En la versión 8 de Java se incluyó una API que facilitaba en gran medida el trabajo con colecciones. Este nos permite realizar operaciones sobre la colección, como ejemplo por buscar, filtrar, reordenar, reducir, etc.

Dicho tipo de programación consiste, más que nada, en ahorrar tiempo y líneas de código, además de facilitar el entendimiento de un código. Para programar de esta forma usamos dos tipos de funciones, por una parte, las Lambdas y, por otra, las Method reference operator o también conocidas como Double colon (::). A continuación, te mostramos sus estructuras:

Lambdas

Consisten en tres partes: los parámetros, arrow operator (->) (en este caso el guion y el signo mayor, que deben estar seguidos, no pueden tener espacios entre sí), y el cuerpo de la función.

() -> System.out.println("Hello Functional World")

En este caso, los parámetros son nulos, por lo tanto, es necesario usar los paréntesis vacíos, agregar el arrow operator (->) y, finalmente, el cuerpo de la función que, en este caso, imprimirá el mensaje en la consola. En el caso de que tenga un solo parámetro se pueden obviar los paréntesis, a menos que se indique el tipo de la variable.

a -> System.out.println(a)  
(String a) -> System.out.println(a)

Si existe más de un parámetro los paréntesis son necesarios. En todos los casos indicados, el cuerpo solo cumple una sentencia, por esto no es necesario que termine en punto y coma (;). Pero, si tuviera más de una, sería necesaria la siguiente estructura:

a->{  
System.out.println(a);  
System.out.println(a.length());  
}

Si fuese necesario retornar algo y es una función de una sola sentencia, no es necesario especificar el return.

a -> a+1  
a -> { return a+1; }

En ambos casos, las líneas de código hacen lo mismo, retornando el valor a + 1, por lo que cumplen la misma función.

‍

‍

Double Colon (::)

Esta funciona de manera muy parecida a las Lambdas, sin embargo, el Double colon se utilizará en lugar de Lambdas en el caso de que la función tenga solo una sentencia y que dicha sentencia sea un método estático, un constructor o un método de una instancia. En este caso, los parámetros se obvian y solo se llama a la función

a -> System.out.println(a)  
System.out::println

Estas dos cumplen el mismo fin.

Ejemplo

Digamos que tenemos un array de Strings y quisiéramos mostrar en consola cada uno de los valores, con la programación funcional nos ahorraríamos algunas líneas:

ArrayList<String> names =  
Lists.newArrayList(  
"John", "John", "Mariam", "Alex", "Mohammado", "Mohammado", "Vincent", "Alex", "Alex");

Antes de la programación funcional se hubiese tenido que hacer de la siguiente manera:

for (String name : names) {  
System.out.println(name);  
}

Ahora sería algo así:

names.forEach(name->System.out.println(name));

Incluso podría acortarse aún más, así:

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

El resultado de lo anterior sería algo como esto:

John  
John  
Mariam  
Alex  
Mohammado  
Mohammado  
Vincent  
Alex  
Alex

‍

‍

API [Stream](https://educacionadistancia.juntadeandalucia.es/centros/jaen/mod/url/view.php?id=203776" \o "STREAM)

[Stream](https://educacionadistancia.juntadeandalucia.es/centros/jaen/mod/url/view.php?id=203776) es una librería que facilita mucho el trabajo con collections, como List y Sets, y consiste en solo llamar un método.

names.[stream](https://educacionadistancia.juntadeandalucia.es/centros/jaen/mod/url/view.php?id=203776)()

Luego de esto se llama al resto de las funciones. Ahora hablaré de aquellas que considero más útiles.

‍

‍**Filter**

Esta funcion es muy útil para filtrar acorde a una condición, la cual tiene que ser de tipo boolean. Siguiendo con el ejemplo de los nombres, digamos que necesitamos solo los nombres que contengan la letra “o”:

names.[stream](https://educacionadistancia.juntadeandalucia.es/centros/jaen/mod/url/view.php?id=203776)().filter(name-> name.contains("o")).forEach(System.out::println);

dando como resultado:

John  
John  
Mohammado  
Mohammado

Map

Este método es útil para cuando se quiere cambiar el tipo de respuesta, como convertir algún tipo instancia de una clase a otra. Digamos que se tiene la clase *person* y la clase *persona*

public class Person {  
Integer id;  
String firstName;  
String lastName;  
String email;  
String gender;  
Integer age;

// Getter y Setter o use Lombok :)  
  
};  
  
public class Persona {  
Integer id;  
String nombre;  
String apellido;  
String correo;  
String genero;  
Integer edad;  
  
  
// Getter y Setter o use Lombok :)  
};

En sí son la misma clase, solo que una tiene los atributos en ingles y otra en español. En el caso de que tengas una lista de person (para el ejemplo se llamara people) y necesitas una lista de persona sería algo como esto:

List<Persona> personas =  
people.[stream](https://educacionadistancia.juntadeandalucia.es/centros/jaen/mod/url/view.php?id=203776)()  
map(  
person -> {  
Persona persona = new Persona();  
  
persona.setId(person.getId());  
persona.setNombre(person.getFirstName());  
\*  
resto de los setters  
\*/  
eturn persona;  
})

.collect(Collectors.toList());

O digamos que, de la lista de nombres del ejemplo anterior, quisieras tener un arreglo (array) del largo (length) de cada nombre:

names.[stream](https://educacionadistancia.juntadeandalucia.es/centros/jaen/mod/url/view.php?id=203776" \o "STREAM)()  
.map(name -> name.length())  
.collect(Collectors.toList());

Aquí sin afectar al array de names utilizado en el ejemplo anterior, se puede obtener otro arreglo (array) con el respectivo largo (lenght) de cada nombre.

‍

‍**Sorted**

Esta función, cumple el rol de ordenar el arreglo (array) acorde a una regla de comparación.

names.[stream](https://educacionadistancia.juntadeandalucia.es/centros/jaen/mod/url/view.php?id=203776)()  
.sorted(Comparator.naturalOrder())  
.forEach(System.out::println);

En este caso usamos Comparator, que tiene varios métodos que nos ayudan a comparar dos objetos, por ejemplo, naturalOrder, que en este caso ordenaría los nombres en orden alfabético. También podemos usar reverseOrder que ordenará los nombres en sentido contrario. El output final sería:

Alex  
Alex  
Alex  
John  
John  
Mariam  
Mohammado  
Mohammado  
Vincent

Pero digamos que también quisiéramos hacer otro tipo de comparación, por ejemplo, con el largo del nombre. En este caso podríamos llamar a la función comparing que toma como parámetro una lambda.

names.[stream](https://educacionadistancia.juntadeandalucia.es/centros/jaen/mod/url/view.php?id=203776" \o "STREAM)()  
.sorted(Comparator.comparing(String::length))  
.forEach(System.out::println);

Lo anterior daría una respuesta como esta:

John  
John  
Alex  
Alex  
Alex  
Mariam  
Vincent  
Mohammado  
Mohammado

Por último, en caso de que deseara agregar n filtros más, por ejemplo, quisiera ordenar los nombres por su largo y luego ordenarlos por orden alfabético, entonces, primero compararía el largo del String y luego, en el caso de que los String fuesen de igual largo, compararía en orden alfabético. Para hacer esto, se tendría que hacer un llamado a thenComparing:

names.[stream](https://educacionadistancia.juntadeandalucia.es/centros/jaen/mod/url/view.php?id=203776" \o "STREAM)()

.sorted( Comparator  
.comparing(String::length)  
.thenComparing(Comparator.naturalOrder()) )  
.forEach(System.out::println);

El resultado sería así:

Alex  
Alex  
Alex  
John  
John  
Mariam  
Vincent  
Mohammado  
Mohammado

thenComparing funciona para agregar mas filtros y se puede usar n cantidad de veces(comparing().thenComparing().thenComparing()…thenComparing()), dejando prioridad a los filtros anteriores, como en el caso anterior se le dio prioridad al largo del String, y luego se compararon según el orden natural, es importante que se la primera comparación sea solo comparing y luego llamar a thenComparing, sino no compilaría.

‍

‍**Distinct**

Distinct ayuda a evitar datos duplicados y su uso es muy sencillo. Digamos que en el ejemplo anterior queremos ver los nombres sin que estos se repitan:

names.[stream](https://educacionadistancia.juntadeandalucia.es/centros/jaen/mod/url/view.php?id=203776)()  
.distinct()  
.forEach(System.out::println);

El resultado se mostraría así:

John  
Mariam  
Alex  
Mohammado  
Vincent

‍

‍**Limit**

La función Limit ayuda a reducir el tamaño del arreglo (array) al número de registros que necesites:

names.[stream](https://educacionadistancia.juntadeandalucia.es/centros/jaen/mod/url/view.php?id=203776)()  
.limit(5)  
.forEach(System.out::println);

El resultado se mostraría así:

John  
John  
Mariam  
Alex  
Mohammado

‍**forEach**

Esta función ejecuta una sentencia por cada uno de los valores de los arreglos (arrays), pero es necesario comentar que esta función no tiene ningún valor de retorno porque en los ejemplos anteriores siempre se utilizó System.out.println().

‍

‍**count**

Esta función es similar al size() de un arrayList, entrega la cantidad de valores que hay en la lista:

int cantidad = names.[stream](https://educacionadistancia.juntadeandalucia.es/centros/jaen/mod/url/view.php?id=203776)().limit(5).count();

‍

‍**reduce**

Esta función permite tomar todos los valores y almacenarlos en una sola variable, digamos que en el ejemplo de los nombres quisiera dejar un solo String con todos los nombres concatenados; con reduce haría algo así:

String reduce = names.[stream](https://educacionadistancia.juntadeandalucia.es/centros/jaen/mod/url/view.php?id=203776)().reduce("", String::concat);

El resultado sería:

JohnJohnMariamAlexMohammadoMohammadoVincentAlexAlex

Es importante destacar que el primer parámetro del reduce cuenta como el inicio del nuevo String y luego concatenará todos los nombres. También es bastante útil para datos numéricos, como para saber la suma de todos estos.

Integer[] integers = {1, 2, 3, 4, 5};  
Integer reduce = Arrays.[stream](https://educacionadistancia.juntadeandalucia.es/centros/jaen/mod/url/view.php?id=203776)(integers).reduce(0, Integer::sum);

Lo que daría como resultado la cantidad de 15.

‍

‍**collect**

Por último, esta función permite guardar el [stream](https://educacionadistancia.juntadeandalucia.es/centros/jaen/mod/url/view.php?id=203776" \o "STREAM) como un collection o map.

List<String> collect = names.[stream](https://educacionadistancia.juntadeandalucia.es/centros/jaen/mod/url/view.php?id=203776)()  
.filter(s -> s.contains("o"))  
.collect(Collectors.toList());  
  
System.out.println("sin filtro");  
System.out.println(names);  
System.out.println("con filtro");  
System.out.println(collect);

Resultando así:

\* sin filtro  
[John, John, Mariam, Alex, Mohammado, Mohammado, Vincent, Alex, Alex]  
\* con filtro  
[John, John, Mohammado, Mohammado]

En el caso de que quisiéramos contar cuántas veces se repite un nombre, se podría hacer un map de la siguiente manera:

Map<String, Long> counting =  
names.[stream](https://educacionadistancia.juntadeandalucia.es/centros/jaen/mod/url/view.php?id=203776)().collect(Collectors.groupingBy(Function.identity(), Collectors.counting()));  
System.out.println(counting);

Retornando así:

{Alex=3, Vincent=1, Mariam=1, Mohammado=2, John=2}