

# Java Avanzado Programación Funcional

## Introducción

Lambda expresión tiene unos parámetros de entrada y un valor devuelto, por ejemplo:

```
int compare(String o1, String o2)  
  
_____  
(o1, o2) -> o1.length() - o2.length()  
Parámetros           Valor devuelto  
  
f(String, String) -> int
```

```
Comparator<String> comparadorLongitud =  
    (o1, o2) -> o1.length() - o2.length();  
  
Collections.sort(nombres, comparadorLongitud);
```

Las lambdas expressions permite evitar la encapsulación del comparator en una nueva clase.

Este código puedo optimizarlo aún más utilizando la lambda expresión directamente como parámetro del Collections.sort:

```
Collections.sort(nombres, (o1, o2) -> o1.length() - o2.length());
```

También en la versión 8 de Java se han ampliado los tipos del JDK que ya conocemos para sacar ventaja de las lambdas expresión, por ejemplo, podemos utilizar nuevos métodos y referenciar métodos ya existentes de una clase:

```
Collections.sort(nombres, Comparator.comparing(String::length));
```

↗ ↑  
Nuevos métodos en Comparator Method reference

Como resultado veremos la diferencia de cómo se haría la comparación en Java sin lambda expressions y Java 8 con lambda expressions:

```
Comparator<String> comparadorLongitud = new Comparator<String>() {  
  
    @Override  
    public int compare(String o1, String o2) {  
        return o1.length() - o2.length();  
    }  
  
};  
  
Collections.sort(nombres, comparadorLongitud);
```



```
Collections.sort(nombres, Comparator.comparing(String::length));
```

Con Java 8 logramos mejorar nuestro código haciéndolo más compacto, expresivo y reutilizable.

## Interfaces

### Default Methods

Es una de las novedades introducidas en Java 8, recordando la definición de interface es aquella que permite definir qué aspectos tendrán un conjunto de clases de nuestra aplicación sin definir su comportamiento exacto.

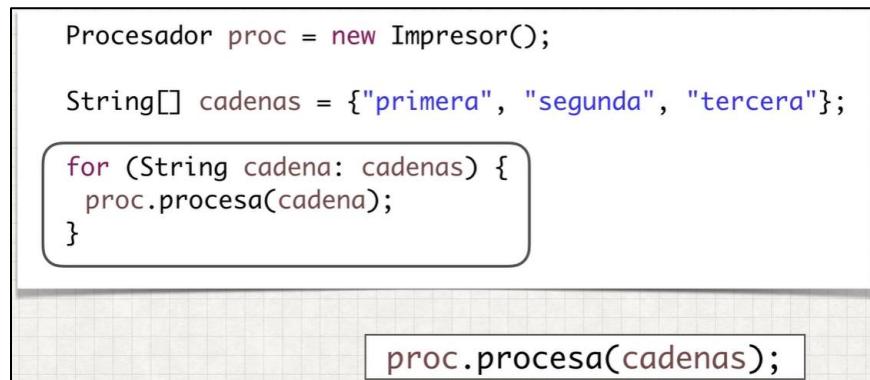
Las mismas son implementadas por distintas clases que dan una implementación específica el método correspondiente.

Por ejemplo:

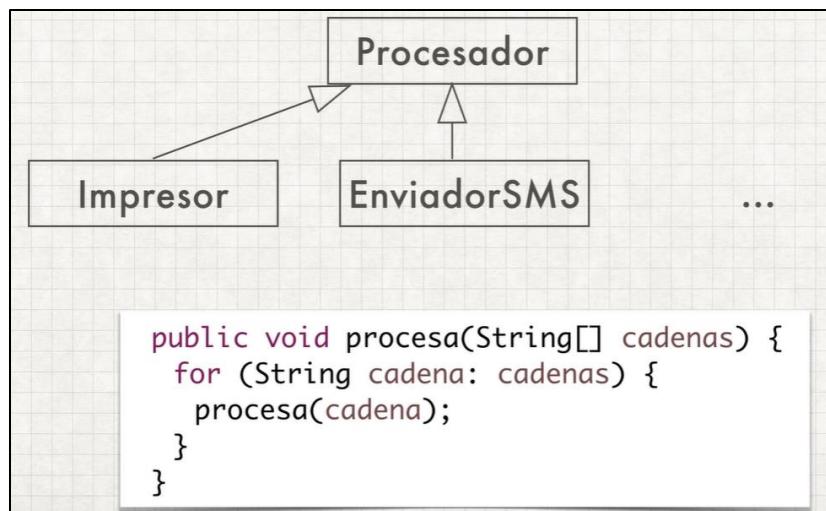
```
public interface Procesador {  
    void procesa (String cadena);  
}  
  
public class Impreso implements Procesador {  
  
    @Override  
    public void procesa(String cadena) {  
        System.out.println("procesando " + cadena);  
    }  
}
```

- Antes de Java 8 las interfaces no podían disponer de ninguna implementación de métodos.
- A partir de java 8 podemos implementar métodos default en las interfaces, lo cual es útil debido al límite de Java de no proveer herencia múltiple y para definir un comportamiento común para varios clientes de una API.

Por ejemplo, tenemos varios clientes que utilizan la interface procesa en un bucle for, por lo que cada vez que esta necesidad aparece, se repite el mismo código, por lo que quisiéramos que el cliente llame a procesa con un array de cadenas y que la lógica se defina dentro de dicho método:



Una forma clásica de hacerlo es añadir el método que queremos en la interface, pero esto nos genera un problema de que el mismo deberá ser implementado en todas las clases que implementen la interface.



Si hay muchas clases que implementan la interface, cada una de ellas deberá implementar el método `procesa(String[] cadenas)` y su lógica siempre será idéntica, recorrer el array y ejecutar el método `procesa` de un solo String, el cuál si es peculiar a cada una de las implementaciones diferentes.

## ¿Por qué debe cada subclase implementar esta versión del método procesa con un array de String?

- Antes de Java 8, esta situación se resolvía con clases abstractas que permiten combinar métodos con implementación, con los llamado métodos abstractos, pero como hemos mencionado utilizar clases abstractas no es tan conveniente como utilizar interfaces.
- En Java 8 se han introducido los **default methods** añadiendo la palabra clave default a un método podemos proporcionar en la misma interface una implementación por defecto a un método:

```
default void procesa(String[] cadenas) {  
    for (String cadena: cadenas) {  
        procesa(cadena);  
    }  
}
```

Este método puede ser sobrescrito por cualquier clase que implementa la interface si así se considera necesario.

Por otra parte, tampoco hay ninguna limitación en usar desde los default methods métodos que no están implementados en la interface.

Gracias a los default methods las clases que implementen dicha interface solo deberán implementar los métodos abstractos a la misma, es decir los que son particulares para cada implementación, no debe implementar los métodos para los cuales la interface ya tiene toda la información para complementarlos.

De esta forma el cliente puede usar indistintamente los métodos procesa con un String o con un String[].

## Características

- Una interface puede tener cualquier número de métodos default.
- Las clases que implementa la interface puede sobre-escribir los métodos default.
- Si una clase implementa dos interfaces y recibe dos métodos default con la misma forma, es obligatorio sobrescribir el método.

## Static Methods

Otro de los cambios introducidos en las interfaces en Java 8 es permitir la definición de métodos estáticos, los cuáles eran exclusivos de las clases.

Este cambio permite introducir implementación en la definición de una interface, como en las clases los métodos estáticos no pueden referenciar a ningún método o variable no estáticos, sirven para proporcionar funcionalidad que no depende sobre que instancia se ejecuta el método.

Un ejemplo podría ser una interface que sirve para obtener traducciones de mensajes a partir de una clave y del Local al queremos traducirlo.

Para hacer más prácticas la interface podemos añadir un método que obtuviese el Locale por defecto del entorno sobre el que se ejecuta la aplicación:

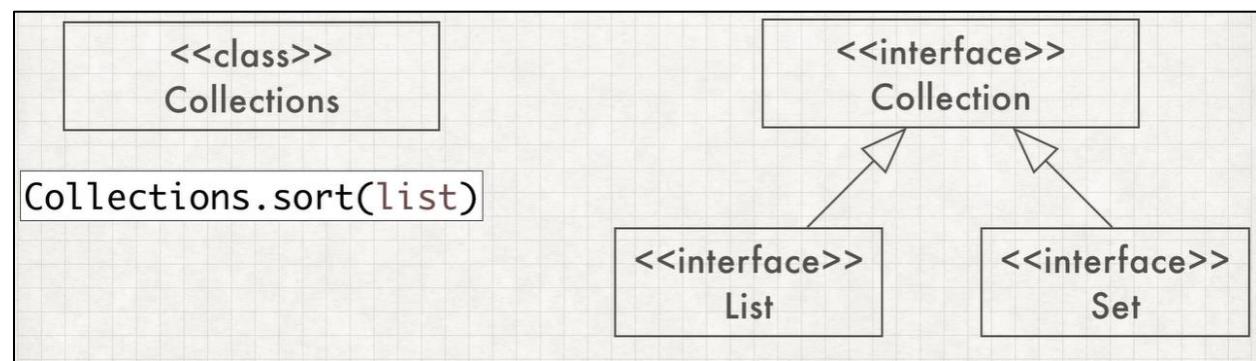
### Visibilidad public

```
interface Traductor {  
    static Locale obtenLocaleEntorno() {  
        return Locale.getDefault();  
    }  
  
    String traducion (Locale locale, String clave);  
}
```

En versiones anteriores a Java 8 antes de poder incluir métodos static en las interfaces se debía resolver de una manera no muy elegante.

Por ejemplo, el diseño del framework Collection, el cual está liderado por la interfaz Collection que describe que métodos serán comunes a todas la estructuras de datos que la implementen. A partir de esta se van definiendo subinterfaces más específicas y después numerosas implementaciones con distintas características.

Este framework además dispone de distintos métodos de utilidades generales no específicos a ninguna instancia, al no poder incluir estos métodos en la interface Collection, se crea una clase Collections y se ubican allí estos métodos. Collections dispone de un amplio catálogo de métodos estáticos como por ejemplo el método sort que sirve para ordenar una lista de elementos.



**Estos tipos de construcciones con los métodos estáticos en las interfaces ya no son necesarios.**

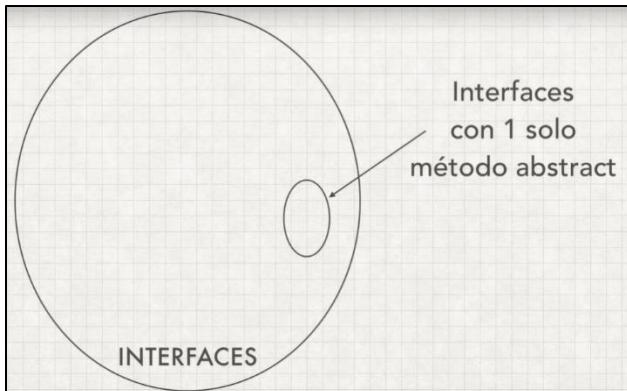
## @FunctionalInterface

Hasta ahora hemos visto que a partir de java 8 podemos tener los nuevos métodos static, los métodos default y los métodos abstractos, es decir los que necesitan implementación:

```
interface Traductor {  
  
    static Locale obtenLocaleEntorno() {  
        return Locale.getDefault();  
    }  
  
    String traduccion (Locale locale, String clave);  
  
    default List<String> traduccion(Locale locale, List<String> claves) {  
        List<String> traducciones = new ArrayList<>();  
        for (String clave: claves) {  
            traducciones.add (traduccion(locale, clave));  
        }  
        return traducciones;  
    }  
}
```

Una interface puede tener cualquier número de métodos static, abstract y default.

Para el tema que estamos por ver nos interesan las interfaces que poseen solo 1 método abstract, y cualquier cantidad de static y default.



Estas interfaces ya tenían una presencia muy importante en las versiones anteriores de Java, tradicionalmente conocidas como **Single Abstract Method (SAM)**. Algunas de ellas:

- Callable y Runnable para trabajar con Threads.
- ActionListener: para trabajar con interfaces gráficas.
- Comparator

A partir de Java 8 utilizaremos otro termino para referirnos a estas interfaces: **Functional Interface**, por lo tanto, una interface es una Functional Interface cuando el número de métodos abstractos que posee es exactamente uno.

En vista del rol importante que jugaran las functional interface en las lambda se ha añadido una notación explícitamente para ellas: **@FunctionalInterface**.

Esta notación es reconocida por el compilador y actúa de validador para este tipo de interfaces.

Ejemplos:

```
@FunctionalInterface  
interface Sumador {  
    int suma (int a, int b, int c);  
    int suma (int a, int b);  
}
```

ERROR

```
@FunctionalInterface  
interface Sumador {  
    int suma (int a, int b);  
}
```

OK

```
@FunctionalInterface  
interface Sumador {}
```

ERROR

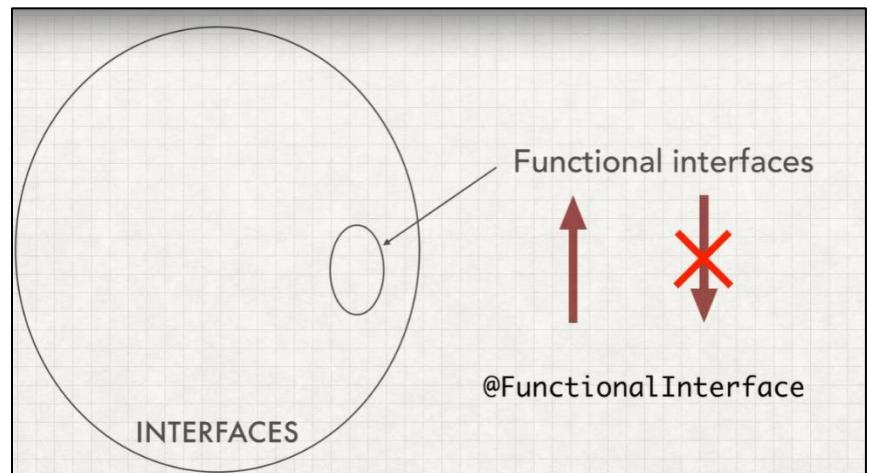
```
@FunctionalInterface  
interface Sumador {  
  
    default int suma (int a, int b, int c) {  
        return suma(a, suma(b,c));  
    }  
    int suma (int a, int b);  
}
```

OK

Es importante no confundir los conceptos **Functional Interface** con la annotation **@FunctionalInterface**.

Las funcional interface son el subconjunto de interfaces con un método abstract, mientras que las interfaces **@FunctionalInterface** son las interfaces con esta anotación en virtud de la cual el compilador comprueba que sean interfaces funcionales.

Todos las **@FunctionalInterface** son Functional Interfaces, pero la inversa no es necesariamente cierta. Las **@FunctionalInterface** forman un subconjunto de las Functional Interfaces.



# Lambda

## Tipos de Lambda

En esta sección veremos cómo ve Java las expresiones lambda, es decir que papel ocupan las expresiones lambdas en el sistema de tipos de Java.

La sintaxis de la parte derecha de la asignación representa la principal novedad de Java 8, es una expresión que representa una determinada operación sobre unos parámetros y que produce un resultado.

Ahora la cuestión a resolver es **¿Cómo han encajado las lambdas en Java?**

La siguiente expresión representa una operación, lo más parecido que tenemos en Java a una operación es un método, pero no podemos tener una expresión en Java que nos devuelva un método.

```
Comparator<String> comparadorLongitud =  
    (o1, o2) -> o1.length() - o2.length();
```

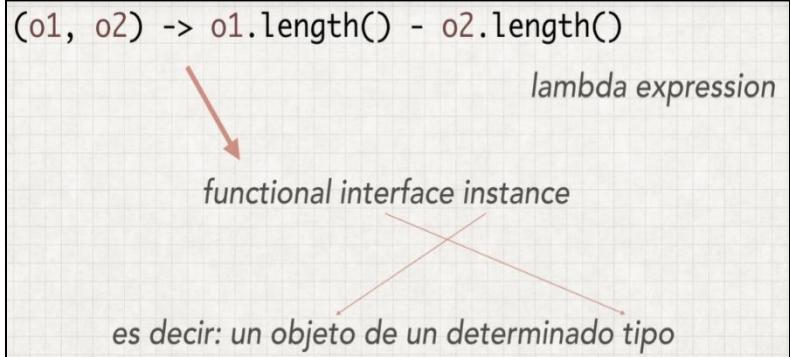
(o1, o2) -> o1.length() - o2.length() ?

En Java tenemos valores primitivos y referencias a objetos. Los métodos forman parte de un tipo. Ahora nuestra duda es a resolver es **¿cómo encajan los lambdas en este mapa?**

La expresión anterior recibe el nombre de **lambda expression**, el término lambda proviene de la familia de los lenguajes funcionales y estos a su vez lo toman de una rama de las matemáticas. Una expresión es un elemento de un programa que una vez evaluado produce un resultado, así que la lambda expression anterior produce un resultado, el cuál debe ser asumible por Java.

Si vamos a la documentación de Java veremos **que el resultado de una lambda expression es una functional interface instance**, es decir un objeto de un determinado tipo de interfaz funcional.

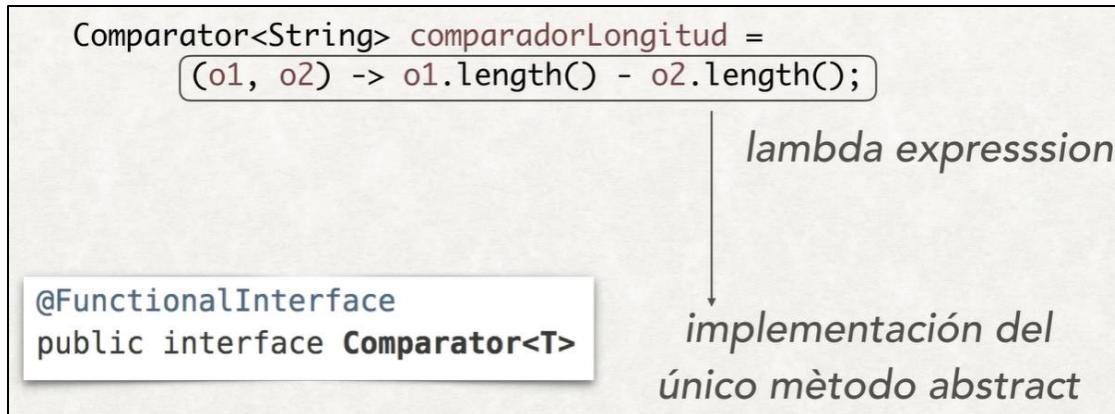
**Aunque una expression lambda tenga la apariencia de método o de operación su resultado es una instancia de un determinado tipo, encaja por lo tanto en el sistema de tipos de Java.**



Una vez entendido este concepto debemos explicar como se corresponde la expresión lambda con la definición de la interfaz funcional.

Hemos visto que la expresión lambda especifica una operación, esta operación es la implementación del único método abstract que por definición tiene una functional interface.

Ejemplo:



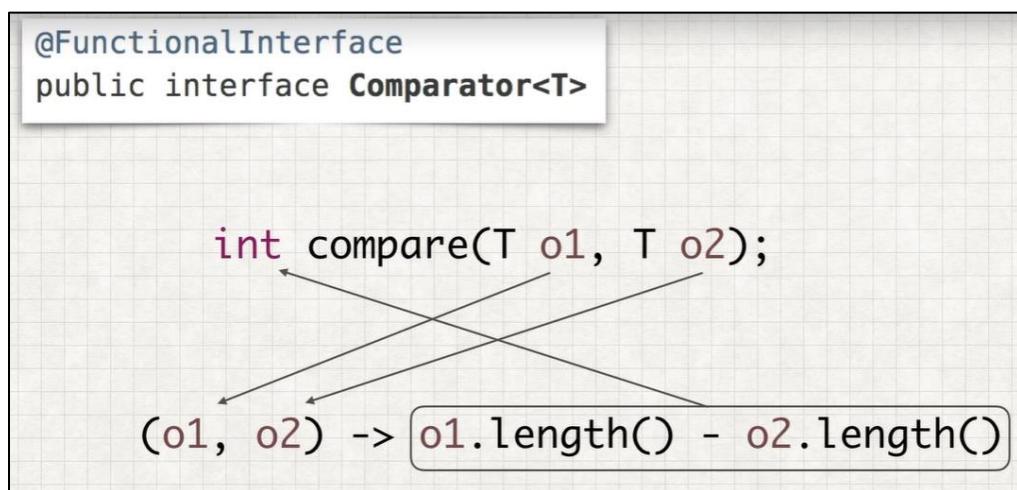
En este caso Comparator es efectivamente una functional interface existente desde la versión 1.2 de Java y modificada para incluir en su definición la annotations de @FunctionalInterface en las versiones de Java 8 en adelante.

Para ver que la lambda expression sirve como implementación del método abstract de Comparator vayamos a su definición:

La interface Comparator, incluye el método abstracto compare, cuya firma indica que se requieren dos parámetros del tipo genérico T y se produce un resultado entero.

La lambda expresión tiene precisamente esta estructura requerida:

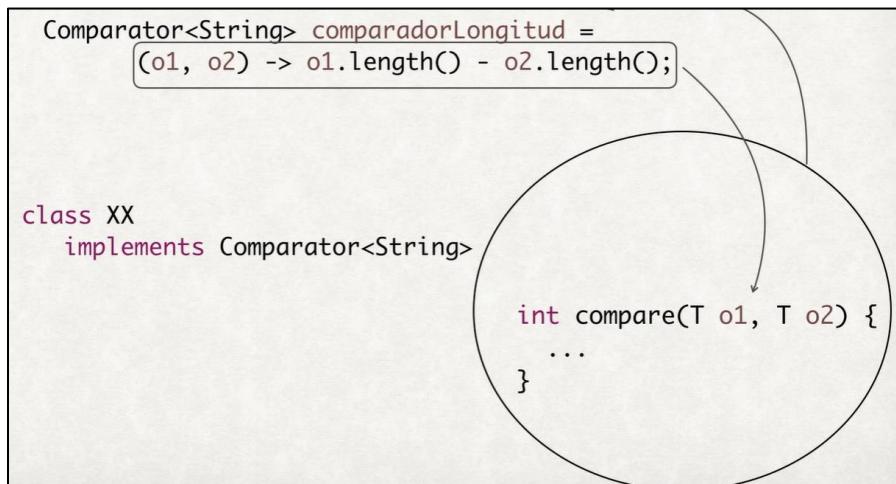
- La parte izquierda de la expresión indica los parámetros que recibe lambda.
- La parte derecha produce un resultado del tipo entero requerido.



Conceptualmente el proceso de valuar la sentencia que contiene la lambda expresión sería similar a construir un nuevo tipo que implementa la interfaz funcional requerida.

En el caso del ejemplo un Comparador de String, utilizar la lambda expression como implementación del método abstract de la interfaz, es decir del método compare, siendo T un String.

Y finalmente construir un objeto del este tipo recién creado como resultado de la expresión y asignarlo a la variable, en esta caso comparadorLongitud.

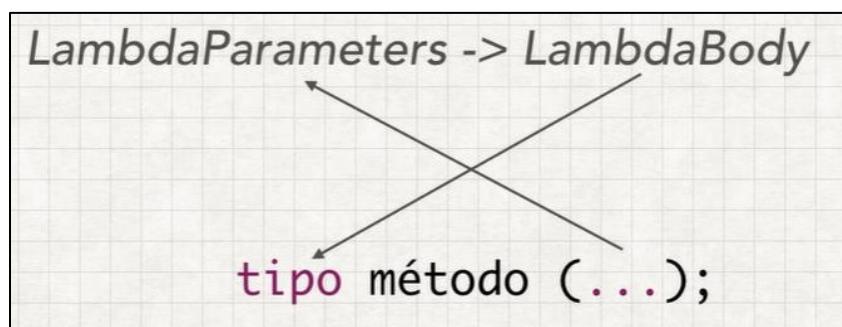


## Lambda Expressions

La definición general de la estructura de las Lambda Expression es:

**LambdaParameters -> LambdaBody**

Esta expresión debe corresponderse con el método abstracto que implementa:



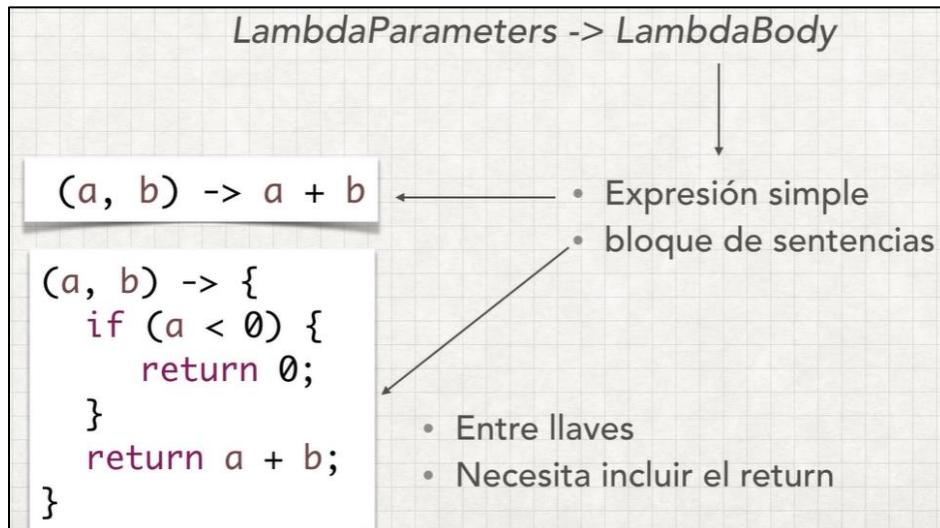
Desglosemos un poco como se componen estas expresiones lambdas:

**LambdaParameters:** lista de parámetros separados por comas (",") y entre paréntesis

- Los paréntesis son opcionales si solo hay un parámetro, en cualquier otro caso 0 o + de 1 parámetro los paréntesis son obligatorios.
- A diferencia de la declaración de un método, no es necesario especificar el tipo de los parámetros, el compilador va a inferir su tipo a partir del destino de la lambda expression.

**LambdaBody:** admite 2 formas diferentes:

- Expresión simple
- Bloque de sentencias:
  - Entre llaves.
  - Necesitan incluir el return.



### Ejemplo de Lambda expressions

Método abstract	lambda expression
int m (int a, int b)	(a, b) -> a (int a, int b) -> a
int m (int a)	(a) -> a a -> a a -> { return a; }
int m ()	() -> 5
void m (int b)	(a) -> {}

Analizando un poco los ejemplos podemos observar lo siguiente:

1. Tenemos dos formas de expresar la expresión lambda ya que los tipos de datos en los parámetros de Lambda son opcionales.
2. Si tenemos un solo parámetro el uso de los paréntesis es opcional, también se puede usar un bloque de expresiones en el cuerpo del Lambda, en dicho caso debemos incluir el return. Las 3 expresiones son equivalentes.
3. Representa una expresión lambda sin parámetros y que devuelve siempre el valor 5.
4. La expresión lambda no devuelve ningún valor, en este caso no podemos utilizar como el cuerpo del lambda una expresión simple, ya que esta siempre devuelve un valor.

## Functional interfaces

### java.util.function

En la versión 8 del JDK se han añadido un conjunto de interfaces funcionales de uso general, que se han añadido en el package java.util.functions:

Interface Summary	
Interface	Description
<code>BiConsumer&lt;T,U&gt;</code>	Represents an operation that accepts two arguments and performs a side-effect on the first argument.
<code>BiFunction&lt;T,U,R&gt;</code>	Represents a function that accepts two arguments and produces a result.
<code>BinaryOperator&lt;T&gt;</code>	Represents an operation upon two operands.
<code>BiPredicate&lt;T,U&gt;</code>	Represents a predicate (boolean-valued function) of two arguments.
<code>BooleanSupplier</code>	Represents a supplier of boolean-valued data.
<code>Consumer&lt;T&gt;</code>	Represents an operation that accepts a single input argument and performs a side-effect on it.
<code>DoubleBinaryOperator</code>	Represents an operation upon two double arguments.
<code>DoubleConsumer</code>	Represents an operation that accepts a single double argument and performs a side-effect on it.
<code>DoubleFunction&lt;R&gt;</code>	Represents a function that accepts a double argument and produces a result.
<code>DoublePredicate</code>	Represents a predicate (boolean-valued function) of a double argument.

## Interface Function<T,R>

Una función muy general es aquella que acepta un tipo de un objeto y devuelve otro, es decir una función de un conjunto a otro. Este tipo está representado por una interfaz funcional **Función<T,R>**, con los tipos genéricos T representando el dominio de la función, los valores que acepta y R el recorrido, es decir el tipo de resultado que produce.

El método abstracto para implementar es apply.

### Interface Function<T,R>

$$f(x) \rightarrow y$$

R apply(T t);

Un ejemplo sería devolver el tipo de la persona que se pasa como parámetro:

```
Function<Persona, String> nombre = per -> per.getNombre();
```

Para las funciones también existen variantes cuando o bien el tipo de dominio T o el tipo de recorrido R o los dos son primitivos.

- En caso de que el dominio sea primitivo el nombre de la función es el nombre del tipo primitivo seguido de la palabra Function, por ejemplo: **IntFunction<R>**
- En caso de que el recorrido sea primitivo el nombre de la función es el texto To seguido del tipo primitivo y Function, por ejemplo: **ToLongFunction<T>**
- En caso de que tanto dominio como recorrido sean primitivos, el nombre es el tipo de dominio seguido de To, el tipo del recorrido y la palabra Function, por ejemplo: **DoubleToIntFunction**.

## Composiciones con Function

Funciones sencillas se pueden combinar para construir funciones más complejas.

Function ofrece 2 posibilidades para componer funciones:

- El método **andThen** que crea una función añadiendo a una existente otra que debe realizarse posteriormente, por ejemplo:

*andThen : añadir funcionalidad posterior*

```
Function<Persona, String> nombre = per -> per.getNombre();
nombre = nombre.andThen(it -> it.toUpperCase());
```

Nombre en primer lugar es una función que devuelve el nombre de una persona, pero en la segunda asignación es una función que sobre un nombre devuelve el mismo en UpperCase

- **Compose:** añade funcionalidad anterior, se ejecuta primero la funcionalidad expresada en el parámetro y después la de la propia función, por ejemplo:

*compose : añadir funcionalidad anterior*

```
Function<Coche, Persona> propietario = it -> it.getPropietario();
Function<Persona, String> nombre = per -> per.getNombre();
```

```
Function<Coche, String> nombrePropietario = nombre.compose(propietario);
```

Si propietario obtiene el propietario de un coche y nombre el nombre de una persona, nombre.compose(propietario) obtiene el nombre del propietario de un coche, primero se ejecuta propietario y después nombre.

## Interface UnaryOperator<T,R>

Un operador es un caso particular de Function donde el tipo del dominio es igual al tipo del recorrido, para estos casos existe el tipo **UnaryOperator<T>**.

$$f:T \rightarrow T$$

Cómo solo podemos usar como genéricos tipos no primitivos hay una serie de tipos con forma de operador que trabajan con tipos primitivos:

- **IntUnaryOperator**
- **LongUnaryOperator**
- **DoubleUnaryOperator**

## Interface Consumer<T>

Otra variación es cuando no existe o bien dominio o recorrido, un **Consumer<T>** representa una operación que representa un parámetro, pero no produce ningún resultado.

Un ejemplo sería una función que imprime por consola el valor pasado como parámetro:

```
Consumer<String> impresor = (it) -> {System.out.println(it);}
```

## Interface Supplier<T>

Un supplier inverso al consumer es una operación que sin ningún parámetro produce un valor, por ejemplo, un generador de números aleatorios:

```
Random random = new Random();
Supplier<Integer> generador = () -> random.nextInt();
```

## Interface Predicate<T>

Un predicado es un tipo específico de Function muy frecuente en nuestros programas.

Se puede utilizar para determinar si el resultado de evaluar un objeto es verdadero o falso.

$$f(x) \rightarrow \text{boolean}$$

```
Predicate<String> cadenaCorta = it -> it.length() < 10;
```

En el ejemplo se usa para identificar que las cadenas cortas son cadenas de menos de 10 caracteres.

Hay variantes para el caso de predicados para tipos primitivos:

- **IntPredicate**
- **LongPredicate**
- **DoublePredicate**

## Binary \*

Se utiliza el prefijo Bi o Binary cuando se duplica el número de parámetros, aplicado a los tipos que ya hemos visto produce una multitud de interfaces funcionales nuevas, como por ejemplo las siguientes:

**BinaryOperator<T>** **f(t,t) -> t** : se aplica sobre 2 parametros y produce un resultado del mismo tipo que los parámetros.

**BiFunction<T,U,R>** **f(t,u) -> r** : mismo funcionamiento que interface Function<T,R> pero ahora con dos parámetros.

**BiPredicate<T,U>** **f(t,u) -> boolean** : variante para Predicate, pero en este caso de 2 parametros.

**DoubleBinaryOperator** **f(double,double) -> double** : función de 2 doubles transformación a otro double.

## Method references

Para explicar esto partamos de una lista de String que queremos ordenar según su longitud:

```
List<String> lista = Arrays.asList("ab", "b", "ccc");  
lista.sort((o1, o2) -> o1.length() - o2.length());
```

Para esto usamos una lambda expression que produzca la implementación de comparador que requiere el método sort de lista. Pero ¿qué sucedería si ya existe un método que realiza esta comparación en otra clase?

Por ejemplo:

```
class Utilidades {  
    public static int compare(String o1, String o2) {  
        return o1.length() - o2.length();  
    }  
}
```

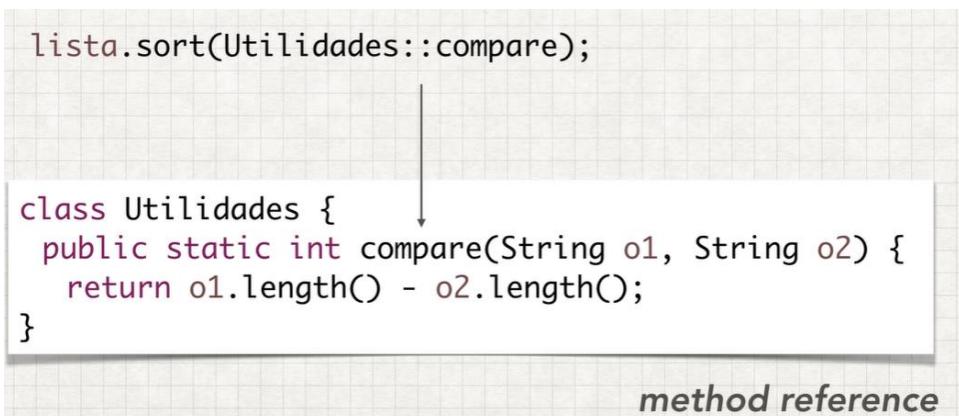
Podríamos reusar este método llamando el mismo en el cuerpo de la expresión lambda invocar el método static de utilidades:

```
lista.sort((o1,o2) -> Utilidades.compare(o1, o2));
```

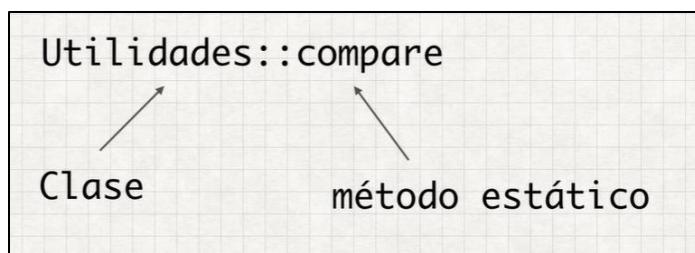
Este código es mejor que el anterior pero todavía hay una duplicidad que se puede evitar:

```
lista.sort(Utilidades::compare);
```

En este caso no estamos pasando al método sort una lambda expression que invoca a su vez al método compare de utilidades. Estamos pasando directamente una referencia al método compare, lo que evita la redundancia, estamos reusando completamente el método que ya existe. Este recurso recibe el **method reference**, a partir de Java 8 podemos usar referencias a métodos, al igual que lambda expression para proporcionar implementaciones de Functional Interfaces.

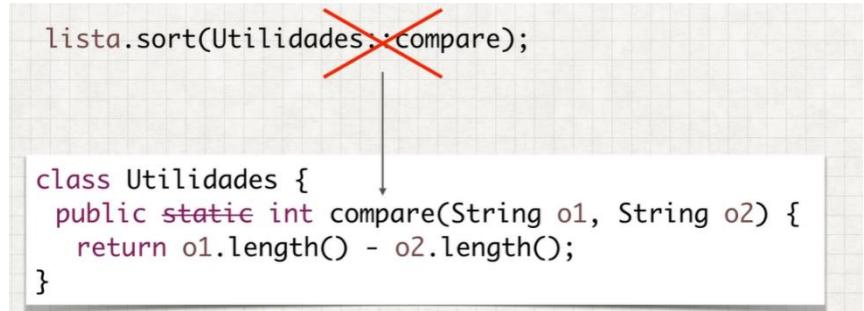


La sintaxis de un method reference, es relativamente sencilla, en primer lugar se escribe el nombre de la clase que invocamos, después se escriben dos veces dos puntos “::” y a continuación el método al que referenciamos (no se escriben paréntesis al final).



El método que hemos referenciado en utilidades es estático ¿qué sucedería si no lo fuera?

Si quitamos el método static de la sintaxis anterior el method reference no funcionará, la sintaxis que hemos visto **solamente funciona como métodos estáticos**.



Para invocar un método no estático necesitamos hacerlo a partir de una referencia a un objeto determinado:

```
Utilidades util = new Utilidades();
lista.sort(util::compare);

class Utilidades {
    public int compare(String o1, String o2) {
        return o1.length() - o2.length();
}
```

Llamamos a esta referencia, referencia a un método de una instancia particular.

#### Referencia a un método de una instancia arbitraria

Como antes he de proporcionar una implementación del método compare que recibe dos Strings y devuelve un entero:

```
int compare(String o1, String o2);
```

En este caso quiero usar el método de String compareTolgnoreCase, para realizar la ordenación, este es un método no estático que compara el String sobre el que se invoca el método con el String que recibe como parámetro:

```
public final class String ... {
    ...
    public int compareToIgnoreCase(String str)
```

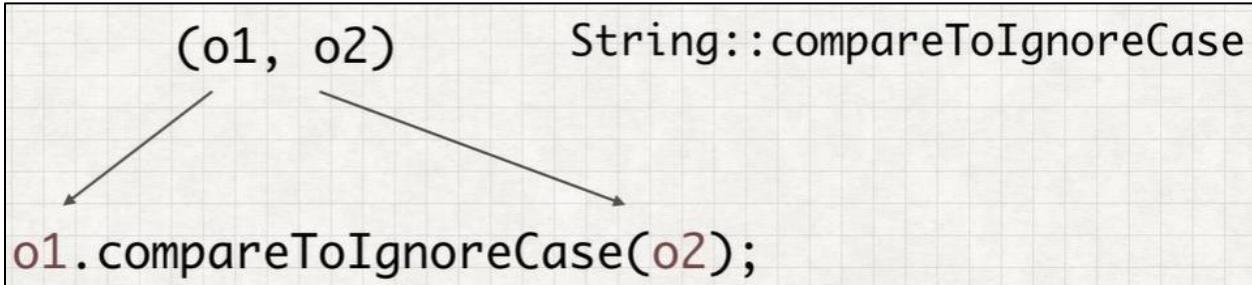
La forma de resolver esto es pasando el nombre de la clase, los cuatro puntos y el nombre del método no estático:

```
lista.sort(String::compareToIgnoreCase);
```

Pasamos a explicarlo un poco más a detalle, la firma del método a implementar tiene dos parámetros y devuelve un valor entero:

```
int compare(String o1, String o2);
```

La referencia al método no estático es el nombre de la clase cuatro puntos y el nombre del método no estático, entonces cuando se recibe la invocación con el método de compare lo que se ejecuta es el método referenciado (compareToIgnoreCase) sobre el primer parámetro y pasando como parámetro el resto de los argumentos:



### Referencia a un constructor

Si necesitamos implementar un método que dado un String nos devuelva el Integer creado a partir de este String, podemos usar una referencia al constructor de Integer usando después de los cuatro puntos, la palabra new. El funcionamiento en este caso cuando se invoca el método es usar el constructor referenciado y pasar como parámetro el argumento usado:

```
Function<String, Integer> conversor = Integer::new;
```

```
conversor.apply("3")
```

```
new Integer("3")
```

## Optional

Su uso viene a combatir el uso de los null y los problemas que conlleva el mismo.

¿Qué problema hay con null?

Observemos lo siguiente:

```
String saluda (Persona persona) {  
    return "Hola " + persona.getNombre();  
}
```

Tendremos un método que a partir de una persona que se pasa por parámetro devuelve un String con la forma de saludar a esta persona, ahora ¿qué sucede si se pasa como parámetro al método persona la referencia null?

Nos daría un error de ejecución muy conocido como es el NullPointerException:

```
saluda(null);  
  
↓  
  
Exception in thread "main" java.lang.NullPointerException  
at Main.saluda(Main.java:28)  
at Main.main(Main.java:23)
```

Nuestro método es frágil la solución es comprobar antes de ejecutar persona.getNombre() que la persona no sea nula:

```
String saluda (Persona persona) {  
    if (persona == null) {  
        return "Estoy solo";  
    }  
    return "Hola " + persona.getNombre();  
}
```

Pero esta forma de resolverlo no es conveniente, ya que oculta la parte útil de nuestro método haciendo que 3 de las 4 líneas sean solo para defendernos de la referencia nula.

Viendo la firma del método ¿cómo puedo saber si es obligatorio pasar una referencia a un objeto Persona existente o si admite el caso de una referencia nula?

**Optional** nos permite hacer explícito cuando una referencia debe ser siempre para un objeto existente o cuando puede no referenciar a ningún valor. Optional tiene un parámetro tipo para el tipo que contiene:

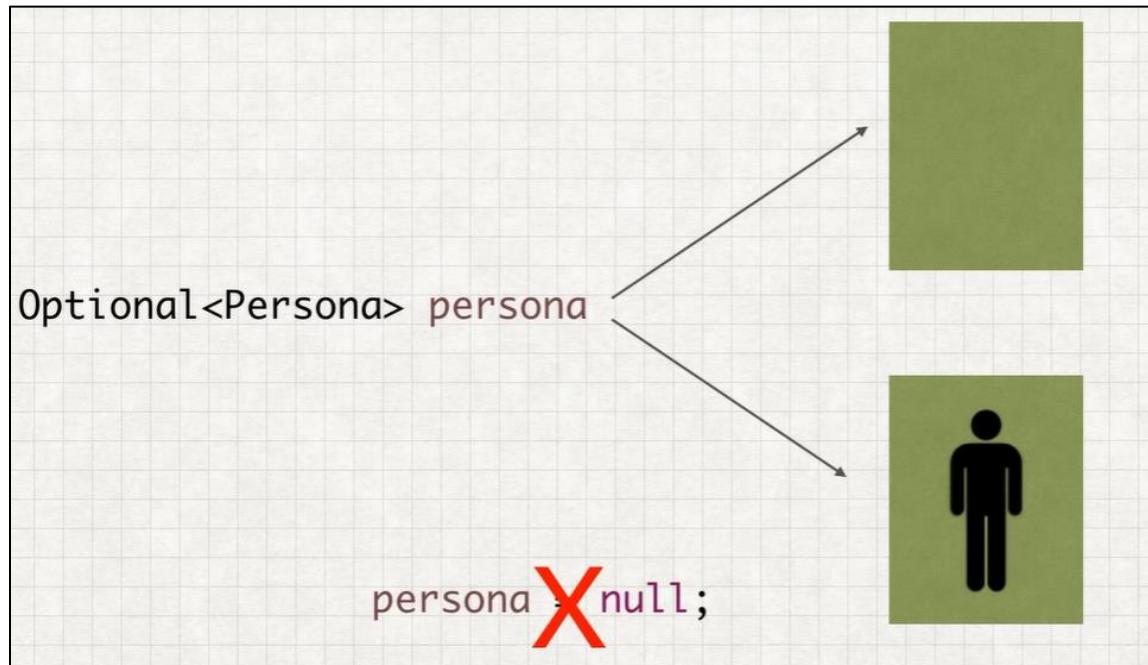
```
String saluda ( Optional<Persona> persona ) {  
    ...  
}
```

En nuestro caso si el método saludo debe devolver un saludo para el caso donde no haya persona el tipo de parámetro será **Optional<Persona>**. Podemos ver por lo tanto una referencia optional como una referencia a un contenedor que a su vez puede contener o no un determinado valor.

Entonces Persona declarado como Optional de Persona puede tener 2 valores:

1. Una referencia a un contenedor vacío.
2. Una referencia a un contenedor que a su vez contiene una referencia a un objeto Persona.

Esto evita que la referencia a Persona no sea nula nunca.



Con **Optional** desaparece la necesidad de usar la referencia null.

## Creación

Para crear una referencia a un optional hay distintas opciones:

1. Si el Optional va a tener valor podemos usar el método estático **of**, le pasamos la referencia que debe contenedor que no debe ser null:

```
Optional<Persona> persona = Optional.of(juan);
```

2. Si el Optional no va a tener valor podemos usar el método estático **empty**:

```
Optional<Persona> persona = Optional.empty();
```

3. En caso de querer crear el Optional de una referencia que puede ser null podemos usar el método estático **ofNullable**. Este uso no es el más frecuente, aparece principalmente cuando estamos tratando con código anterior a Java 8:

```
Optional<Persona> persona = Optional.ofNullable(juan);
```

Una vez introducido el tipo Optional ya tenemos explícito en la firma del método que este está preparado para devolver un saludo, aunque no haya referencia a un objeto Persona.

Implementaríamos el método usando el método **isPresent** que nos indica si el Optional contiene o no un valor y el método **get** para acceder al valor que por el if anterior ahora sabemos que existe:

```
String saluda (Optional<Persona> persona) {  
  
    if (persona.isPresent()) {  
        return "Hola " + persona.get().getNombre();  
    } else {  
        return "Estoy solo";  
    }  
}
```

Aunque el código funcione no es la forma más habitual de trabajar con optional ya que es poco “idiomático”.

## map

El método **map** sirve para ejecutar una función sobre el valor de un Optional si está presente y devuelve un Optional vacío si no lo está. Nos sirve por ejemplo para implementar el método saluda en el caso de que quiera devolver un saludo sin valor en caso de que la persona pasada sea un Optional vacío. Si Persona es un Optional con valor, el valor devuelto será un Optional<String> con valor y si el Optional<Persona> pasado como parámetro no tiene valor, el método me va a devolver un Optional<String> también sin valor:

```
Optional<String> saluda (Optional<Persona> persona) {  
    return persona.map(it -> "Hola " + it.getNombre());  
}
```

## orElse

Si quiero devolver un valor en el caso de que el Optional no tenga contenido disponemos del método **orElse**, el cual devuelve el valor contenido en el Optional si este existe o el parámetro que acepta este método en caso contrario. De esta forma podría forzar al método que tengo a devolver un valor String si combino el map con el orElse:

```
String saluda (Optional<Persona> persona) {  
    return persona.map(it -> "Hola " + it.getNombre())  
        .orElse("Estoy solo");  
}
```

De estas dos formas la referencia a **null** a desaparecido de nuestro código.

## map y orElse

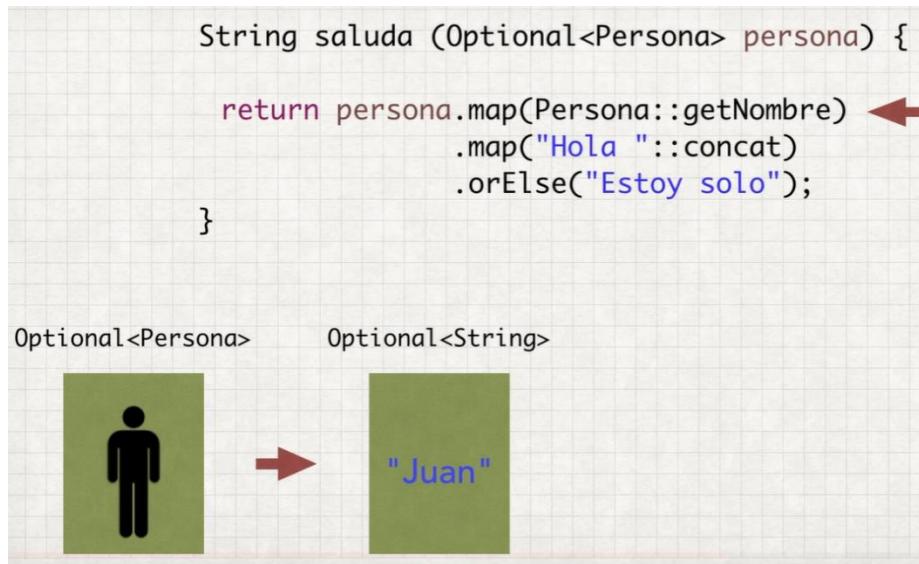
Otro ejemplo que combina el uso de **map** con **orElse** es poder reescribir el código anterior con otra versión:

```
String saluda (Optional<Persona> persona) {  
  
    return persona.map(Persona::getNombre)  
        .map("Hola " :: concat)  
        .orElse("Estoy solo");  
}
```

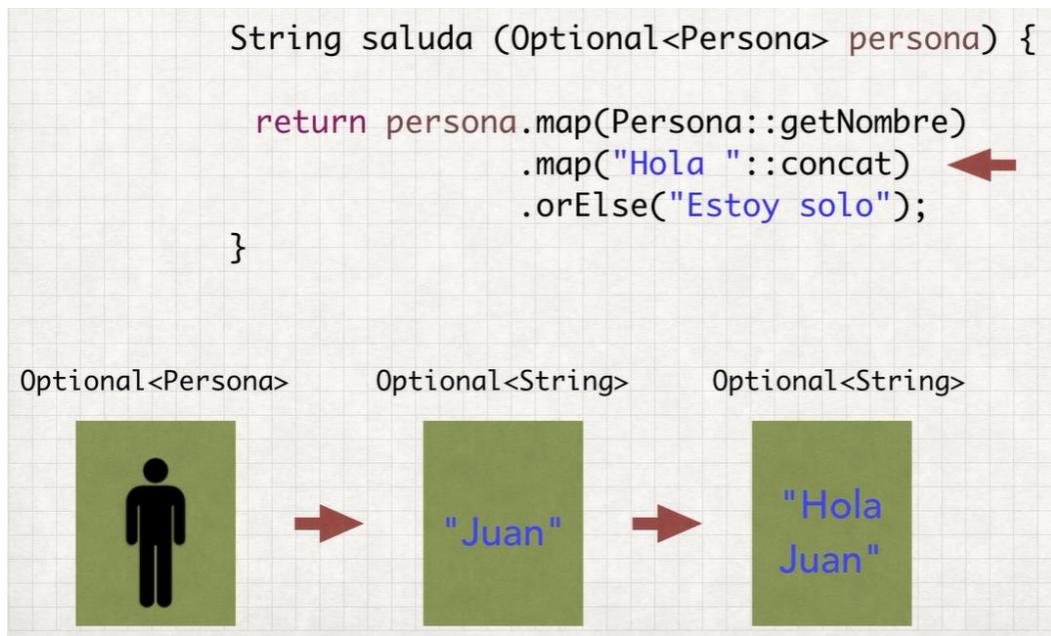
Veamos la ejecución paso a paso de este código:

4. Primero veamos para el caso donde Persona tiene contenido.

1. El **map** pasando como de una referencia al método `getNombre` devuelve un `Optional` de `String`. En nuestro caso al tener la variable `Persona` valor con el nombre de la Persona del contenido.

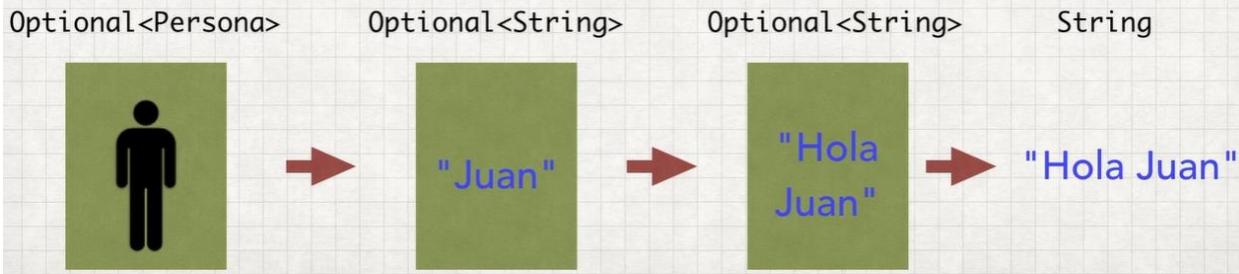


2. El siguiente **map** concatena la palabra “Hola” al contenido en caso de existir que es nuestro caso:



3. Finalmente, **orElse** devuelve el valor del contenido ya que este existe:

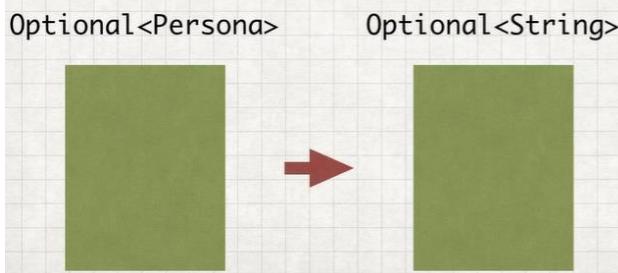
```
String saluda (Optional<Persona> persona) {  
  
    return persona.map(Persona::getNombre)  
        .map("Hola " :: concat)  
        .orElse("Estoy solo"); ←  
}
```



5. La ejecución en caso de pasarnos una variable de Persona sin contenido sería:

1. El primer **map** al no tener Persona contenido, nos devolvería un **Optional<String>** vacío:

```
String saluda (Optional<Persona> persona) {  
  
    return persona.map(Persona::getNombre) ←  
        .map("Hola " :: concat)  
        .orElse("Estoy solo");  
}
```



2. Lo mismo sucedería con el segundo **map** con la referencia al método concat:

```
String saluda (Optional<Persona> persona) {  
  
    return persona.map(Persona::getNombre)  
        .map("Hola " ::concat) ←  
        .orElse("Estoy solo");  
}
```

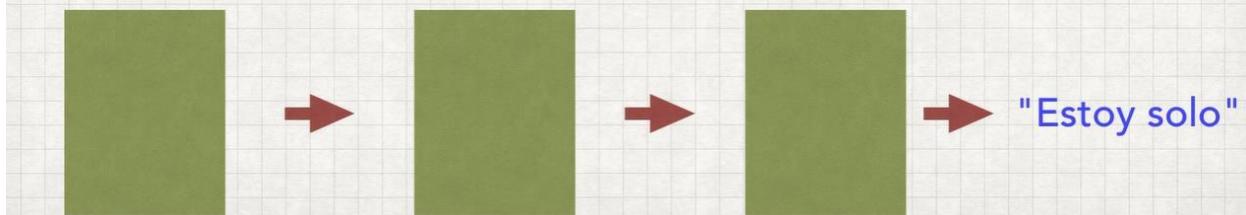
Optional<Persona>      Optional<String>      Optional<String>



3. Al llegar al **orElse** el método devolverá el parámetro especificado ya que el Optional sobre el que se ejecuta no tiene contenido:

```
String saluda (Optional<Persona> persona) {  
  
    return persona.map(Persona::getNombre)  
        .map("Hola " ::concat)  
        .orElse("Estoy solo"); ←  
}
```

Optional<Persona>      Optional<String>      Optional<String>      String



¿Qué sucede cuando a un map le pasamos una función que a su vez puede devolver un Optional?

Modifiquemos la definición de la clase Persona para que ahora el nombre sea Optional.

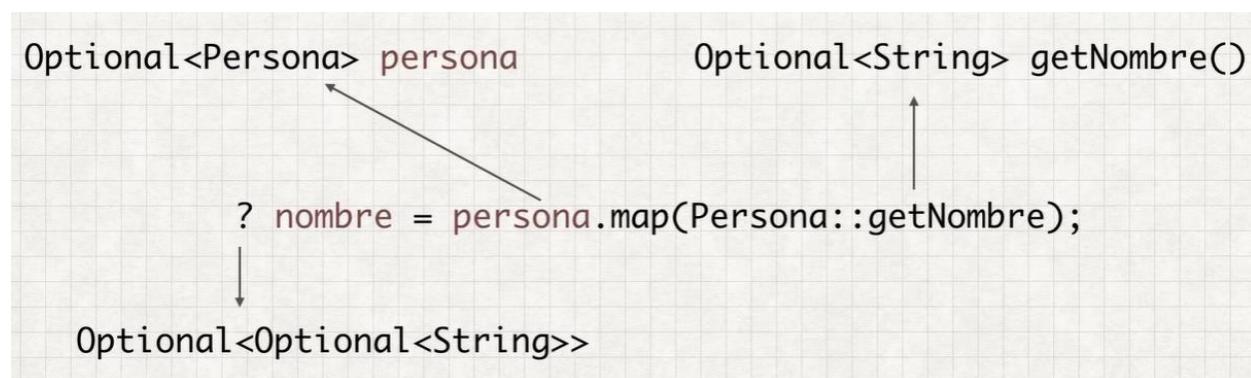
```
static class Persona {  
    private Optional<String> nombre;  
  
    Optional<String> getNombre() {  
        return nombre;  
    }  
}
```

¿Qué pasaría si ahora sobre Persona ejecutamos el método map con una función que devuelve a su vez un Optional?

```
String saluda(Optional<Persona> persona) {  
  
    ? nombre = persona.map(Persona::getNombre);  
    ...  
}
```

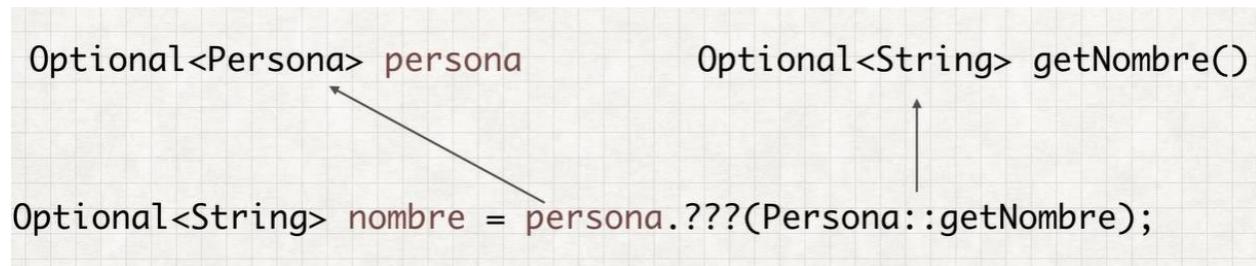
Dicho de otra forma **¿cuál es el tipo de la variable nombre que recibe este resultado?**

Como ya hemos visto persona es un Optional<Persona> y getNombre es una función de Persona a Optional<String>, por este motivo el tipo devuelto y por lo tanto el tipo de nombre es **Optional<Optional<String>>**:



## flatMap

En este caso lo que queremos es un método que devuelva un Optional<String>:



Que el Optional sea vacío, si bien Persona o el nombre de Persona son ausentes y el nombre de esta Persona si los dos están presentes.

El método que ofrece este comportamiento es **flatMap**:

```
public<U> Optional<U> flatMap(Function<? super T, Optional<U>> mapper)
```

Este método sirve cuando, como nuestro caso, sobre un Optional pasamos una función que devuelve a su vez un Optional. Lo que hace es aplanar, aplastar las dos posibles fuentes de ausencia en el resultado, o bien la referencia original es ausente o el mapper devuelve un valor ausente, en un solo Optional, por lo tanto, usando **flatMap** obtenemos el tipo que queríamos:

```
Optional<String> nombre = persona.flatMap(Person::getNombre);
```

Veamos su uso en un caso un poco más complicado:

```
static class Persona {  
    private Optional<String> nombre;  
  
    Optional<String> getNombre() {  
        return nombre;  
    }  
  
    String saluda (Optional<Person> persona) {  
        return persona.flatMap(Person::getNombre)  
            .map("Hola " :: concat)  
            .orElse("Estoy solo");  
    }  
}
```

En este caso hemos variado la firma del método saluda para que siempre devuelva un String. Lo que hace es devolver la cadena de saludo con **flatMap** y una referencia al método getNombre:

6. Si este método devuelve un Optional con valor para String entonces saluda con "Hola " + el nombre.
7. Si el valor devuelto por el flatMap es ausente o bien porque no hay persona o la persona no tiene nombre, entonces devuelve la cadena "Estoy solo".

De esta forma el método siempre devuelve una cadena con valor.

Solo con el uso de **Optional** no obtenemos grandes beneficios en nuestro código, la idea es usar Optional para progresivamente ir eliminando el uso de null de nuestro código. El resultado será un código más explícito sobre cuando se requiere un valor o cuando se permite trabajar sin él, lo que a su vez se traducirá a un código más compacto y con menos errores.

## Streams

### Introducción

**Los streams son una agrupación de elementos sobre los que podemos especificar operaciones.**

Los streams tienen mucho en común con las collections. Para introducirlos veremos cómo traducir un caso sencillo de colecciones a su equivalente a Streams y analizaremos qué diferencias hay y qué ventajas nos proporcionan el uso de Streams.

Supongamos que tenemos una collection de personas y queremos obtener los nombres de estas personas.

- Con collections podríamos crear un método que construyese una lista de nombres de String para guardar el resultado, iterase la lista de personas proporcionadas como parámetro y para cada una de ellas guardar el nombre y devolver la lista de nombres:

```
private List<String> convertCollectionVersion(List<Persona> personas) {  
  
    List<String> nombres = new ArrayList<>();  
    for (Persona persona: personas) {  
        nombres.add(persona.getNombre());  
    }  
  
    return nombres;  
}
```

- Con Streams el punto de partida sería el mismo la lista de personas pasada como parámetro, la clase List como es una Collection dispone de un método stream que nos devuelve un Stream. En este momento disponemos de la nueva API que nos da acceso a las nuevas funcionalidades de la clase Stream:

```

private List<String> convertStreamVersion(List<Persona> personas) {
    return personas.stream()
        .map(it -> it.getNombre())
        .collect(Collectors.toList());
}

```

Uno de esos métodos es **map** con el cual indicamos que queremos realizar una conversión sobre los elementos del Stream y la misma la especificamos como una función que se aplicara sobre cada uno de los elementos, en este caso obteniendo el nombre. A partir de este momento el Stream no es de elementos Persona, sino que es de elementos Stream.

Finalmente recogeremos los elementos del Stream acumulándolos en una estructura de lista, este trabajo lo realiza una estructura de la API basándose en Collectors.

Unas de las ventajas que presenta el código de Streams sobre el código de Collections son:

- El código es declarativo, indicamos que queremos realizar sobre los elementos, no el cómo, por lo que el código es más explícito y compacto.
- Mejor gestión de la complejidad
- Integración con la API funcional, por ejemplo, en el método **map** vemos un ejemplo del uso de lambda expression. La API de Stream esta muy ligada a las interfaces funcionales y permite la reutilización de funciones comunes en múltiples situaciones.
- Uso potencial del paralelismo

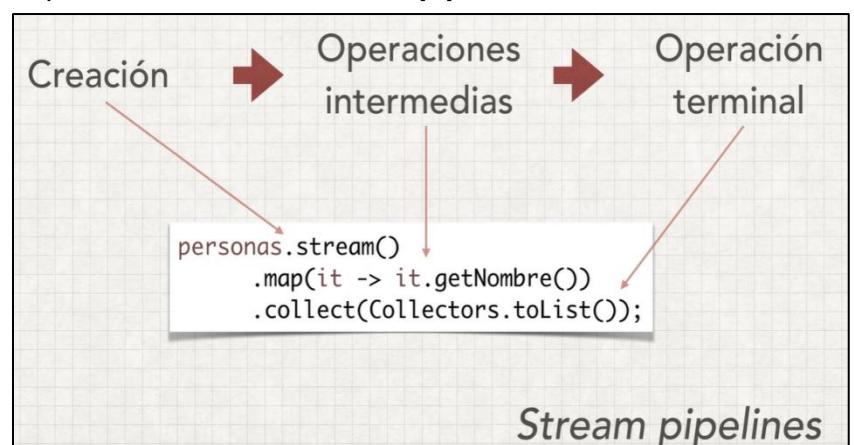
## Stream Pipeline

En la introducción a los Streams hemos visto como estos permiten expresar una secuencia de operaciones que queremos realizar sobre un conjunto de elementos. Vamos a investigar un poco más sobre esto para adentrarnos en lo que se conoce como **stream pipeline**.

El uso típico de un stream consiste en las siguiente etapas:

1. Se produce la creación del Stream
2. Operaciones intermedias sobre el conjunto de elementos
3. Una operación terminal produce el resultado final

Este conjunto de pasos define el llamado **stream pipeline**.



## Creación

Veamos las distintas formas de crear streams:

- **Collection stream():**

La interfaz collection añade un método stream(), por lo que tenemos la posibilidad de crear los mismos en muchas de las clases de la API.

```
List<Persona> personas = Arrays.asList(juan, antonia);
Stream<Persona> s1 = personas.stream();
```

En el ejemplo se crea un Stream a partir de un objeto List.

- **Arrays.asList():**

Podemos utilizar un array como el origen de un stream por medio del método estático stream de la clase Arrays.

```
Arrays.asList() Persona[] personas = {juan, antonia};
Stream<Persona> s2 = Arrays.stream(personas);
```

- **Utilidades de stream:**

La propia interface Stream ofrece distintas formas de creación mediante métodos estáticos, como por ejemplo el método estático of permite pasar una enumeración de objetos para crear un nuevo Stream.

```
Stream<Persona> s3 = Stream.of(juan, antonia);
```

- **Otras APIs:**

Otras APIs se han modificado en Java 8 para permitir la creación de Streams, por ejemplo, la clase Files que permite obtener un Stream para operar sobre las líneas de un fichero.

```
Stream<String> s4 = Files.lines("fichero.txt");
```

## Operaciones intermedias principales

Una vez creado el Stream podemos realizar con él una serie de operaciones intermedias.

Las operaciones intermedias principales por realizar las podemos dividir en tres tipos:

- Convertir
- Filtrar
- Ordenar

### Convertir

Una operación de conversión transforma los elementos del Stream, por ejemplo, podemos al método **map** una función de Persona a String:

```
Stream<String> nombres = personas.map(it -> it.getNombre());
```

String

Persona

$f : \text{Persona} \rightarrow \text{String}$   
Function<Persona, Nombre>

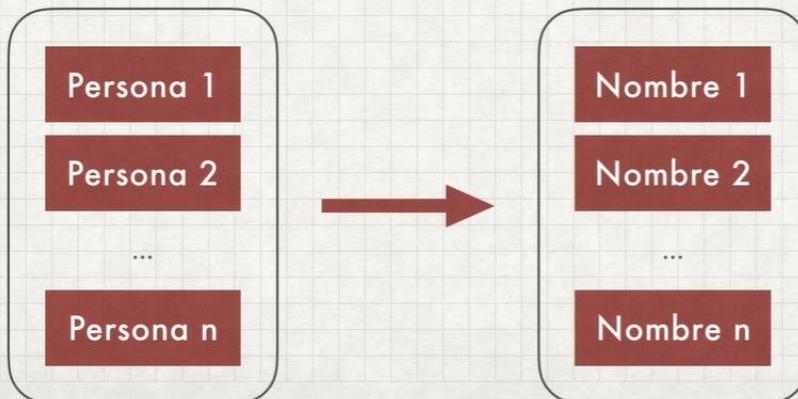
El resultado es actuar sobre un Stream del objeto de tipo Persona y devolver un String de cadenas de caracteres.

Para ser más estrictos la función que acepta en este caso el método map es una función de Persona o de algún tipo superclase de Persona a String o a algún tipo subclase de String:

Function<? super Persona, ? extends String>

El Stream original de personas se convierte en un Stream de nombres:

```
Stream<String> nombres = personas.map(it -> it.getNombre());
```



Una operación de conversión produce un nuevo Stream con las siguientes características:

- Mismo número de elementos que el Stream inicial.
- Mismo orden de elementos que el Stream inicial.
- Los elementos pueden tener un tipo diferente al tipo original.

## Filtrar

Consiste en filtrar los elementos del Stream.

Por ejemplo, podemos usar el método **filter** que acepta un Predicate:

- Si devuelve true sobre un objeto este se conserva en el nuevo Stream.
- Si devuelve false no forma parte de él.

```
Stream<String> nombresA = nombres.filter(it -> it.startsWith("A"));
```

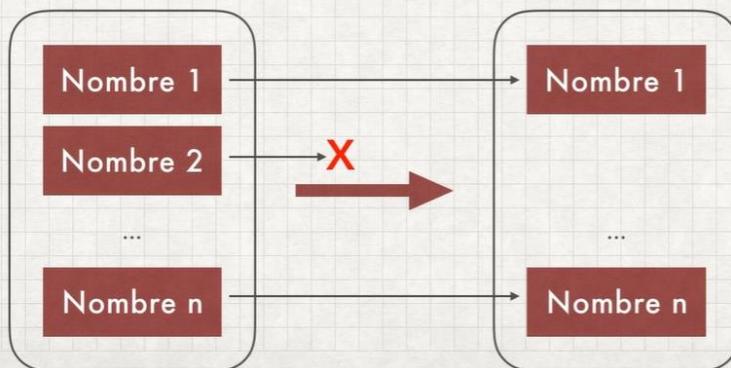
$f : String \rightarrow boolean$

Predicate<String>

Para ser más específicos diremos que acepta que acepta un predicate sobre String o algún supertipo de String:

Predicate<? super String>

```
Stream<String> nombresA = nombres.filter(it -> it.startsWith("A"));
```



Una operación de filtrado produce un nuevo Stream con las siguientes características:

- El nuevo Stream tiene distinto número de elementos al Stream inicial.
- Los elementos que siguen en el nuevo Stream están en el mismo orden.
- Los elementos son del mismo tipo que el Stream original.

## Ordenar

Consiste en ordenar un Stream, esto se logra por medio de la ejecución del método **sorted** obtenemos un stream ordenado.

Al método sorted debemos pasarle un Comparator o una lambda expression que lo produzca.

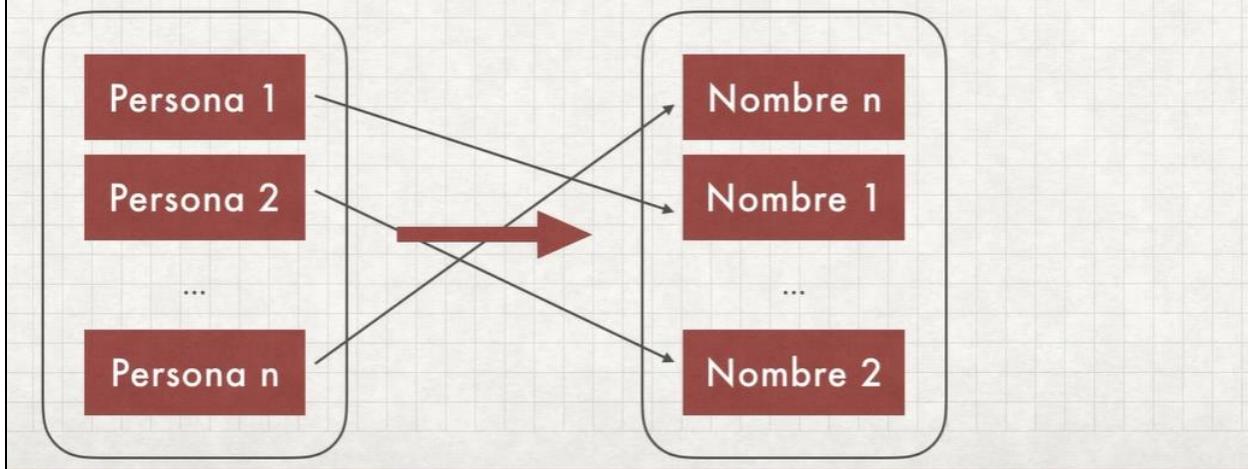
En nuestro caso el comparador debe ser de String o de algún super tipo de String:

```
nombres.sorted((o1, o2) -> o2.length() - o1.length())
```

Comparator<? super String>

También existe otra versión del método sorted sin parámetros que realizara una ordenación natural, es decir siguiendo una noción de orden del tipo contenido en el Stream.

```
nombres.sorted((o1, o2) -> o2.length() - o1.length())
```

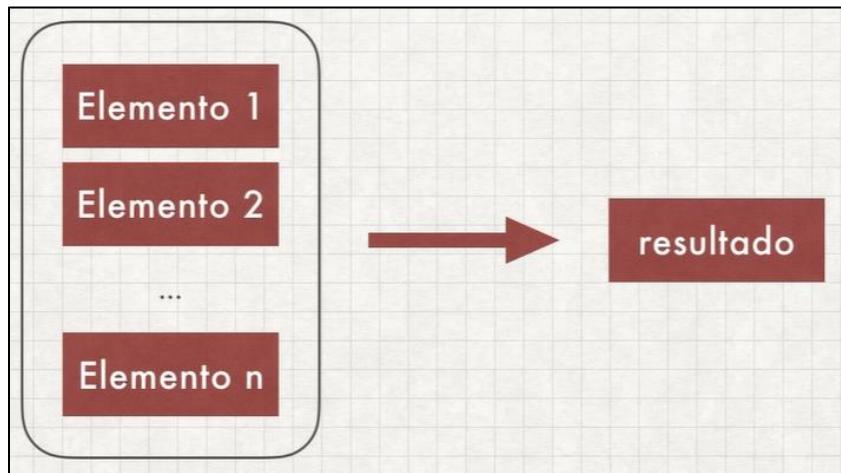


Una operación de ordenación produce un nuevo Stream con las siguientes características:

- El nuevo Stream tiene el mismo número de elementos que el original.
- Los elementos están en distinto orden (ordenados).
- Los elementos son del mismo tipo que el Stream original.

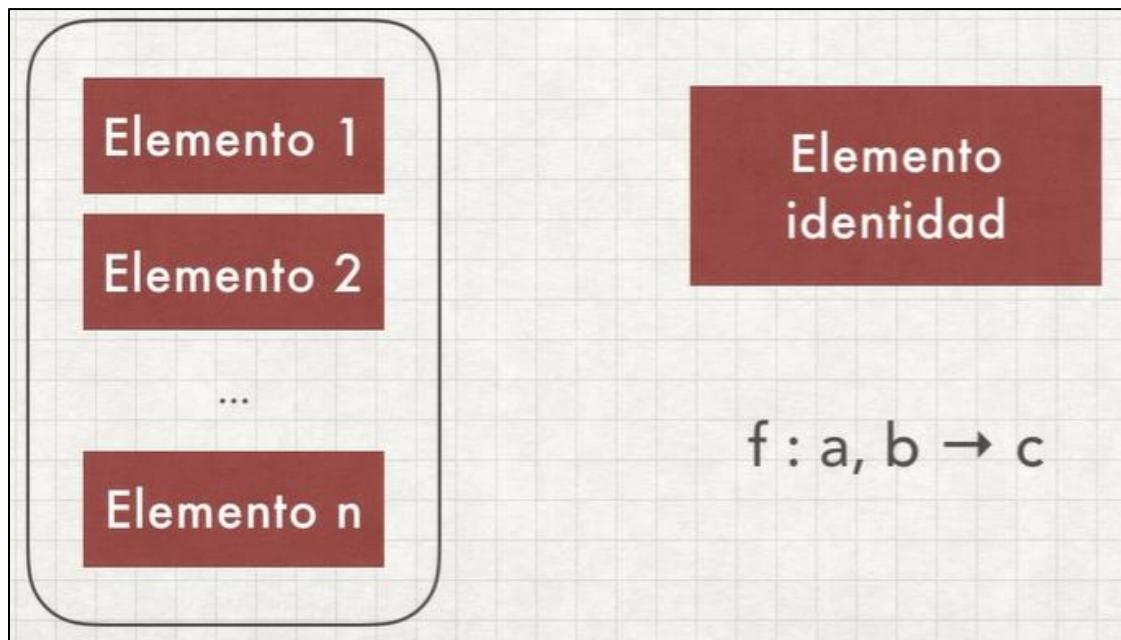
## Operaciones terminales

Un ejemplo de operación terminal es **reduce**, el cual opera sobre los elementos de un Stream para producir un resultado final.



En una de sus formas habituales los ingredientes de **reduce** son:

- El Stream original sobre el cual invocamos el método
- Un elemento identidad o inicial
- Un bioperador entre dos objetos del tipo contenido en el Stream, el cual produce un nuevo objeto del mismo tipo



Para dar un ejemplo supongamos que tenemos un Stream de cadenas y queremos obtener un Stream con una cadena resultado con todos los elementos separados por saltos de línea.

En este caso necesitaremos:

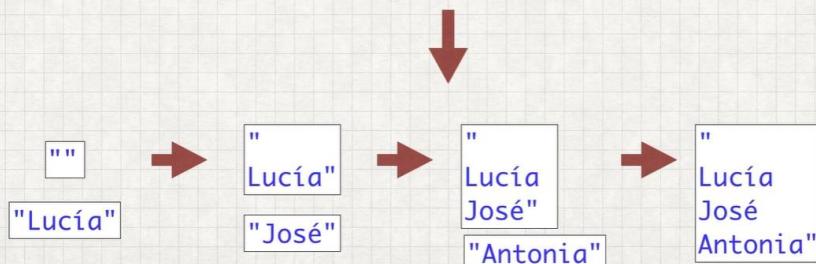
- Elemento identidad que será la cadena vacía ("")
- Operación a realizar sería concatenar los elementos añadiendo un salto de línea entre ellos:

$$(a, b) \rightarrow a + "\n" + b$$

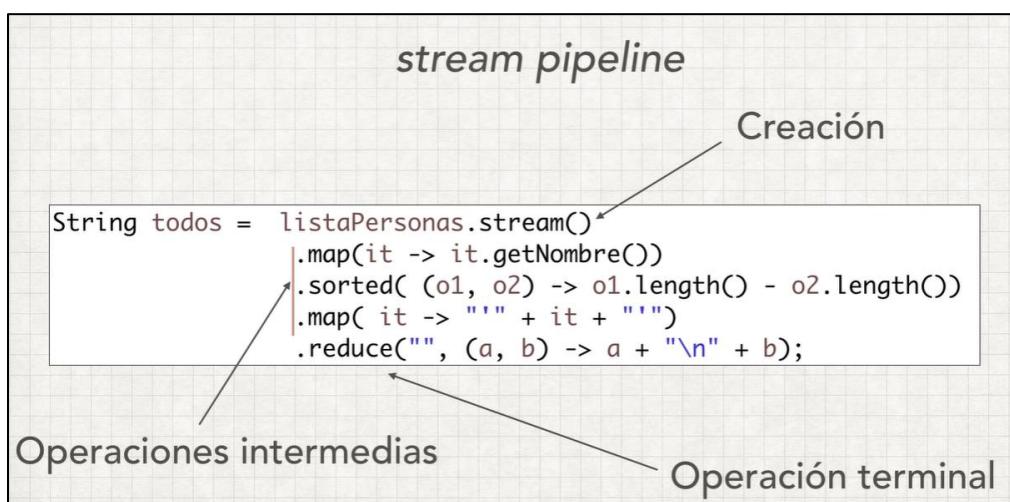
La forma de operación de **reduce** es realizar la operación indicada entre todos los elementos del Stream empezando con el elemento identidad.

Veamos un ejemplo:

```
String todos = Stream.of("Lucía", "José", "Antonia")
    .reduce("", (a, b) -> a + "\n" + b);
```



En este caso partimos de la creación de un Stream por el método `of` y pasandole los elementos como argumentos y luego realizar la operación de **reduce** con los parámetros indicados. Se realizarán sucesivas operaciones de concatenación con un salto de línea intermedio, partiendo del elemento identidad y actuando sobre cada uno de los elementos del Stream.



## Lazyness

En esta sección veremos como se ejecutan en el tiempo las operaciones que forman el Stream pipeline.

Veremos una estrategia utilizada en otros ámbitos se llamada **lazyness** a la opción de dejar el trabajo para el último momento y esto es precisamente lo que hace un Stream.

Lo que viene a responder este concepto es en qué momento se realizan todas las operaciones del Stream pipeline:

```
String todos = listaPersonas.stream()
    .map(it -> it.getNombre())
    .sorted( o1, o2) -> o1.length() - o2.length())
    .map( it -> " " + it + " ")
    .reduce("", (a, b) -> a + "\n" + b);
```

Pongamos el siguiente ejemplo:

```
Stream<String> parcial = listaPersonas.stream()
    .map(it -> it.getNombre());
```

En este caso debemos preguntarnos si ¿se ejecuta el método `getNombre()` sobre un elemento de la lista de personas? **La respuesta es NO, las operaciones del pipeline se van a ejecutar lo más tarde posible.**

**El pipeline no se inicia hasta encontrar una operación terminal.** Es esta operación la que dispara la ejecución del pipeline.

Ahora volviendo al ejemplo anterior, ¿Qué produce el método `map`?

La respuesta es que `map` ejecutado sobre el `Stream<Persona>` devuelve un `Stream<String>`. Las operaciones sobre este Stream van a obtener los elementos de resultado de ejecutar `map` que a su vez va a obtener los resultados del Stream original.

```
Stream<String> parcial = listaPersonas.stream()
    .map(it -> it.getNombre());
```

↑  
¿Que produce map?

Stream<Persona>



Stream<String>

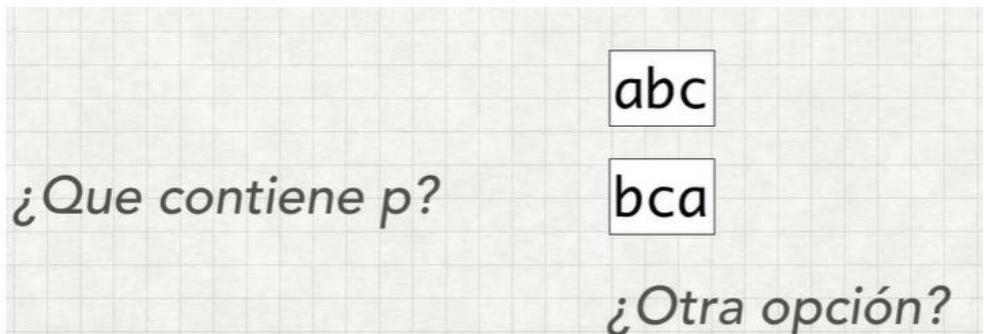
Sigamos viendo los efectos del pipeline sobre los Stream, con el siguiente ejemplo:

```
Stream<String> strings = Stream.of("b", "c", "a");  
  
strings.sorted();  
  
String p = strings.reduce("", String::concat);
```

strings referencia un Stream creado a partir de 3 elementos, luego se ejecuta el método sorted sobre dicha agrupación y el resultado obtenido por este método se ignora.

Luego se ejecuta el método reduce con una concatenación de los elementos y se asigna el resultado a la variable p.

La pregunta para responder es: ¿cuál es el contenido de p?



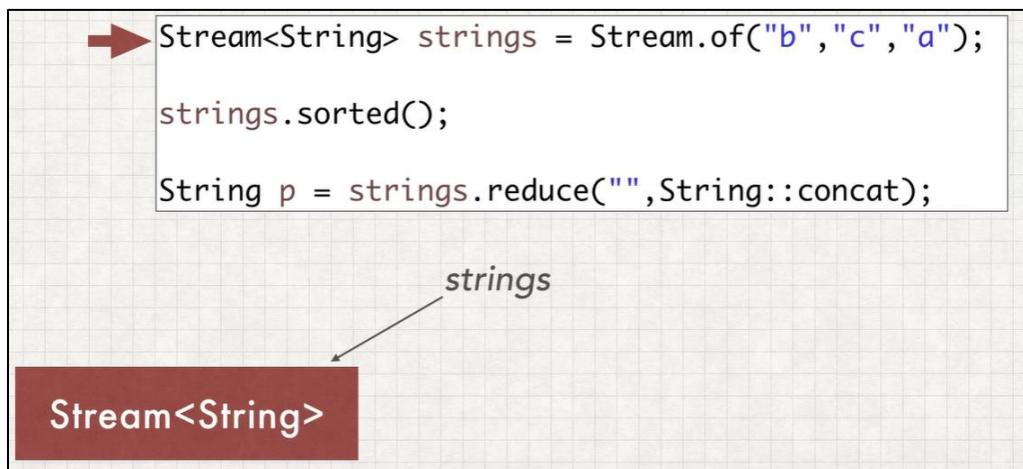
Si ejecutamos el código el resultado obtenemos es una excepción:

```
Exception in thread "main" java.lang.IllegalStateException:  
    stream has already been operated upon or closed  
    at java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:229)  
    at java.util.stream.ReferencePipeline.reduce(ReferencePipeline.java:474)  
    at org.formacion.Main.main(Main.java:16)
```

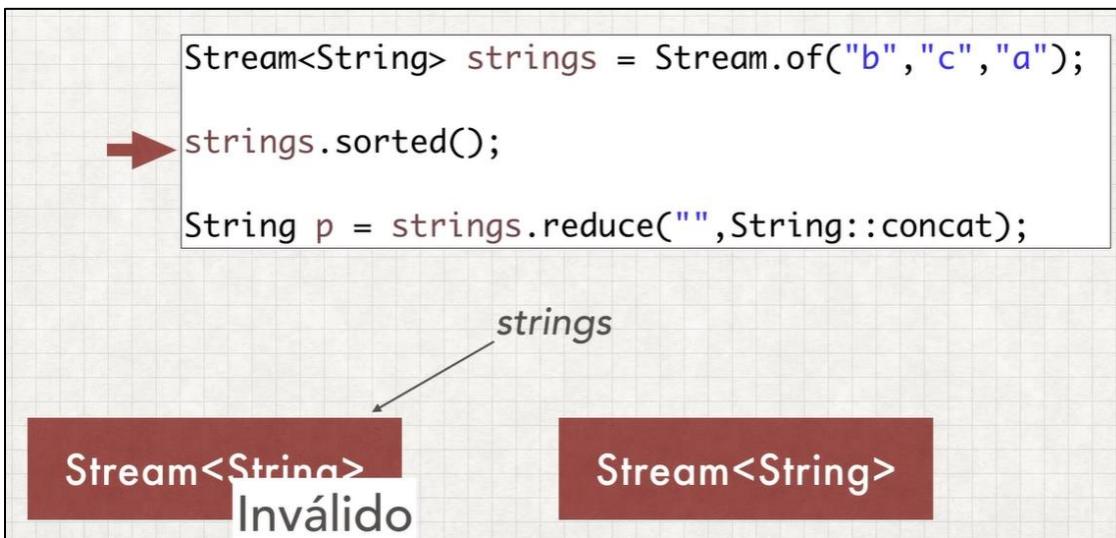
Esto es debido a que en la tercera línea estamos intentando usar un Stream que como indica el error ya está cerrado o se ha ejecutado sobre él alguna operación.

Sigamos la ejecución paso a paso:

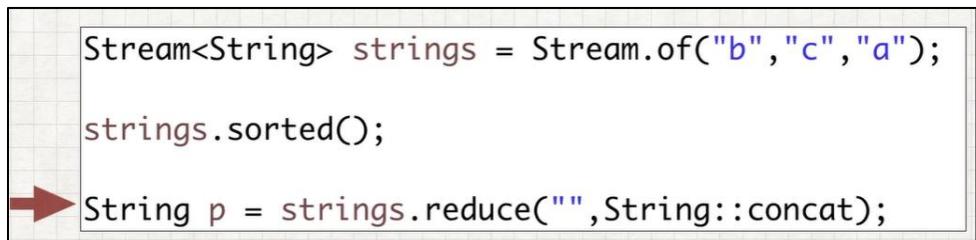
1. Se crea el Stream a partir de las 3 cadenas pasadas al método of. El resultado es un Stream<String> que es asignado a la variable strings.



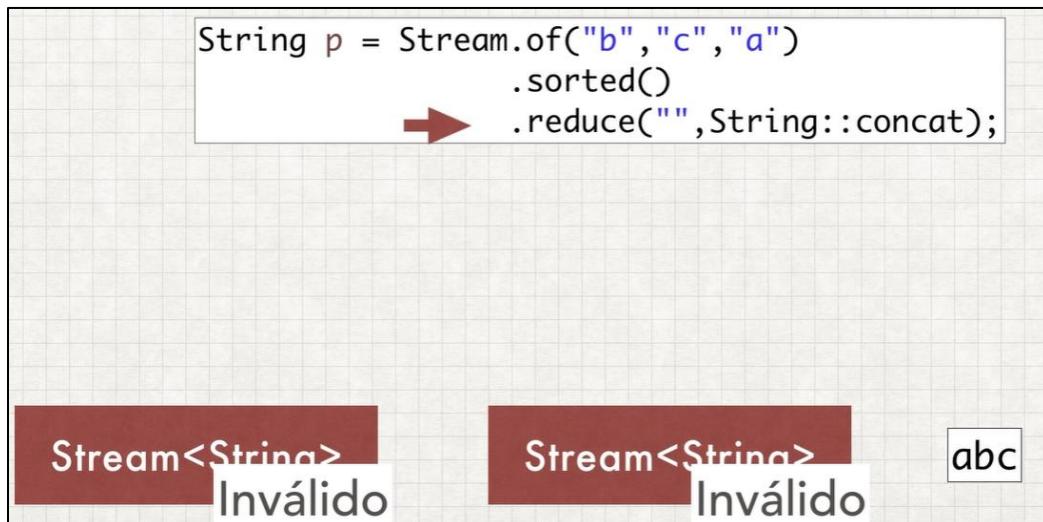
2. A continuación, se ejecuta el método sorted, el cuál produce un nuevo Stream también del tipo String que es devuelto por el método e ignorado, por lo que la variable strings sigue referenciando a Stream original. La ejecución del método sorted además de producir el nuevo Stream deja al primero invalido.



3. Cuando se ejecuta el método reduce no actúa sobre el nuevo Stream creado sino sobre el original que ha quedado invalido y por lo tanto se produce el error:



Evidentemente si ejecutamos el pipeline mediante un **method chaining** el resultado es correcto. Se crea el Stream original con el método of, se ejecuta el método sorted que crea el nuevo Stream y deja el anterior inválido y sobre este objeto creado se ejecuta el método reduce que construye el resultado final asignado a la variable p y deja el Stream inválido:



## Reducciones “MUTABLE”

En esta sección vamos a ver una alternativa para realizar operaciones terminales sobre un Stream, como así también los inconvenientes pueden presentar las operaciones de reduce y como se pueden resolver usando una estructura **mutable** para ir construyendo resultados.

En secciones anteriores vimos que una de las operaciones terminales sobre un Stream era el método **reduce**, en este caso partiendo desde un Stream vacío va a ir construyendo nuevos streams a partir del resultado parcial concatenando el nuevo elemento que encuentra en el Stream:

```
String p = Stream.of("b", "c", "a")
    .sorted()
    .reduce("", String::concat);
```

El problema de esta implementación es que, aunque funcione no es eficiente. Cada paso del stream para procesar un elemento construye una nueva cadena con un resultado parcial, en este caso en lugar de trabajar con una estructura intermedia como String (que es inmutable) que requiere crear un nuevo objeto para cada modificación para cada paso, sería mejor trabajar con un tipo **mutable** como StringBuilder.

Una implementación más eficiente crearía un único objeto StringBuilder que iría acumulando el resultado, podrías por ejemplo usar el método **forEach** que sobre un Stream acepta un Consumer indicando la operación a realizar sobre cada elemento.

El siguiente código primero crea un Stream a partir de 3 Strings, luego crea un StringBuilder que será el que irá acumulando el texto resultado y finalmente ejecuta el método forEach de stream pasando un Consumer. **Esta operación terminal (forEach)** ejecuta el Consumer sobre cada uno de los elementos del Stream.

```
Stream<String> stream = Stream.of("a","b","c");  
  
StringBuilder sb = new StringBuilder();  
stream.forEach(s->sb.append(s));
```

Esta implementación es mucho más eficiente que la implementación inicial con el método reduce en el caso de que el Stream sea suficientemente grandes.

Introducimos de esta manera el método terminal forEach:

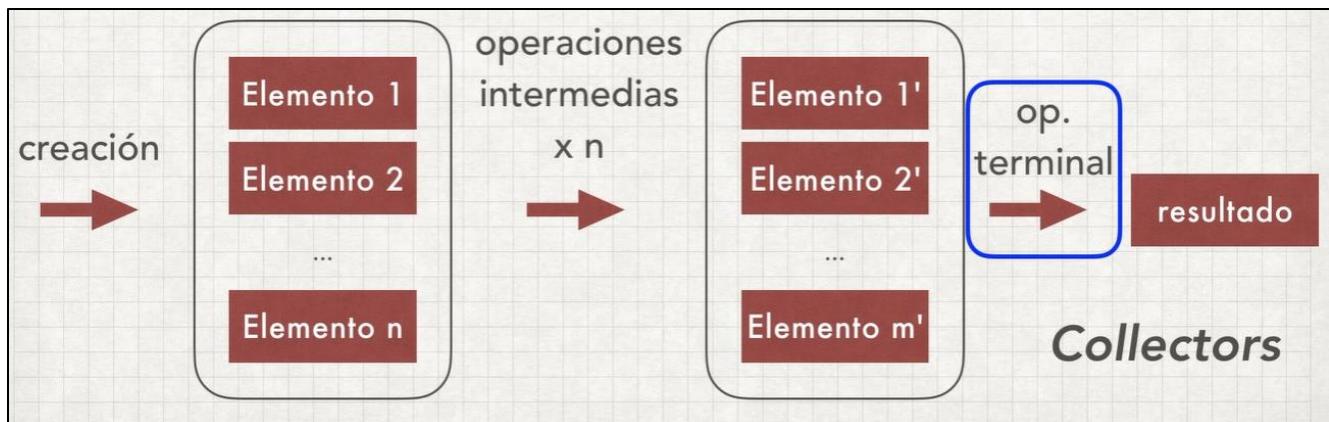
```
void forEach(Consumer<? super T> action);
```

El cuál además de finalizar el Stream no devuelve ningún valor. Toda su acción se basa en los efectos colaterales que produce la acción que se pasa como Consumer por parámetro.

El uso típico es usar como Consumer una modificación sobre una estructura mutable como por ejemplo el StringBuilder.

## Collectors

La Clase Collectors nos ofrece un catálogo de recursos que podemos utilizar de alternativa para construir las operaciones terminales de un Stream.



## Veamos un ejemplo:

Una necesidad muy habitual a la hora de trabajar con Collections es tener que transformar la colección original en otra, por ejemplo, realizando algún tipo de agrupación:

El siguiente código construido en Java 1.7 transforma una lista de alumnos guardada en la variable `alumnos` en un map donde los alumnos están clasificados según su escuela.

```
List<Alumno> alumnos = ...  
  
Map<String, List<Alumno>> porEscuela = new HashMap<>();  
  
for (Alumno alumno: alumnos) {  
    if (! porEscuela.containsKey(alumno.getEscuela())) {  
        porEscuela.put(alumno.getEscuela(), new ArrayList<>());  
    }  
    porEscuela.get(alumno.getEscuela()).add(alumno);  
}
```

Todo proceso se realiza de forma manual, desde la creación de la estructura resultado, la iteración de los alumnos o el hecho de probar si es el primer alumno que encontramos para una escuela por lo que debemos crear una entrada nueva.

Con los Streams la conversión entre estructuras donde se aplica una regla para transformar, agrupar o ordenar los resultados, se puede expresar de una forma mucho más conveniente utilizando **Collectors**.

**Los Collectors son estructuras modificables que reciben los elementos del Stream y construyen un resultado.**

Para nuestro caso de ejemplo devolver todos los alumnos agrupados por escuela nos bastaría con esta sentencia:

```
porEscuela = alumnos.stream()  
                .collect(Collectors.groupingBy(Alumno::getEscuela));
```

Donde utilizamos el método **collect** como operación terminal que recibe como parámetro un `Collectors` que construye el método estático `groupingBy` de `Collectors`. Este método necesita una función que le indique que criterio usar para la agrupación, en nuestro caso el método `getEscuela` de `Alumno`.

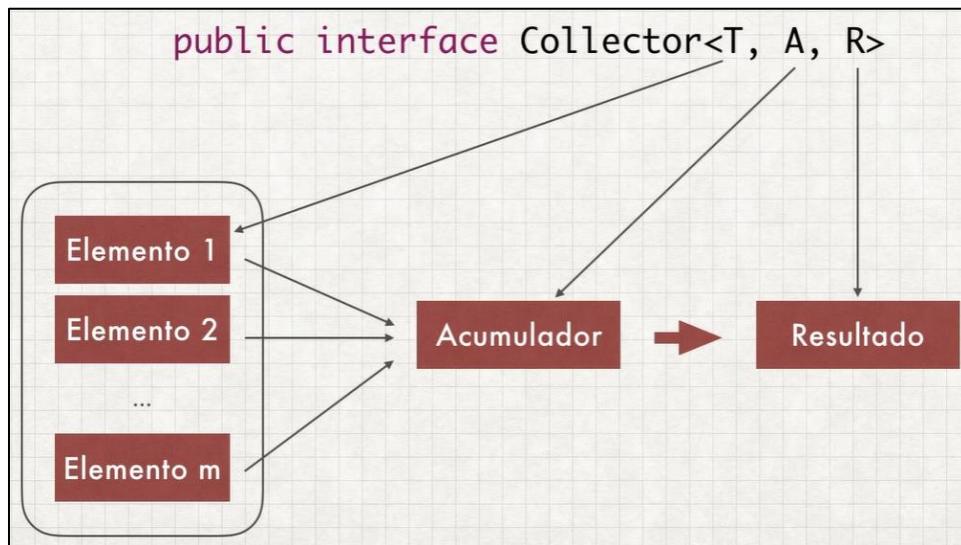
Podemos mejorar un poco el código podemos importar los nombres de los `Collectors` con un `import static` y ya podemos usar el método `groupingBy` directamente, compactando aún más la sentencia:

```
import static java.util.stream.Collectors.*;  
  
porEscuela = alumnos.stream().collect(groupingBy(Alumno::getEscuela));
```

## ¿Cómo se define un Collector?

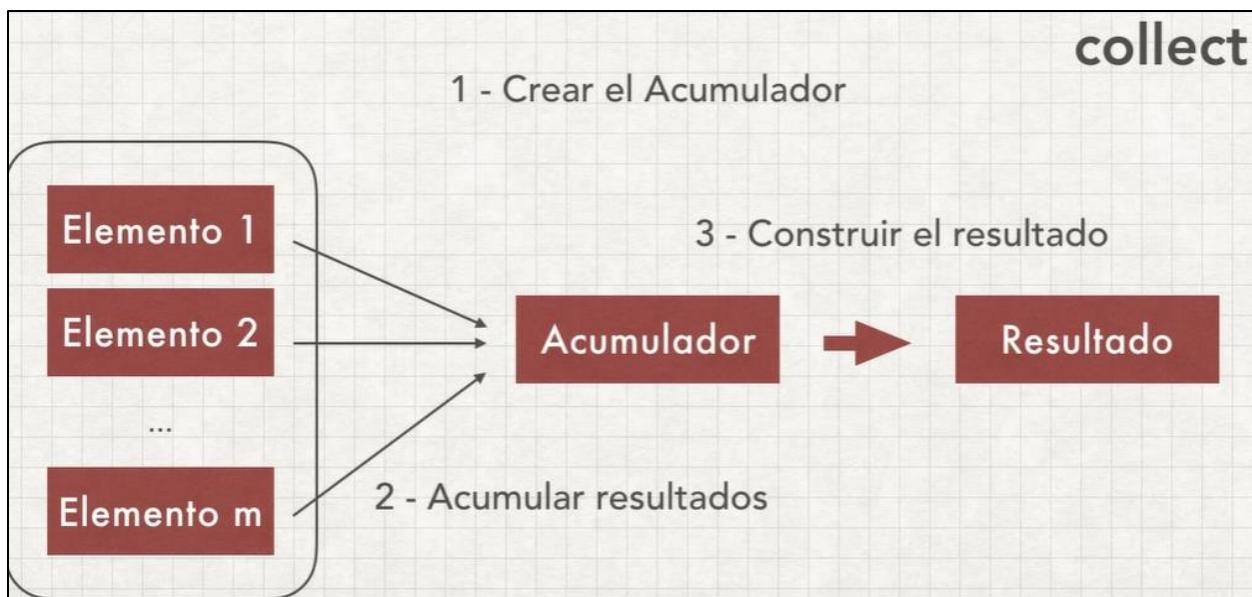
Como ya hemos dicho la tarea de un Collector es ir recibiendo todos los elementos del Stream realizando la tarea de acumulación y producir un determinado resultado. La interface que representa este tipo Collector tiene tres parámetros tipo:

- **T** representa los elementos tipo de Stream
- **A** representa el tipo que realiza las tareas de acumulación
- **R** representa el resultado devuelto por el método collect

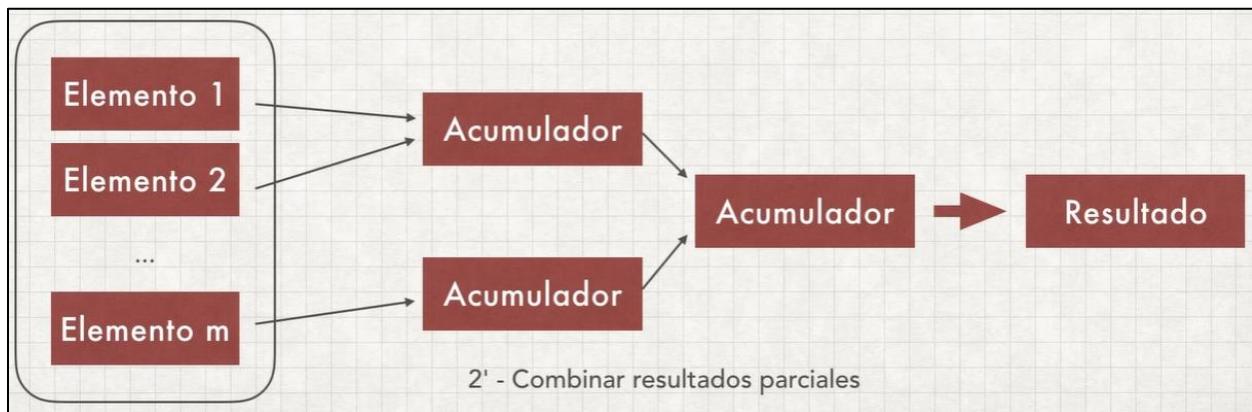


Una vez tenemos el Collector la tarea a realizar por el método **collect** es:

1. **Crear el acumulador** que va a ser invocado
2. Para cada elemento del Stream se realiza la **acumulación de resultados**
3. **Produce el resultado** a partir del estado almacenado en el acumulador.

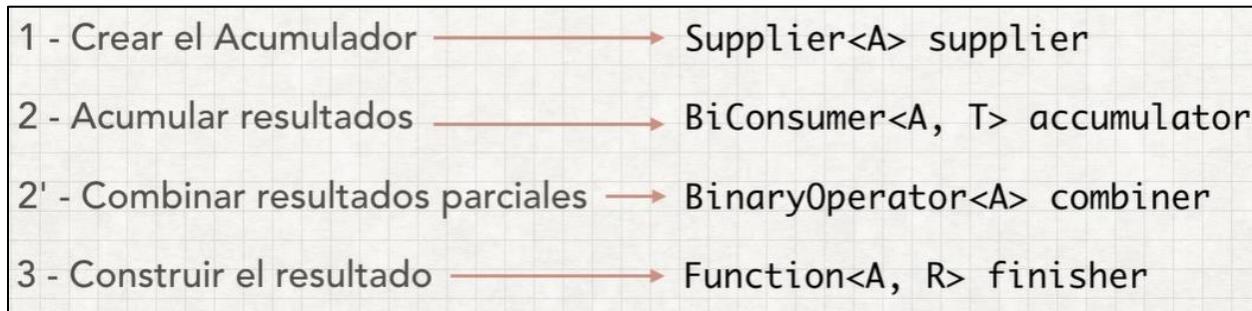


Alternativamente si se dan las circunstancias necesarias se puede producir el proceso de **collect** de forma paralela, podría ser el caso que, para mejorar el rendimiento, distintos hilos de ejecución fuesen procesando los elementos del Stream simultáneamente. En dicho caso tendríamos distintos acumuladores recibiendo elementos del Stream y produciendo resultados parciales. Esto hace necesario un nuevo paso que consiste en combinar estos resultado parciales y crear el acumulador final, con los datos de todos los elementos del Stream:



Todas estas tareas están representadas por interfaces funcionales sobre los tipos del Collector:

1. Crear el Acumulador se corresponde a crear un Supplier para el tipo A (tipo del acumulador).
2. Acumular resultados en el acumulador se representa como un BiConsumer con dos parámetros, A el acumulador y T el elemento de tipo que recibe.
- 2'. Combinar resultados parciales se corresponde con un BinaryOperator.
3. Construir el resultado no tiene porque corresponderse con el tipo usado para el acumulador, para esto se necesita una función entre el tipo acumulador y el resultado final el forma de Function<A, R> la cual recibe el nombre de finisher.



Con estos ingredientes podríamos crear nuestro propio Collector. Generalmente usaremos un Collector creado a partir del catalogo que se nos ofrece, pero no tiene porque ser necesariamente así, ya que podemos crear un Collector que se ajuste a nuestras necesidades particulares, proporcionando las funciones supplier, accumulator, combiner y finisher al método **of** de Collector.

```
public static<T, R> Collector<T, R, R> of (Supplier<R> supplier,
                                             BiConsumer<R, T> accumulator,
                                             BinaryOperator<R> combiner,
                                             Function<A, R> finisher,
                                             Characteristics... characteristics)
```

El parámetro final llamado **Characteristics** es un enum que describe algunas características que un Collector puede o no cumplir y afectan a cuando es posible realizar algún tipo de optimización.

Las características definidas son:

- **CONCURRENT**: indica si el collector puede ser invocado de forma concurrentemente o no.
- **UNORDERED**: indica si el resultado del Collector depende del orden en que recibe los elementos.
- **IDENTITY\_FINISH**: indica si el finisher es la función identidad, es decir si el resultado final es el propio acumulador o no.

## Métodos

Collectors ofrece un amplio catálogo de “acumuladores” típicos como, por ejemplo:

- Para realizar Agrupaciones
- Operaciones aritméticas (promedios, sumas)
- Conversión a Collections (List, Set, Map)

### Agrupaciones

Ya hemos visto como pasando una función al método groupingBy el collect produce un map donde los elementos del Stream quedan clasificados por el resultado de ejecutar esta función:

Por ejemplo si el método getMunicipio() sobre una persona produce un String con el nombre del municipio el map resultado tendrá una entrada para cada municipio que tenga personas y para esta entrada el valor será la lista de personas de este municipio:

```
Map<String, List<Persona>> porMunicipio =
    personas.stream()
        .collect(Collectors.groupingBy(Persona::getMunicipio));
```

"municipio 1"	[ Persona1, Persona2 ]
"municipio 2"	[ Persona n, Persona m ... ]
"municipio 3"	[ Persona y, Persona z ... ]

## Operaciones aritméticas

Disponemos de Collectors que realizan operaciones aritméticas como sumatorias o promedios, si proporcionamos un elemento que convierta los valores del stream a un tipo numérico, por ejemplo:

Este código muestra la edad media de las personas del Stream haciendo uso del método **averagingInt**.

```
Double edadMedia = personas.stream()
    .collect(Collectors.averagingInt(Persona::getEdad));
```

## Conversión a Collections

El Collectors devuelto por **toSet()** recoge en un Set los elementos del Stream.

```
Set<Persona> set = personas.stream().collect(Collectors.toSet());
```

## Combinación de collectors

Los Collectors se pueden combinar, por ejemplo, queremos obtener un map para cada entrada por cada municipio con personas en el Stream, pero en vez de guardar la lista de personas de este municipio, queremos su edad media. Para esto combinaremos el groupingBy con el averagingInt:

```
Map<String, Double> porMunicipio = personas.stream().collect(
    Collectors.groupingBy(
        Persona::getMunicipio,
        Collectors.averagingInt(Persona::getEdad)));
```

"municipio 1"	edad media municipio1
"municipio 2"	edad media municipio 2

## Parallel Stream

Una de las ventajas importantes de los Streams comparados con las estructuras iterativas clásicas con las versiones anteriores a Java 8 es que expresan el que se quiere realizar y no tanto el cómo hacerlo. Esto a parte de mejorar la legibilidad del código, permite actuar sobre la implementación permitiendo por ejemplo optimizar esta ejecución.

Hoy en día con las nuevas arquitecturas una de las formas más evidentes de mejorar el rendimiento de las aplicaciones es usando **parallelismo**.

Stream nos ofrece paralelismo por medio del método **parallelStream()**, el Stream devuelto por este método permitirá en según que situaciones la ejecución paralela de las operaciones sobre él.

**default Stream<E> parallelStream()**

```
/**  
 * Returns a possibly parallel {@code Stream} with this  
 * collection as its source. ...  
 */
```

Y ¿por qué posiblemente?

Veamos un ejemplo:

Tenemos un ejemplo de stream que utiliza el método reduce para obtener la suma de enteros. A reduce le pasamos un valor inicial 0 y un BiOperador que suma los elementos y ejecutando el siguiente código obtendremos que la suma es 55:

```
List<Integer> lista = Arrays.asList(1,2,3,4,5,6,7,8,9,10);  
Integer suma = lista.stream()  
    .reduce(0, (a,b)-> a+b);
```



suma : 55

Ahora si cambiamos la implementación utilizando ahora **parallelStream()**, el resultado es 55 también:

```
List<Integer> lista = Arrays.asList(1,2,3,4,5,6,7,8,9,10);  
Integer suma = lista.parallelStream()  
    .reduce(0, (a,b)-> a+b);
```



suma : 55

¿Cómo podemos saber si este Stream se ha ejecutado de forma paralela o secuencial?

Una forma de poder ver que sucede es utilizando el método **peek**, el cuál recibe un Consumer de los elementos del Stream:

```
List<Integer> lista = Arrays.asList(1,2,3,4,5,6,7,8,9,10);  
Integer suma = lista.parallelStream()  
    .peek(System.out::println)  
    .reduce(0, (a,b)-> a+b);
```

**Peek** no afecta la ejecución del Stream, pero nos permite recibir notificaciones de que elemento se está tratando en cada momento.

De esta manera podemos ver como se ejecuta el Stream con stream() y con parallelStream().

En este caso modificamos nuestro código añadiendo un peek que envía a la consola el elemento que se está tratando en cada momento:

<pre>lista.stream()     .peek(System.out::println)     ...</pre>	<pre>lista.parallelStream()     .peek(System.out::println)     ...</pre>																				
 <table border="1" style="border-collapse: collapse; width: 100%;"><tr><td style="padding: 5px;">1</td></tr><tr><td style="padding: 5px;">2</td></tr><tr><td style="padding: 5px;">3</td></tr><tr><td style="padding: 5px;">4</td></tr><tr><td style="padding: 5px;">5</td></tr><tr><td style="padding: 5px;">6</td></tr><tr><td style="padding: 5px;">7</td></tr><tr><td style="padding: 5px;">8</td></tr><tr><td style="padding: 5px;">9</td></tr><tr><td style="padding: 5px;">10</td></tr></table>	1	2	3	4	5	6	7	8	9	10	 <table border="1" style="border-collapse: collapse; width: 100%;"><tr><td style="padding: 5px;">6</td></tr><tr><td style="padding: 5px;">3</td></tr><tr><td style="padding: 5px;">4</td></tr><tr><td style="padding: 5px;">5</td></tr><tr><td style="padding: 5px;">1</td></tr><tr><td style="padding: 5px;">2</td></tr><tr><td style="padding: 5px;">9</td></tr><tr><td style="padding: 5px;">10</td></tr><tr><td style="padding: 5px;">8</td></tr><tr><td style="padding: 5px;">7</td></tr></table>	6	3	4	5	1	2	9	10	8	7
1																					
2																					
3																					
4																					
5																					
6																					
7																					
8																					
9																					
10																					
6																					
3																					
4																					
5																					
1																					
2																					
9																					
10																					
8																					
7																					

- Si ejecutamos el código con stream() el orden de los elementos recibidos por peek es el esperado.
- Si ejecutamos el código con parallelStream() vemos que el orden ya no es secuencial.

### ¿Y si no es paralelizable?

Supongamos que la operación a realizar ahora es concatenar la lista de números que contiene la variable lista, el código podría similar al siguiente:

```
Convertimos los elementos a String
lista.parallelStream()
    .map(String::valueOf)
    .collect(Collectors.joining(", "));
```

Actua sobre un stream de String,  
concatenando el contenido con el  
separador indicado

### ¿Cuál será el resultado que nos devolverá esta expresión?

Podríamos esperar dos alternativas: lista ordenada o no.

1,2,3,4,5,6,7,8,9,10

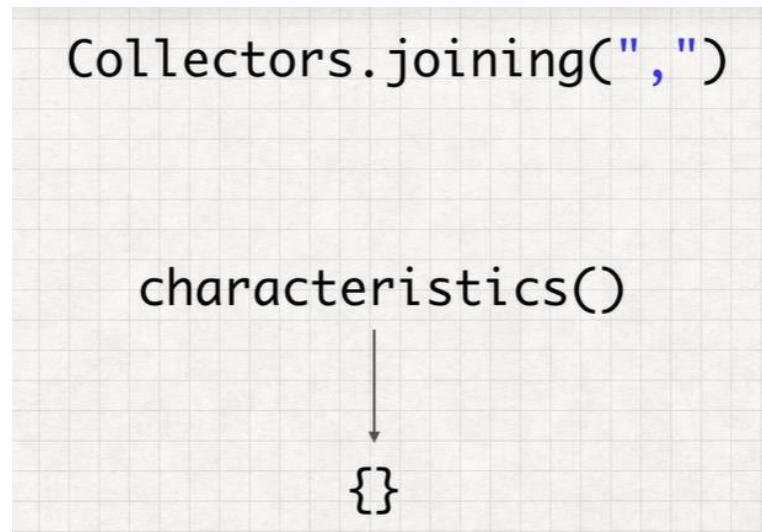
1,4,5,2,3,9,10,6,7,8

En este caso el ejemplo debemos tener en cuenta que estamos haciendo, por ejemplo, una operación de suma que tiene propiedades commutativas y asociativas, se pueden ejecutar en cualquier orden y el resultado es correcto.

La concatenación del ejemplo que queremos hacer no cumple con las condiciones, por lo que se imprime siempre la lista ordenada, es decir el Stream no se va a ejecutar de forma paralela.

### ¿Qué ha hecho el parallelStream()?

En este caso al ser el origen del Stream una lista que es por definición ordenada el Collectors debería tener como mínimo la característica UNORDERED para poderse ejecutar de forma paralela, al no tenerla significa que el resultado de joining depende del orden del que le vienen los elementos y como el Stream esta ordenado la operación no se puede ejecutar de forma paralela:



## Adaptaciones del API

### Collection

En Java 8 se han agregado a la interfaz java.util.Collection dos método que ya hemos visto a la largo de este documento:

```
default Stream<E> stream()
```

```
default Stream<E> parallelStream()
```

## Spliterator

Otro método nuevo en Collection es **Spliterator**, el cuál amplia la funcionalidad ofrecida por iterator:

```
default Spliterator<E> spliterator()
```

Por ejemplo, podemos ejecutar una operación sobre cada elemento de una colección usando el método **forEachRemaining**:

```
nombres.spliterator().forEachRemaining(System.out::println);
```

Un uso más importante de spliterator es interno, la distintas implementaciones de Collection pueden sobrescribir el spliterator devuelto y personalizar dos métodos importantes:

- **trySplit()**: un spliterator devuelve otro spliterator con el que dividir su trabajo, este método es fundamental para paralelizar el proceso de una collection. Se obtiene un spliterator y si hay algún Thread disponible para ayudar en el proceso se invoca al método trySplit(), un Thread actúa sobre el spliterator original y el otro sobre el nuevo spliterator devuelto:

```
Spliterator<T> trySplit();
```

- **characteristics()**: cada collection tiene sus características propias y esto afecta a sus spliterators. Estas características afectan a la forma en que se puede particionar el trabajo sobre los elementos de la collection, algunos ejemplos son: si todos los elementos de la colección son diferentes, si están ordenados y si la colección es inmutable o no.

## Removelf

En las versiones anteriores de Collection ya teníamos métodos para eliminar elementos como **remove** y **removeAll**. En Java 8 se agrega otra opción que es **removelf** al cuál se le pasa un Predicado y su función es eliminar todos los elementos que cumplen el predicado indicado. El valor devuelto será verdadero si se ha producido alguna eliminación y falso si no:

```
default boolean removeIf(Predicate<? super E> filter)
```

**Los métodos que hemos comentado en son heredados por todas las implementaciones de Collection.**

## List

List añade dos métodos en Java 8:

### ReplaceAll

Nos permite sustituir todos los elementos de un array con el resultado de aplicar un operador sobre el elemento.

```
default void replaceAll(UnaryOperator<E> operator)
```

### Sort

Nos permite realizar la ordenación de los elementos pasando un comparador como parámetro, este método es simplemente una simplificación que nos evita tener que usar Collections para esta tarea.

```
default void sort(Comparator<? super E> c)
```

## Map

Map en la versión de Java 8 agrego varios métodos entre los cuales tenemos:

### GetOrDefault

Además del método get que devuelve el valor asociado a una llave o null si no existe, en Java 8 se agrega el método **getOrDefault**, el cuál nos permite indicar que valor nos interesa en caso de no existir la key:

```
default V getOrDefault(Object key, V defaultValue)
```

### ForEach

Disponemos de un forEach para realizar una acción sobre cada entrada del map, ya que cada entrada del map consiste en una clave y valor, al forEach debemos pasarle un BiConsumer:

```
default void forEach(BiConsumer<? super K, ? super V> action)
```

### Replace

Que asigna a la entrada key el valor value pasado como 2do parámetro en caso de no existir ningún valor para la clave no se modifica nada. Este método devuelve el valor original si se ha producido una modificación o null si la entrada para la key no existe.

```
default V replace(K key, V value)
```

**Replace** esta sobrecargado con otra variante que permite especificar el valor anterior requerido:

```
default boolean replace(K key, V oldValue, V newValue)
```

Funciona como el primer replace explicado, pero ahora la modificación solo se realiza si la clave existe y el valor asociado es oldValue.

### PutIfAbsent

Es en cierta manera el complemento de replace, introduce la tupla key-value si no hay entrada para la clave, si ya existe la entrada se deja el valor actual. El valor devuelto es el valor que queda asociado a la clave, el antiguo si ya existía el nuevo si no.

```
default V putIfAbsent(K key, V value)
```

### Remove

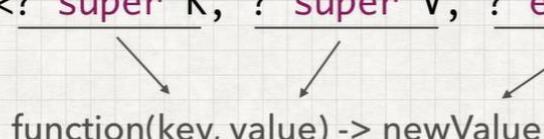
Remove se ha sobrecargado para poder especificar el valor que esperamos encontrar en la entrada a eliminar, solo se eliminará la entrada si la clave existe y si su valor asociado es el valor indicado como parámetro.

```
default boolean remove(Object key, Object value)
```

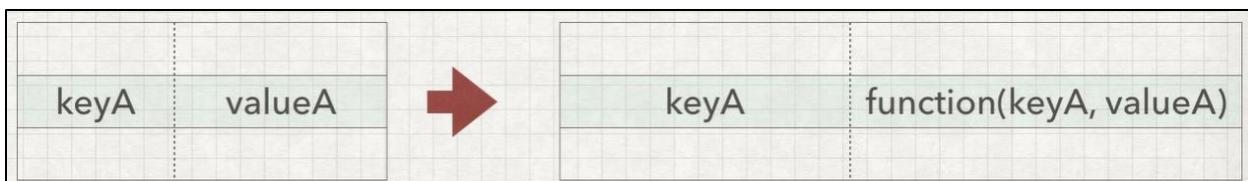
### ReplaceAll

Podemos sustituir todos los valores del map con **replaceAll**, el nuevo valor para una entrada ya existente nos lo proporcionada la BiFunction que pasamos como parámetro, los dos parámetros de la BiFunction son la clave y el valor actual, es decir la función que pasamos que proporciona el nuevo valor recibe como parámetros el key-value actual.

```
default void replaceAll(  
    BiFunction<? super K, ? super V, ? extends V> function)
```



ReplaceAll actuará sobre todas las entradas sustituyendo el valor actual por el resultado de la función sobre los valores que hemos indicado:



## Compute

Con el mismo tipo de función que `replaceAll` podemos actuar no sobre todas las entradas, sino sobre entradas asociadas a una clave particular. Esto lo hacemos mediante el método `compute`, el cuál asocia a la clave el valor devuelto por `remappingFunction`.

```
default V compute(K key,  
BiFunction<? super K, ? super V, ? extends V> remappingFunction)
```

Una diferencia importante con `replaceAll` es que si la función devuelve null se elimina la entrada asociada a esta clave.

## ComputeIfPresent

Es una variante de `compute` que a diferencia del anterior solo actúa si ya existe entrada con la clave indicada y valor no null.

```
default V computeIfPresent(K key,  
BiFunction<? super K, ? super V, ? extends V> remappingFunction)
```

## ComputeIfAbsent

Es el inverso a `computeIfPresent`. En este caso no necesitamos una `BiFunction` ya que por propia definición del método sabemos que ya no habrá valor actual, por lo que la función será solo sobre la clave por lo que en lugar de la `BiFunction` necesitamos una `Function`:

```
default V computeIfAbsent(K key,  
Function<? super K, ? extends V> mappingFunction)
```

## Merge

Nos permite modificar un valor, pero ahora basándonos en el valor proporcionado por parámetro y el valor existente, no tenemos en cuenta la clave.

```
default V merge(K key, V value,  
BiFunction<? super V, ? super V, ? extends V> remappingFunction)
```

Si no existe ninguna entrada para esta clave se usa el valor 1 como nuevo valor, en caso de que existiese una entrada se combina el valor actual con el parámetro realizando una suma entre los dos valores.

```
Map<String, Integer> contadores = ...
```

```
contadores.merge("clave", 1, (a,b) -> a+b);
```

## Comparator

Como ya hemos comentado Comparator pasa a ser en Java 1.8 un interfaz funcional, esto implica que un método que espera un Comparator puede recibir una lambda expression que implemente su método abstract, por ejemplo, ordenar una lista de personas por su fecha de nacimiento:

```
lista.sort((o1,o2)-> o1.getNacimiento() - o2.getNacimiento());
```

El método compare recibe dos objetos del tipo que comparamos y devuelve un entero resultado de la comparación.

Este tipo de comparación que devuelve valores enteros es muy típico, pero es poco sólida, puede fallar en casos que manejen números enteros negativos de valor absoluto muy alto, para solventar esto Comparator ha añadido un nuevo método estático **comparingInt** que permite ordenar basándose en una función que devuelve un entero:

```
public static <T> Comparator<T> comparingInt(  
   ToIntFunction<? super T> keyExtractor)
```

En nuestro caso usamos una referencia al método getNacimiento:

```
lista.sort(Comparator.comparingInt(Persona::getNacimiento));
```

### Formas de reusar comparadores existentes

Para ver esto podemos crear un método estático en Persona que nos devuelva los comparadores que vamos definiendo, así que definimos nuestro método por edad que devuelve un comparador de Persona a partir de su fecha de nacimiento:

```
class Persona {  
    ...  
    static Comparator<Persona> porEdad() {  
        return Comparator.comparingInt(Persona::getNacimiento);  
    }  
}
```

Nuestro código de ordenación queda de la siguiente forma:

```
lista.sort(Persona.porEdad());
```

Si quisieramos invertir la ordenación no tendríamos por qué crear un nuevo comparador, el método **reversed()** sirve para obtener un comparador a partir de invertir el resultado de uno existente:

```
lista.sort(Persona.porEdad().reversed());
```

## Comparing

En la Interface Comparator tenemos el método **comparing** que espera una función que devuelva un objeto comparable.

Por ejemplo, en el siguiente código aprovechamos que el resultado de apellido1 es ordenable (por ser un String) para crear un comparador de Persona:

```
static Comparator<Persona> porApellido1() {  
    return Comparator.comparing(Persona::getApellido1);  
}
```

Comparing tiene una segunda versión para proporcionar un Comparator a usar con el valor devuelto de la función:

```
public static <T, U> Comparator<T> comparing(  
    Function<? super T, ? extends U> keyExtractor,  
    Comparator<? super U> keyComparator)
```

Por ejemplo, se puede utilizar si queremos ordenar según el valor de apellido uno, pero por un criterio distinto al orden natural.

## Composición

Los casos más complejos de comparación a menudo proceden de la composición de casos más simples.

Comparator ofrece el método **thenComparing** en distintas variantes para usar una segunda comparación en caso de que la primera no desempate, es decir devuelva un resultado 0 en el método compare.

```
default Comparator<T> thenComparing(Comparator<? super T> other)
```

En nuestro caso podemos ordenar por Edad y por Apellido1 con el comparador por edad combinándolo con el comparador por apellido1:

```
lista.sort( Persona.porEdad()  
            .thenComparing( Persona.porApellido1()));
```

## Nulls

Los nulls muchas veces nos estropean nuestro código de comparación, pero en Java 8 ya no debe ser así.

Por ejemplo, si nuestra lista puede tener objetos nulos podemos hacer que estas referencias queden al final con el método **nullsLast**, el cuál construye un Comparador que trata los nulls situándolos al final.

```
public static <T> Comparator<T> nullsLast(  
    Comparator<? super T> comparator)
```

El parámetro comparator es el comparador en el que delega la comparación si los dos objetos son no nulls.

En nuestro ejemplo podemos modificar nuestro código para aceptar nulls usando el método nullsLast:

```
lista.sort( Comparator.nullsLast(  
    Persona.porEdad().thenComparing(  
        Persona.porApellido1()))  
)
```

Al igual que existe un nullLast existe un **nullsFirst** con identifica firma que coloca los nulls como es de esperar, al principio.