulación.

## **INTERFAZ STREAM**

### **Tuberías o pipelines:**

Por ejemplo, podemos calcular la suma de las edades de la siguiente forma:

```
int sumaEdades = Arrays.stream(tClie)
    .map(Cliente::edad)
    .reduce(0, (a, b) -> a + b);
```

## INTERFAZ STREAM

### Tuberías o pipelines:

donde el primer parámetro es el valor inicial de la acumulación y también el valor por defecto, que se devuelve si el **Stream** está vacío. El segundo parámetro es el criterio de acumulación, que en nuestro caso es la suma.

También podemos concatenar dos **Stream** con el método estático definido en la interfaz **Stream**,

```
static Stream<T> concat(Stream<? extends T> prim, Stream<? extends T> seg)
```

## INTERFAZ STREAM

### Tuberías o pipelines:

que devuelve un nuevo **Stream** con los elementos del segundo a continuación de los del primero. Por ejemplo, si creamos un nuevo **Stream** de enteros, **streamNuevo**, y lo concatenamos con **streamEnteros** sin repeticiones,

```
Stream<Integer> streamNuevo = Stream.of(-1, -6, -3, -3)
Stream.concat(streamEnteros, streamNuevo)
    .distinct( )
    .forEach(x -> System.out.print(x + " "));
```

## **INTERFAZ STREAM**

### **Tuberías o pipelines:**

obtendremos por pantalla

**4 3 7 1 0 8 9 5 2 6 -1 -6 -3**

A menudo nos interesará crear una tabla con los elementos de un **Stream**. Para ello disponemos del método

**Object[ ] toArray( )**

## **INTERFAZ STREAM**

### **Tuberías o pipelines:**

Por ejemplo, si queremos una tabla con los números pares sin repetir,

```
Object[ ] tObject = Stream.of(-1, -6, -3, -3, 2, 4, 2, -1)
    .distinct( )
    .filter(x -> x % 2 == 0)
    .toArray( );
```

## INTERFAZ STREAM

### Tuberías o pipelines:

Para transformar la tabla de tipo **Object[ ]** en una de tipo **Integer[ ]**, podemos usar el método **copyOf( )** de la clase **Arrays**, sobrecargada con una versión que admite como último parámetro la clase de la tabla destino.

```
Integer[ ] tInt = Arrays.copyOf(tObject, tObject.length, Integer[ ].class);
```



## INTERFAZ STREAM

### Tuberías o pipelines:

Sin embargo, el método **toArray( )** de la interfaz **Stream** también está sobrecargado con una versión que admite como parámetro un método que construya la tabla del tipo que deseemos (en nuestro caso usaremos un constructor), con lo cual nos ahorramos la transformación con **copyOf( )**.

## **INTERFAZ STREAM**

### **Tuberías o pipelines:**

```
Integer[ ] tInt = Stream.of(-1, -6, -3, -3, 2, 4, 2, -1)
    .distinct( )
    .filter(x -> x % 2 == 0)
    .toArray(Integer[ ]::new);
    // constructor de tabla de entero
System.out.println(Arrays.toString(tInt));
```

## INTERFAZ STREAM

### Tuberías o pipelines:

También podemos agrupar los elementos de un **Stream** en una colección, un mapa o una cadena. O hacer estadísticas de sus datos. Todo esto se consigue con el método **collect( )**. Es tan rico como complejo y no vamos a estudiarlo aquí a fondo. Solo veremos algunas aplicaciones sencillas que resultan muy útiles. En todos los casos se le pasa como parámetro un objeto de la clase **Collector**, que se obtiene a partir de distintos métodos de la clase **Collectors**. Por ejemplo, si queremos una lista con los valores de un **Stream**, pasamos como argumento el colector devuelto por **Collectors.toList( )**.

## **INTERFAZ STREAM**

### **Tuberías o pipelines:**

```
List<Integer> listaNumeros = Stream.of(2, 5, 1, 4, -6, -3, -3)  
    .collect(Collectors.toList( ));
```

También podemos extraer un conjunto, en vez de una lista:

```
Set<Integer> conjuntoNumeros = Stream.of(5, 1, 2, 6, 3, 9, 4, 1, 7, 3, 5)  
    .collect(Collectors.toSet( ));
```

## INTERFAZ STREAM

### Tuberías o pipelines:

con lo cual se eliminan automáticamente las repeticiones, resultando

**[1, 3, 5, 7, 9]**

Se puede escoger una implementación concreta de lista o de conjunto. Por ejemplo, si queremos un conjunto ordenado, usaremos **Collectors.toCollection(TreeSet::new)**.

```
Set<Integer> conjuntoNumeros = Stream.of(5, 1, 2, 6, 3, 9, 4, 1, 7, 3, 5)  
    .collect(Collectors.toCollection(TreeSet::new));
```

## **INTERFAZ STREAM**

### **Tuberías o pipelines:**

Cualquier elemento que se inserte en **conjuntoNumeros** lo hará manteniendo el orden natural.

```
conjuntoNumeros.add(-5);  
conjuntoNumeros.add(13);  
System.out.println(conjuntoNumeros);
```

Se mostrará:

```
[-5, 1, 3, 5, 7, 9, 13]
```

## INTERFAZ STREAM

### Tuberías o pipelines:

Volvamos al **Stream** de clientes. Si queremos crear un mapa de los DNI (claves) sobre los nombres (valores) de los clientes, usaremos **Collectors.toMap( )** y deberemos especificar los atributos clave y el valor, por ese orden.

```
Map<String, String> mapaClientes = Stream.of(tClie)
    .collect(Collectors.toMap(c -> c.dni, c -> c.nombre));
```

obteniéndose el mapa

```
{111=Marta, 211=Ana, 112=Carlos, 115=Jorge}
```

## **INTERFAZ STREAM**

### **Tuberías o pipelines:**

Con **Collectors.averagingInt( )** podemos calcular el promedio de las edades

```
double edadMedia = Stream.of(tClie)
                        .collect(Collectors.averagingInt(c -> c.edad));
```

o una estadística general de las edades

```
IntSummaryStatistics sumarioEdad =
streamClie.collect(Collectors.summarizingInt(c -> c.edad));
```



## **INTERFAZ STREAM**

### **Tuberías o pipelines:**

donde **IntSummaryStatistics** es una clase capaz de calcular diversos parámetros estadísticos.

Podemos ejecutar

```
System.out.println(sumarioEdad);
```

y obtenemos por pantalla un sumario de dichos parámetros.

```
IntSummaryStatistics{count=4, sum=78, min=18, average=19.500000, max=21}
```

## INTERFAZ STREAM

### Tuberías o pipelines:

El método **Collectors.joining( )** permite concatenar los elementos de un **Stream** de cadenas, escogiendo el separador y, de forma optativa, un prefijo y un sufijo. Por ejemplo, con un solo parámetro (el separador)

```
String nombres1 = Arrays.stream(tClie)
    .map(c -> c.nombre)
    .collect(Collectors.joining(", "));
/* separados por comas */
System.out.println(nombres1);
```

## **INTERFAZ STREAM**

### **Tuberías o pipelines:**

mostraría por pantalla

**Marta, Jorge, Carlos, Ana**

En cambio, añadiendo un parámetro para el prefijo y otro para el sufijo

```
String nombres2 = Arrays.stream(tClie)  
    .map(c -> c.nombre)  
    .collect(Collectors.joining(", ", "Nombres: [", "]"));  
System.out.println(nombres2);
```

## **INTERFAZ STREAM**

### **Tuberías o pipelines:**

mostraría

**Nombres: [Marta, Jorge, Carlos, Ana]**

**Actividad Resuelta 13.9.** Implementar el método

**static boolean esPrimo(Integer n),**

que devuelve **true** si **n** es primo y **false** en caso contrario.

## **INTERFAZ STREAM**

### **Tuberías o pipelines:**

Escribir un programa que genere 100 números aleatorios menores que 1000 y que muestre por pantalla todos los que son primos:

1. Ordenados de menor a mayor.
2. Ordenados de mayor a menor.
3. Solo los comprendidos entre 200 y 800.

## **INTERFAZ STREAM**

### **Tuberías o pipelines:**

**Actividad Resuelta 13.10.** Repetir el Apartado 1 de la Actividad resuelta 13.9, pero, en vez de mostrar los números, se devuelven en:

1. Una lista.
2. Una tabla.

## **INTERFAZ STREAM**

### **Tuberías o pipelines:**

**Actividad Resuelta 13.11.** A partir de la tabla de clientes **tClie**, mostrar un listado de los clientes, donde aparezcan sus nombres y edades, ordenados por nombre.

## INTERFAZ STREAM

### Tuberías o pipelines:

**Actividad Resuelta 13.12.** Crear una lista con 40 números enteros aleatorios entre -20 y 20. A partir de ella crear dos **Stream**, uno con los números positivos y otro con los negativos, todos ellos sin repetir.

Mostrar por pantalla el número de elementos de cada **Stream**. Crear otro **Stream** para contar los números que están comprendidos entre -10 y 10 incluidos, sin repeticiones.



## INTERFAZ STREAM

### Tuberías o pipelines:

**Actividad Resuelta 13.13.** A partir de la clase **Cliente**, crear la clase **DatosCliente** con los atributos **nombre** y **fechaNacimiento**, y los métodos:

- **DatosCliente(Cliente c).**
- **int edad( ).**
- **String toString( ),** que muestre nombre y edad.

## INTERFAZ STREAM

### Tuberías o pipelines:

Con los elementos de la tabla clientes **tClie**, construir un mapa que tenga como clave el atributo **dni** y como valor el objeto **DatosCliente** correspondiente. Mostrar el mapa por pantalla.