

JAVA 8

LAMBDAS & STREAM API

@SuperSerch

¿POR QUÉ TENER LAMBDA Y STREAMS?

- ▶ Facilitar la programación concurrente
- ▶ Reducir los errores al programar iteraciones de colecciones

PERSON

```
class Person {  
    int getAge();  
    Sex getSex();  
    PhoneNumber getPhoneNumber();  
    EmailAddr getEmailAddr();  
    PostalAddr getPostalAddr();  
    ...  
}  
  
enum Sex { FEMALE, MALE }
```

ROBOCALL - PERSONAS EN EDAD DE CONDUCIR

```
class MyApplication {  
    List<Person> list = ...  
  
    void robocallEligibleDrivers() {  
        ...  
    }  
    ...  
}
```

ROBOCALL - PERSONAS EN EDAD DE CONDUCIR

```
void robocallEligibleDrivers() {  
    for (Person p : list) {  
        if (p.getAge()>=16){  
            Phonenumbers num = p.getPhoneNumber();  
            robocall(num);  
        }  
    }  
}
```

ROBOCALL - PERSONAS EN EDAD DE VOTAR

ROBOCALL - PERSONAS EN EDAD DE VOTAR

```
void robocallEligibleVoters() {  
    for (Person p : list) {  
        if (p.getAge()>=18){  
            Phonenumbers num = p.getPhoneNumber();  
            robocall(num);  
        }  
    }  
}
```

ROBOCALL - PERSONAS EN EDAD DE BEBER

ROBOCALL - PERSONAS EN EDAD DE BEBER

```
void robocallEligibleDrinkers() {  
    for (Person p : list) {  
        if (p.getAge()>=21){  
            Phonenumbers num = p.getPhoneNumber();  
            robocall(num);  
        }  
    }  
}
```

ROBOCALL - PERSONAS EN EDAD DE BEBER

```
void robocallEligibleDrinkers() {  
    for (Person p : list) {  
        if (p.getAge() >= 21) {  
            Phonenumbers num = p.getPhoneNumber();  
            robocall(num);  
        }  
    }  
}
```

CÓDIGO REPETIDO

SOLUCIÓN PARAMETRIZAR!

SOLUCIÓN PARAMETRIZAR!

```
void robocallOlderThan(int age) {  
    for (Person p : list) {  
        if (p.getAge()>=age){  
            Phonenum num = p.getPhoneNumber();  
            robocall(num);  
        }  
    }  
}
```

USANDO PARÁMETROS

```
void robocallEligibleDrivers() {  
    robocallOlderThan(16);  
}  
void robocallEligibleVoters() {  
    robocallOlderThan(18);  
}  
void robocallEligibleDrinkers() {  
    robocallOlderThan(21);  
}
```

CAMBIAN LOS REQUERIMIENTOS

- ▶ Enviar publicidad a potenciales pilotos comerciales
 - ▶ edad mínima de 23
 - ▶ pero retiro mandatorio a los 65
- ▶ Agregamos otro parámetro y ahora buscamos rangos
 - ▶ usamos Integer.MAX_VALUE para el tope superior si sólo tenemos el límite inferior
 - ▶ usamos 0 para el tope inferior si sólo tenemos el límite superior

CON RANGO DE EDAD

```
void robocallAgeRange(int low, int high) {  
    for (Person p : list) {  
        if (low <= p.getAge() && p.getAge() < high){  
            Phonenum num = p.getPhoneNumber();  
            robocall(num);  
        }  
    }  
}
```

USANDO RANGO DE EDAD

```
void robocallEligiblePilots() {  
    robocallAgeRange(23, 65);  
}  
void robocallEligibleDrivers() {  
    robocallAgeRange(16, Integer.MAX_VALUE);  
}  
void robocallEligibleVoters() {  
    robocallAgeRange(18, Integer.MAX_VALUE);  
}  
void robocallEligibleDrinkers() {  
    robocallAgeRange(21, Integer.MAX_VALUE);  
}
```

UN REQUERIMIENTO MÁS

- ▶ Enviar publicidad a obligados a cumplir servicio militar
 - ▶ con edades entre 18 y 26
 - ▶ sólo hombres

UN REQUERIMIENTO MÁS

- ▶ Enviar publicidad a obligados a cumplir servicio militar
 - ▶ con edades entre 18 y 26
 - ▶ sólo hombres

```
void robocallSelectiveService() {  
    robocallSexAgeRange(Sex.MALE, 18, 26);  
}
```

CON SEXO Y RANGO DE EDAD

```
void robocallSexAgeRange(Sex sex, int low, int high) {  
    for (Person p : list) {  
        if (p.getSex() == sex &&  
            low <= p.getAge() && p.getAge() < high){  
            Phonenum num = p.getPhoneNumber();  
            robocall(num);  
        }  
    }  
}
```

PERO AL COLOCARLO EN LOS CASOS PREVIOS

```
void robocallEligiblePilots() {  
    robocallSexAgeRange(???, 23, 65);  
}  
void robocallEligibleDrivers() {  
    robocallSexAgeRange(???, 16, Integer.MAX_VALUE);  
}  
void robocallEligibleVoters() {  
    robocallSexAgeRange(???, 18, Integer.MAX_VALUE);  
}  
void robocallEligibleDrinkers() {  
    robocallSexAgeRange(???, 21, Integer.MAX_VALUE);  
}
```

PERO AL COLOCARLO EN LOS CASOS PREVIOS

```
void robocallEligiblePilots() {
    robocallSexAgeRange(???, 23, 65);
}
void robocallEligibleDriver() {
    robocallSexAgeRange(???, 16, Integer.MAX_VALUE);
}
void robocallEligibleVoter() {
    robocallSexAgeRange(???, 18, Integer.MAX_VALUE);
}
void robocallEligibleDrinker() {
    robocallSexAgeRange(???, 21, Integer.MAX_VALUE);
}
```

INTENTO #1 DE ARREGLARLO

```
enum Sex { FEMALE, MALE, DONT_CARE }
void robocallSexAgeRange(Sex sex, int low, int high)
{
    for (Person p : list) {
        if ((sex == DONT_CARE || p.getSex() == sex) &&
            low <= p.getAge() && p.getAge() < high){
            Phonenumber num = p.getPhoneNumber();
            robocall(num);
        }
    }
}
```

INTENTO #1 DE ARREGLARLO



```
enum Sex { FEMALE, MALE, DONT_CARE }
void robocallSexAgeRange(Sex sex, int low, int high)
{
    for (Person p : list) {
        if ((sex == DONT_CARE || p.getSex() == sex) &&
            low <= p.getAge() && p.getAge() < high){
            Phonenumber num = p.getPhoneNumber();
            robocall(num);
        }
    }
}
```

INTENTO #1 DE ARREGLARLO

```
enum Sex { FEMALE, MALE, DONT_CARE }
void robocallSexAgeRange(Sex sex, int low, int high)
{
    for (Person p : list) {
        if ((sex == DONT_CARE) || (p.getSex() == sex) &&
            low <= p.getAge() && p.getAge() < high){
            Phonenumber num = p.getPhoneNumber();
            robocall(num);
        }
    }
}
```

IMPLEMENTACIÓN CONTAMINA EL MODELO

INTENTO #2 DE ARREGLARLO - USAR NULL

```
void robocallSexAgeRange(Sex sex, int low, int high) {  
    for (Person p : list) {  
        if ((sex == null || p.getSex() == sex) &&  
            low <= p.getAge() && p.getAge() < high){  
            Phonenum num = p.getPhoneNumber();  
            robocall(num);  
        }  
    }  
}
```

AL COLOCARLO EN LOS CASOS PREVIOS

```
void robocallSelectiveService() {  
    robocallSexAgeRange(Sex.MALE, 18, 26);  
}  
void robocallEligiblePilots() {  
    robocallSexAgeRange(null, 23, 65);  
}  
void robocallEligibleDrivers() {  
    robocallSexAgeRange(null, 16, Integer.MAX_VALUE);  
}
```

AL COLOCARLO EN LOS CASOS PREVIOS

```
void robocallSelectiveService() {  
    robocallSexAgeRange(Sex.MALE, 18, 26);  
}  
void robocallEligiblePilots() {  
    robocallSexAgeRange(null, 23, 65);  
}  
void robocallEligibleDrivers() {  
    robocallSexAgeRange(null, 16, Integer.MAX_VALUE);  
}
```

- ▶ Pero, ¿y si sex==null significa que no se capturó el sexo?

INTENTO #3 AGREGAR UN BOOLEAN

```
void robocallSexAgeRange(Sex sex, int low, int high) {  
    for (Person p : list) {  
        if ((sex == null || p.getSex() == sex) &&  
            low <= p.getAge() && p.getAge() < high){  
            Phonenum num = p.getPhoneNumber();  
            robocall(num);  
        }  
    }  
}
```

AL COLOCARLO EN LOS CASOS PREVIOS

```
void robocallSelectiveService() {  
    robocallSexAgeRange(Sex.MALE, true, 18, 26);  
}  
void robocallEligiblePilots() {  
    robocallSexAgeRange(null, false, 23, 65);  
}  
void robocallEligibleDrivers() {  
    robocallSexAgeRange(Sex.MALE, false, 16,  
Integer.MAX_VALUE);  
}
```

AL COLOCARLO EN LOS CASOS PREVIOS

```
void robocallSelectiveService() {  
    robocallSexAgeRange(Sex.MALE, true, 18, 26);  
}  
void robocallEligiblePilots() {  
    robocallSexAgeRange(null, false, 23, 65);  
}  
void robocallEligibleDrivers() {  
    robocallSexAgeRange(Sex.MALE, false, 16,  
Integer.MAX_VALUE);  
}
```

- ▶ aún hay que poner algo en el campo de sex, incluso si es ignorado

AL COLOCARLO EN LOS CASOS PREVIOS

```
void robocallSelectiveService() {  
    robocallSexAgeRange(Sex.MALE, true, 18, 26);  
}  
void robocallEligiblePilots() {  
    robocallSexAgeRange(null, false, 23, 65);  
}  
void robocallEligibleDrivers() {  
    robocallSexAgeRange(Sex.MALE, false, 16,  
Integer.MAX_VALUE);  
}
```

- ▶ aún hay que poner algo en el campo de sex, incluso si es ignorado
- ▶ ¿Y si olvidamos cambiar el boolean al cambiar la llamada?

LO QUE EL CALLER QUIERE

```
void robocallSelectiveService() {  
    robocallMatchingPersons(*****)  
}
```

LO QUE EL CALLER QUIERE

```
void robocallSelectiveService() {  
    robocallMatchingPersons(*****)  
}
```

- ▶ *** dada una persona, usando características de la persona, decidir cuando esa persona deberá ser robo-llamada

LO QUE LA BIBLIOTECA QUIERE

```
void robocallMatchingPersons(*****) {  
    for (Person p : list) {  
        if (***) {  
            PhoneNumber num = p.getPhoneNumber();  
            robocall(num);  
        }  
    }  
}
```

LO QUE LA BIBLIOTECA QUIERE

```
void robocallMatchingPersons(*****) {  
    for (Person p : list) {  
        if (***) {  
            PhoneNumber num = p.getPhoneNumber();  
            robocall(num);  
        }  
    }  
}
```

LO QUE LA BIBLIOTECA QUIERE

```
void robocallMatchingPersons(*****) {  
    for (Person p : list) {  
        if (***) {  
            PhoneNumber num = p.getPhoneNumber();  
            robocall(num);  
        }  
    }  
}
```

ES UNA FUNCIÓN!

LO QUE LA BIBLIOTECA QUIERE

```
void robocallMatchingPersons(*****) {  
    for (Person p : list) {  
        if (***) {  
            PhoneNumber num = p.getPhoneNumber();  
            robocall(num);  
        }  
    }  
}
```

ES UNA FUNCIÓN!

- ▶ Función dada una persona, usando características de la persona, decidir cuando esa persona deberá ser robo-llamada
- ▶ El parámetro de esa función es una Persona
- ▶ El resultado de esa función es un boolean

¿QUÉ ES UNA EXPRESIÓN LAMBDA?

¿QUÉ ES UNA EXPRESIÓN LAMBDA?

- ▶ Forma matemática que define la transformación de un valor

¿QUÉ ES UNA EXPRESIÓN LAMBDA?

- ▶ Forma matemática que define la transformación de un valor
- ▶ A las cosas que realizan transformaciones las denominamos Funciones

¿QUÉ ES UNA EXPRESIÓN LAMBDA?

- ▶ Forma matemática que define la transformación de un valor
- ▶ A las cosas que realizan transformaciones las denominamos Funciones
- ▶ `int -> int`
Función que transforma de entero a entero

¿QUÉ ES UNA EXPRESIÓN LAMBDA?

- ▶ Forma matemática que define la transformación de un valor
- ▶ A las cosas que realizan transformaciones las denominamos Funciones
- ▶ `int -> int`
Función que transforma de entero a entero
- ▶ Las expresiones Lambda son la manera de agregar funciones a Java

PROPÓSITO DE LAS EXPRESIONES LAMBDA

PROPÓSITO DE LAS EXPRESIONES LAMBDA

- ▶ Proveen una manera sencilla para distribuir procesamiento específico

PROPÓSITO DE LAS EXPRESIONES LAMBDA

- ▶ Proveen una manera sencilla para distribuir procesamiento específico
- ▶ Útiles para procesar colecciones

PROPÓSITO DE LAS EXPRESIONES LAMBDA

- ▶ Proveen una manera sencilla para distribuir procesamiento específico
- ▶ Útiles para procesar colecciones
- ▶ Facilitar la cooperación entre desarrolladores y usuarios de bibliotecas

PROPÓSITO DE LAS EXPRESIONES LAMBDA

- ▶ Proveen una manera sencilla para distribuir procesamiento específico
- ▶ Útiles para procesar colecciones
- ▶ Facilitar la cooperación entre desarrolladores y usuarios de bibliotecas
- ▶ $(T\ p, T2\ q, \dots) \rightarrow \{\text{bloque de código}\}$

¿CÓMO REPRESENTAMOS FUNCIONES EN JAVA?

```
new Thread(new Runnable() {
    public void run() {
        System.out.println("Estoy en otro hilo.");
    }
}).start();
```

```
Collections.sort(people, new Comparator<Person>() {
    public int compare(Person x, Person y) {
        return
x.getLastName().compareTo(y.getLastName());
    }
});
```

LAMBDAS

LAMBDA

- ▶ Bloque de código

LAMBDA

- ▶ Bloque de código
- ▶ Se basan en Target Typing (Según el contexto se determina el tipo de la Lambda)

LAMBDA

- ▶ Bloque de código
- ▶ Se basan en Target Typing (Según el contexto se determina el tipo de la Lambda)
- ▶ existen en dos categorías:

LAMBDA

- ▶ Bloque de código
- ▶ Se basan en Target Typing (Según el contexto se determina el tipo de la Lambda)
- ▶ existen en dos categorías:
 - ▶ Sin efectos colaterales

LAMBDA

- ▶ Bloque de código
- ▶ Se basan en Target Typing (Según el contexto se determina el tipo de la Lambda)
- ▶ existen en dos categorías:
 - ▶ Sin efectos colaterales
 - ▶ Con efectos colaterales

LAMBDA

- ▶ Bloque de código
- ▶ Se basan en Target Typing (Según el contexto se determina el tipo de la Lambda)
- ▶ existen en dos categorías:
 - ▶ Sin efectos colaterales
 - ▶ Con efectos colaterales
- ▶ Siempre asignadas a una Functional Interface

FUNCTIONAL INTERFACE

FUNCTIONAL INTERFACE

- ▶ Interfaces que tienen sólo un método abstracto.

FUNCTIONAL INTERFACE

- ▶ Interfaces que tienen sólo un método abstracto.
- ▶ Los métodos por defecto, dado que ya están implementados, no cuentan para definir si una Interface es funcional o no

FUNCTIONAL INTERFACE

- ▶ Interfaces que tienen sólo un método abstracto.
- ▶ Los métodos por defecto, dado que ya están implementados, no cuentan para definir si una Interface es funcional o no
- ▶ Si la interface declara un método abstracto, pero que sobreescribe un método de `java.lang.Object`, este tampoco cuenta para definir si es funcional o no

FUNCTIONAL INTERFACE

- ▶ Interfaces que tienen sólo un método abstracto.
- ▶ Los métodos por defecto, dado que ya están implementados, no cuentan para definir si una Interface es funcional o no
- ▶ Si la interface declara un método abstracto, pero que sobreescribe un método de `java.lang.Object`, este tampoco cuenta para definir si es funcional o no
- ▶ Multiples Interfaces se pueden heredar y será una interface funcional si cumple con las reglas anteriores

RECONOCIENDO UNA INTERFACE FUNCIONAL

```
interface Runnable {  
    void run();  
}
```

RECONOCIENDO UNA INTERFACE FUNCIONAL

```
interface Runnable {  
    void run();  
}
```

RECONOCIENDO UNA INTERFACE FUNCIONAL

```
interface NonFunc {  
    boolean equals(Object obj);  
}
```

RECONOCIENDO UNA INTERFACE FUNCIONAL

```
interface NonFunc {  
    boolean equals(Object obj);  
}
```

RECONOCIENDO UNA INTERFACE FUNCIONAL

```
interface Func extends NonFunc {  
    int compare(String o1, String o2);  
}
```

RECONOCIENDO UNA INTERFACE FUNCIONAL

```
interface Func extends NonFunc {  
    int compare(String o1, String o2);  
}
```

RECONOCIENDO UNA INTERFACE FUNCIONAL

```
interface Comparator<T> {  
    boolean equals(Object obj);  
    int compare(T o1, T o2);  
}
```

RECONOCIENDO UNA INTERFACE FUNCIONAL

```
interface Comparator<T> {  
    boolean equals(Object obj);  
    int compare(T o1, T o2);  
}
```

RECONOCIENDO UNA INTERFACE FUNCIONAL

```
interface Foo {  
    int m();  
    Object clone();  
}
```

RECONOCIENDO UNA INTERFACE FUNCIONAL

```
interface Foo {  
    int m();  
    Object clone();  
}
```

RECONOCIENDO UNA INTERFACE FUNCIONAL

```
interface X { int m(Iterable<String> arg); }

interface Y { int m(Iterable<String> arg); }

interface Z extends X, Y {}
```

RECONOCIENDO UNA INTERFACE FUNCIONAL

```
interface X { int m(Iterable<String> arg); }  
interface Y { int m(Iterable<String> arg); }  
interface Z extends X, Y {}
```

RECONOCIENDO UNA INTERFACE FUNCIONAL

```
interface X { Iterable m(Iterable<String> arg); }

interface Y { Iterable<String> m(Iterable arg); }

interface Z extends X, Y {}
```

RECONOCIENDO UNA INTERFACE FUNCIONAL

```
interface X { Iterable m(Iterable<String> arg); }  
interface Y { Iterable<+String> m(Iterable arg); }  
interface Z extends X, Y {}
```

RECONOCIENDO UNA INTERFACE FUNCIONAL

```
interface X { int m(Iterable<String> arg); }
interface Y { int m(Iterable<Integer> arg); }
interface Z extends X, Y {}
```

```
interface X { int m(Iterable<String> arg, Class c); }
interface Y { int m(Iterable arg, Class<?> c); }
interface Z extends X, Y {}
```

```
interface X<T> { void m(T arg); }
interface Y<T> { void m(T arg); }
interface Z<A, B> extends X<A>, Y<B> {}
```

RECONOCIENDO UNA INTERFACE FUNCIONAL

```
interface X { int m(Iterable<String> arg); }  
interface Y { int m(Iterable<Integer> arg); }  
interface Z extends X, Y {}
```

```
interface X { int m(Iterable<String> arg, Class c); }  
interface Y { int m(Iterable arg, Class<?> c); }  
interface Z extends X, Y {}
```

```
interface X<T> { void m(T arg); }  
interface Y<T> { void m(T arg); }  
interface Z<A, B> extends X<A>, Y<B> {}
```

RECONOCIENDO UNA INTERFACE FUNCIONAL

```
interface X { int m(Iterable<String> arg); }  
interface Y { int m(Iterable<Integer> arg); }  
interface Z extends X, Y {}
```

```
interface X { int m(Iterable<String> arg, Class c); }  
interface Y { int m(Iterable<?> arg, Class<?> c); }  
interface Z extends X, Y {}
```

```
interface X<T> { void m(T arg); }  
interface Y<T> { void m(T arg); }  
interface Z<A, B> extends X<A>, Y<B> {}
```

RECONOCIENDO UNA INTERFACE FUNCIONAL

```
interface X { int m(Iterable<String> arg); }  
interface Y { int m(Iterable<Integer> arg); }  
interface Z extends X, Y {}
```

```
interface X { int m(Iterable<String> arg, Class c); }  
interface Y { int m(Iterable<?> arg, Class<?> c); }  
interface Z extends X, Y {}
```

```
interface X<T> { void m(T arg); }  
interface Y<T> { void m(T arg); }  
interface Z<A, B> extends X<A>, Y<B> {}
```

RECONOCIENDO UNA INTERFACE FUNCIONAL

```
interface X { long m(); }
```

```
interface Y { int m(); }
```

```
interface Z extends X, Y {}
```

RECONOCIENDO UNA INTERFACE FUNCIONAL

```
interface X { long m(); }  
interface Y { int m(); }  
interface Z extends X, Y {}
```

RECONOCIENDO UNA INTERFACE FUNCIONAL

```
interface Foo<T, N extends Number> {  
    void m(T arg);  
    void m(N arg);  
}
```

```
interface Bar extends Foo<String, Integer> {}
```

```
interface Baz extends Foo<Integer, Integer> {}
```

RECONOCIENDO UNA INTERFACE FUNCIONAL

```
interface Foo<T, N extends Number> {  
    void m(T arg); X  
    void m(N arg);  
}
```

```
interface Bar extends Foo<String, Integer> {}
```

```
interface Baz extends Foo<Integer, Integer> {}
```

RECONOCIENDO UNA INTERFACE FUNCIONAL

```
interface Foo<T, N extends Number> {  
    void m(T arg);  
    void m(N arg);  
}
```

```
interface Bar extends Foo<String, Integer> {}
```

```
interface Baz extends Foo<Integer, Integer> {}
```

RECONOCIENDO UNA INTERFACE FUNCIONAL

```
interface Foo<T, N extends Number> {  
    void m(T arg);   
    void m(N arg);  
}
```

```
interface Bar extends  Foo<String, Integer> {}
```

```
interface Baz extends  Foo<Integer, Integer> {}
```

SINTAXIS DE UNA LAMBDA

(parameters) -> expression

(parameters) -> { statements; }
//return necesario

EJEMPLO - COMPARATOR

```
List<Animal> farm = loadAnimals();
Collections.sort(farm, new Comparator<Animal>() {
    @Override
    public int compare(Animal a1, Animal a2) {
        return a1.name.compareTo(a2.name);
    }
});
```

EJEMPLO - COMPARATOR

```
List<Animal> farm = loadAnimals();
Collections.sort(farm, (Animal a1, Animal a2)
-> a1.name.compareTo(a2.name));
```

EJEMPLO - COMPARATOR

```
List<Animal> farm = loadAnimals();
Collections.sort(farm, (Animal a1, Animal a2)
-> a1.name.compareTo(a2.name));
```

EJEMPLO - COMPARATOR

```
List<Animal> farm = loadAnimals();
Collections.sort(farm,
    (a1, a2) -> a1.name.compareTo(a2.name));
```

EJEMPLOS

```
(int x, int y) -> x + y
```

```
(x, y) -> x - y
```

```
() -> 10
```

```
(String s) -> System.out.println(s)
```

```
x -> x * 2
```

```
c -> { int s = c.size(); c.clear(); return s; }
```

¿QUÉ ES UNA LAMBDA?

¿QUÉ ES UNA LAMBDA?

- ▶ instancias especiales de "objetos"

¿QUÉ ES UNA LAMBDA?

- ▶ instancias especiales de "objetos"
- ▶ con dependencias autocontenidoas (capturadas)

¿QUÉ ES UNA LAMBDA?

- ▶ instancias especiales de "objetos"
- ▶ con dependencias autocontenidoas (capturadas)
- ▶ diseñadas para dar soporte a la iteración interna de colecciones

¿QUÉ ES UNA LAMBDA?

- ▶ instancias especiales de "objetos"
- ▶ con dependencias autocontenidoas (capturadas)
- ▶ diseñadas para dar soporte a la iteración interna de colecciones
- ▶ pueden ser valores de retorno

¿QUÉ ES UNA LAMBDA?

- ▶ instancias especiales de "objetos"
- ▶ con dependencias autocontenidoas (capturadas)
- ▶ diseñadas para dar soporte a la iteración interna de colecciones
- ▶ pueden ser valores de retorno
- ▶ pueden serializarse (mediante un cast)

INTERFACES FUNCIONALES EN JDK 8

INTERFACES FUNCIONALES EN JDK 8

- ▶ Supplier<T>

INTERFACES FUNCIONALES EN JDK 8

- ▶ Supplier<T>
- ▶ Consumer<T>

INTERFACES FUNCIONALES EN JDK 8

- ▶ Supplier<T>
- ▶ Consumer<T>
- ▶ Predicate<T>

INTERFACES FUNCIONALES EN JDK 8

- ▶ Supplier<T>
- ▶ Consumer<T>
- ▶ Predicate<T>
- ▶ Function<T,R>

INTERFACES FUNCIONALES EN JDK 8

- ▶ Supplier<T>
- ▶ Consumer<T>
- ▶ Predicate<T>
- ▶ Function<T,R>
- ▶ UnaryOperator<T>

INTERFACES FUNCIONALES EN JDK 8

- ▶ Supplier<T>
- ▶ Consumer<T>
- ▶ Predicate<T>
- ▶ Function<T,R>
- ▶ UnaryOperator<T>
- ▶ BinaryOperator<T>

INTERFACES FUNCIONALES EN JDK 8

- ▶ Supplier<T>
- ▶ Consumer<T>
- ▶ Predicate<T>
- ▶ Function<T,R>
- ▶ UnaryOperator<T>
- ▶ BinaryOperator<T>

java.util.function

Supplier<T>

Supplier<T>

- ▶ Representa a un proveedor de objetos tipo T

Supplier<T>

- ▶ Representa a un proveedor de objetos tipo T
- ▶ La creación del objeto T se realiza sólo hasta el momento de utilizarse

Supplier<T>

- ▶ Representa a un proveedor de objetos tipo T
- ▶ La creación del objeto T se realiza sólo hasta el momento de utilizarse
- ▶ Su método funcional es Supplier.get()

Supplier<T>

- ▶ Representa a un proveedor de objetos tipo T
- ▶ La creación del objeto T se realiza sólo hasta el momento de utilizarse
- ▶ Su método funcional es Supplier.get()

```
logger.fine("Parámetro de búsqueda: "+parametro);
logger.fine(()->"Parámetro de búsqueda: "+parametro);

public void printOut(Supplier<String> msgSup){
    if (SEND_TO_OUT)
        System.out.println(msgSup.get());
}
```

Consumer<T>

Consumer<T>

- ▶ Representa una operación que acepta un único parámetro de tipo T

Consumer<T>

- ▶ Representa una operación que acepta un único parámetro de tipo T
- ▶ No regresa ningún resultado

Consumer<T>

- ▶ Representa una operación que acepta un único parámetro de tipo T
- ▶ No regresa ningún resultado
- ▶ Se considera como una acción que produce un efecto colateral

Consumer<T>

- ▶ Representa una operación que acepta un único parámetro de tipo T
- ▶ No regresa ningún resultado
- ▶ Se considera como una acción que produce un efecto colateral
- ▶ Su método funcional es Consumer.accept(Object)

Consumer<T>

- ▶ Representa una operación que acepta un único parámetro de tipo T
- ▶ No regresa ningún resultado
- ▶ Se considera como una acción que produce un efecto colateral
- ▶ Su método funcional es Consumer.accept(Object)

```
nombres.forEach(s -> System.out.println(s));
```

Predicate<T>

Predicate<T>

- ▶ Representa un predicado booleano de un argumento

Predicate<T>

- ▶ Representa un predicado booleano de un argumento
- ▶ Su método funcional es Predicate.test(Object)

Predicate<T>

- ▶ Representa un predicado booleano de un argumento
- ▶ Su método funcional es Predicate.test(Object)

```
Stream<User> mayores = personas.stream().filter(  
    p -> p.getAge()>18);
```

Function<T,R>

Function<T,R>

- ▶ Representa una función de transformación de T a R

Function<T,R>

- ▶ Representa una función de transformación de T a R
- ▶ Regresa un objeto de tipo R

Function<T,R>

- ▶ Representa una función de transformación de T a R
- ▶ Regresa un objeto de tipo R
- ▶ Su método funcional es Function.apply(Object)

Function<T,R>

- ▶ Representa una función de transformación de T a R
- ▶ Regresa un objeto de tipo R
- ▶ Su método funcional es Function.apply(Object)

```
Stream<Integer> longitudes = personas
    .stream()
    .map(u -> u.getName() == null
        ? 0 : u.getName().length());
```

UnaryOperator<T>

UnaryOperator<T>

- ▶ Representa una función de transformación de T a T

UnaryOperator<T>

- ▶ Representa una función de transformación de T a T
- ▶ Su método funcional es UnaryOperator.apply(Object)

BinaryOperator<T>

BinaryOperator<T>

- ▶ Representa una función de Transformación de T, T a T

BinaryOperator<T>

- ▶ Representa una función de Transformación de T, T a T
- ▶ Su método funcional es BinaryOperator.apply(Object, Object)

METHOD REFERENCES

```
Consumer<String> consumer = s -> System.out.println(s);  
Consumer<String> consumer = System.out::println;
```

METHOD REFERENCES

- ▶ En algunas ocasiones sólo usamos una Lambda para llamar un método existente

```
Consumer<String> consumer = s -> System.out.println(s);
```

```
Consumer<String> consumer = System.out::println;
```

METHOD REFERENCES

- ▶ En algunas ocasiones sólo usamos una Lambda para llamar un método existente

```
Consumer<String> consumer = s -> System.out.println(s);
```

```
Consumer<String> consumer = System.out::println;
```

- ▶ Referencias a Métodos son una forma simple y fácil de leer para implementar expresiones lambda con métodos que ya tienen nombre

TIPOS DE METHOD REFERENCES

TIPOS DE METHOD REFERENCES

- ▶ Referencia a un método estático -
`ContainingClass::staticMethodName`

TIPOS DE METHOD REFERENCES

- ▶ Referencia a un método estático -
ContainingClass::staticMethodName

- ▶ Referencia a un método de instancia de un objeto específico -
containingObject::instanceMethodName

TIPOS DE METHOD REFERENCES

- ▶ Referencia a un método estático -
ContainingClass::staticMethodName
- ▶ Referencia a un método de instancia de un objeto específico -
containingObject::instanceMethodName
- ▶ Referencia a un método de instancia de cualquier objeto de un tipo definido - **ContainingType::methodName**

TIPOS DE METHOD REFERENCES

- ▶ Referencia a un método estático -
ContainingClass::staticMethodName
- ▶ Referencia a un método de instancia de un objeto específico -
containingObject::instanceMethodName
- ▶ Referencia a un método de instancia de cualquier objeto de un tipo definido - **ContainingType::methodName**
- ▶ Referencia a un constructor - **ClassName::new**

Optional

Optional

- ▶ Contenedor de un objeto que puede o no ser null

Optional

- ▶ Contenedor de un objeto que puede o no ser null
- ▶ Si contiene un valor, `isPresent()` regresa true y `get()` el valor

Optional

- ▶ Contenedor de un objeto que puede o no ser null
- ▶ Si contiene un valor, `isPresent()` regresa true y `get()` el valor
- ▶ Es inmutable y tiene métodos para caso como:
`orElse(T)`
`orElseGet(Supplier<? extends T> other)`
`orElseThrow(Supplier<? extends X> exceptionSupplier)`

CASO DE EJEMPLO: CALCULAR EL PROMEDIO DE EDAD DE LOS ELEGIBLES A BEBER

CASO DE EJEMPLO: CALCULAR EL PROMEDIO DE EDAD DE LOS ELEGIBLES A BEBER

```
... List<Person> list ...
double ageAverageElegibleDrivers() {
    double sum = 0;
    int cont = 0;
    for (Person p : list) {
        if (p.getAge()>=21){
            sun += p.getAge();
            cont++;
        }
    }
    return sum/cont;
}
```

APLICANDO STREAMS

APLICANDO STREAMS

```
... List<Person> list ...
double ageAverageElegibleDrivers(){
    return list.stream()
        .filter(p -> p.age>21)
        .mapToInt(p->p.getAge())
        .average()
        .orElse(0.0);
}
```

STREAM API

STREAM API

- ▶ Diseñado para trabajar principalmente con colecciones

STREAM API

- ▶ Diseñado para trabajar principalmente con colecciones
- ▶ Provee de una manera "declarativa" de procesar una colección

STREAM API

- ▶ Diseñado para trabajar principalmente con colecciones
- ▶ Provee de una manera "declarativa" de procesar una colección
- ▶ Se puede ver como un Iterador elegante de una colección

STREAM API

- ▶ Diseñado para trabajar principalmente con colecciones
- ▶ Provee de una manera "declarativa" de procesar una colección
- ▶ Se puede ver como un Iterador elegante de una colección
- ▶ Facilita la paralelización del procesamiento de la información

¿QUÉ ES UN STREAM?

¿QUÉ ES UN STREAM?

- ▶ Secuencia de elementos

¿QUÉ ES UN STREAM?

- ▶ Secuencia de elementos
- ▶ No almacena valores, los genera bajo demanda

¿QUÉ ES UN STREAM?

- ▶ Secuencia de elementos
 - ▶ No almacena valores, los genera bajo demanda
- ▶ Fuente de datos

¿QUÉ ES UN STREAM?

- ▶ Secuencia de elementos
 - ▶ No almacena valores, los genera bajo demanda
- ▶ Fuente de datos
 - ▶ Consumen desde una fuente de datos como Colecciones, Arreglos e I/O

¿QUÉ ES UN STREAM? (2)

¿QUÉ ES UN STREAM? (2)

- ▶ Operaciones de Agregación

¿QUÉ ES UN STREAM? (2)

- ▶ Operaciones de Agregación
- ▶ Operaciones estilo base de datos como: filter, map, reduce, findFirst, allMatch, sorted, ...

¿QUÉ ES UN STREAM? (2)

- ▶ Operaciones de Agregación
 - ▶ Operaciones estilo base de datos como: filter, map, reduce, findFirst, allMatch, sorted, ...
- ▶ Encadenamiento

¿QUÉ ES UN STREAM? (2)

- ▶ Operaciones de Agregación
 - ▶ Operaciones estilo base de datos como: filter, map, reduce, findFirst, allMatch, sorted, ...
- ▶ Encadenamiento
 - ▶ La mayoría de las operaciones regresan un stream, por lo que se pueden encadenar. El procesamiento se hace sólo al alcanzar una operación terminal

¿QUÉ ES UN STREAM? (3)

¿QUÉ ES UN STREAM? (3)

- ▶ Iteración Interna

¿QUÉ ES UN STREAM? (3)

- ▶ Iteración Interna
- ▶ La iteración es decidida internamente y reflejará la forma en que se encadenaron las operaciones

STREM VS COLLECTION

STREAM VS COLLECTION

- ▶ Sin Almacenamiento

STREAM VS COLLECTION

- ▶ Sin Almacenamiento
- ▶ Funcionales

STREAM VS COLLECTION

- ▶ Sin Almacenamiento
- ▶ Funcionales
- ▶ Flojas

STREAM VS COLLECTION

- ▶ Sin Almacenamiento
- ▶ Funcionales
- ▶ Flojas
- ▶ Posiblemente Ilimitadas

STREAM VS COLLECTION

- ▶ Sin Almacenamiento
- ▶ Funcionales
- ▶ Flojas
- ▶ Posiblemente Ilimitadas
 - ▶ `limit(n)` `findFirst()`

STREAM VS COLLECTION

- ▶ Sin Almacenamiento
- ▶ Funcionales
- ▶ Flojas
- ▶ Posiblemente Ilimitadas
 - ▶ `limit(n)` `findFirst()`
- ▶ Consumibles

PIPELINE

PIPELINE

- ▶ Fuente de datos

PIPELINE

- ▶ Fuente de datos
- ▶ una o varias operaciones Intermedias

PIPELINE

- ▶ Fuente de datos
- ▶ una o varias operaciones Intermedias
 - ▶ Son operaciones que regresan otro stream, por lo que se pueden encadenar, no se ejecutan sino hasta que el stream tenga una operación terminal y deberían utilizar funciones sin efectos colaterales

PIPELINE

- ▶ Fuente de datos
- ▶ una o varias operaciones Intermedias
 - ▶ Son operaciones que regresan otro stream, por lo que se pueden encadenar, no se ejecutan sino hasta que el stream tenga una operación terminal y deberían utilizar funciones sin efectos colaterales
- ▶ una operación Terminal

PIPELINE

- ▶ Fuente de datos
- ▶ una o varias operaciones Intermedias
 - ▶ Son operaciones que regresan otro stream, por lo que se pueden encadenar, no se ejecutan sino hasta que el stream tenga una operación terminal y deberían utilizar funciones sin efectos colaterales
- ▶ una operación Terminal
 - ▶ Son operaciones que "Terminan" un stream, usualmente tienen efectos colaterales e inician el procesamiento del stream y producen un resultado

¿DE DÓNDE OBTENEMOS LOS STREAMS? (FUENTES)

- ▶ De una Colección usando los métodos `stream()` y `parallelStream()`
- ▶ De un arreglo `Arrays.stream(Object[])`
- ▶ De las clases stream `Stream.of(Object[])`,
`IntStream.range(int, int)`
- ▶ De un archivo `BufferedReader.lines()`
- ▶ De directorios, patrones, números aleatorios y/o creadas por nosotros

APLICANDO STREAMS

APLICANDO STREAMS

```
... List<Person> list ...
double ageAverageElegibleDrivers(){
    return list.stream()
        .filter(p -> p.age>21)
        .mapToInt(p->p.getAge())
        .average()
        .orElse(0.0);
}
```

APLICANDO STREAMS

```
... List<Person> list ...
double ageAverageElegibleDrivers(){
    return list.stream()
        .filter(p -> p.age>21)
        .mapToInt(p->p.getAge())
        .average()
        .orElse(0.0);
}
```

fuente

intermedias

terminal

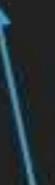
```
graph TD
    A[fuente] --> B[list]
    B --> C[stream()]
    C --> D[filter(p -> p.age > 21)]
    D --> E[mapToInt(p -> p.getAge())]
    E --> F[average()]
    F --> G[orElse(0.0)]
```

LOS STREAMS SON OBJETOS

```
int highScore = students.stream()  
    .filter(s -> s.graduationYear() == 2015)  
    .map(s -> s.getScore())  
    .max();
```

LOS STREAMS SON OBJETOS

```
int highScore = students.stream()  
    .filter(s -> s.graduationYear() == 2015)  
    .map(s -> s.getScore())  
    .max();
```



regresa un int

LOS STREAMS SON OBJETOS

```
int highScore = students.stream()  
    .filter(s -> s.graduationYear() == 2015)  
    .map(s -> s.getScore())  
    .max();
```

regresa un int

map debe envolver el int
en un Integer

LOS STREAMS SON OBJETOS

```
int highScore = students.stream()  
    .filter(s -> s.graduationYear() == 2015)  
    .map(s -> s.getScore())  
    .max();
```

↑
max extrae cada Integer
para obtener el mayor

regresa un int

map debe envolver el int
en un Integer

STREAMS DE PRIMITIVOS

```
int highScore = students.stream()
    .filter(s -> s.graduationYear() == 2015)
    .mapToInt(s -> s.getScore())
    .max();
```

STREAMS DE PRIMITIVOS

```
int highScore = students.stream()
    .filter(s -> s.graduationYear() == 2015)
    .mapToInt(s -> s.getScore())
    .max();
```

mapToInt produce un stream de int,
así ya no hay boxing o unboxing

STREAMS DE PRIMITIVOS

```
int highScore = students.stream()
    .filter(s -> s.graduationYear() == 2015)
    .mapToInt(s -> s.getScore())
    .max();
```

mapToInt produce un stream de int,
así ya no hay boxing o unboxing

- ▶ Para mejorar la eficiencia existen tres tipos de stream primitivos

STREAMS DE PRIMITIVOS

```
int highScore = students.stream()
    .filter(s -> s.graduationYear() == 2015)
    .mapToInt(s -> s.getScore())
    .max();
```

mapToInt produce un stream de int,
así ya no hay boxing o unboxing

- ▶ Para mejorar la eficiencia existen tres tipos de stream primitivos
- ▶ IntStream, DoubleStream y LongStream

STREAMS DE PRIMITIVOS

```
int highScore = students.stream()
    .filter(s -> s.graduationYear() == 2015)
    .mapToInt(s -> s.getScore())
    .max();
```

mapToInt produce un stream de int,
así ya no hay boxing o unboxing

- ▶ Para mejorar la eficiencia existen tres tipos de stream primitivos
 - ▶ IntStream, DoubleStream y LongStream
- ▶ Usar métodos como mapToInt(), mapToDouble() y mapToLong()

OPERACIONES INTERMEDIAS - FUENTES

- ▶ **generate(Supplier<T> s)**
 - ▶ Genera una secuencia infinita y desordenada de los valores producidos por **Supplier**
- ▶ **of(T t) of(T... values)**
 - ▶ Genera un stream con un solo elemento **T** o una secuencia de valores **T**
- ▶ **iterate(T seed, UnaryOperator<T> f)**
 - ▶ Genera una secuencia infinita producida de aplicar iterativamente la función **f(seed), f(f(seed))...**

OPERACIONES INTERMEDIAS - FILTRADO

- ▶ `filter(Predicate<? super T> predicate)`
 - ▶ Regresa un stream que contiene sólo los elementos que corresponden tras aplicar el **Predicate** proporcionado
- ▶ `limit(long maxSize)`
 - ▶ Regresa un stream truncado con a lo mucho **maxSize** elementos
- ▶ `skip(long n)`
 - ▶ Regresa un stream con los elementos sobrantes tras descartar los primeros **n** elementos

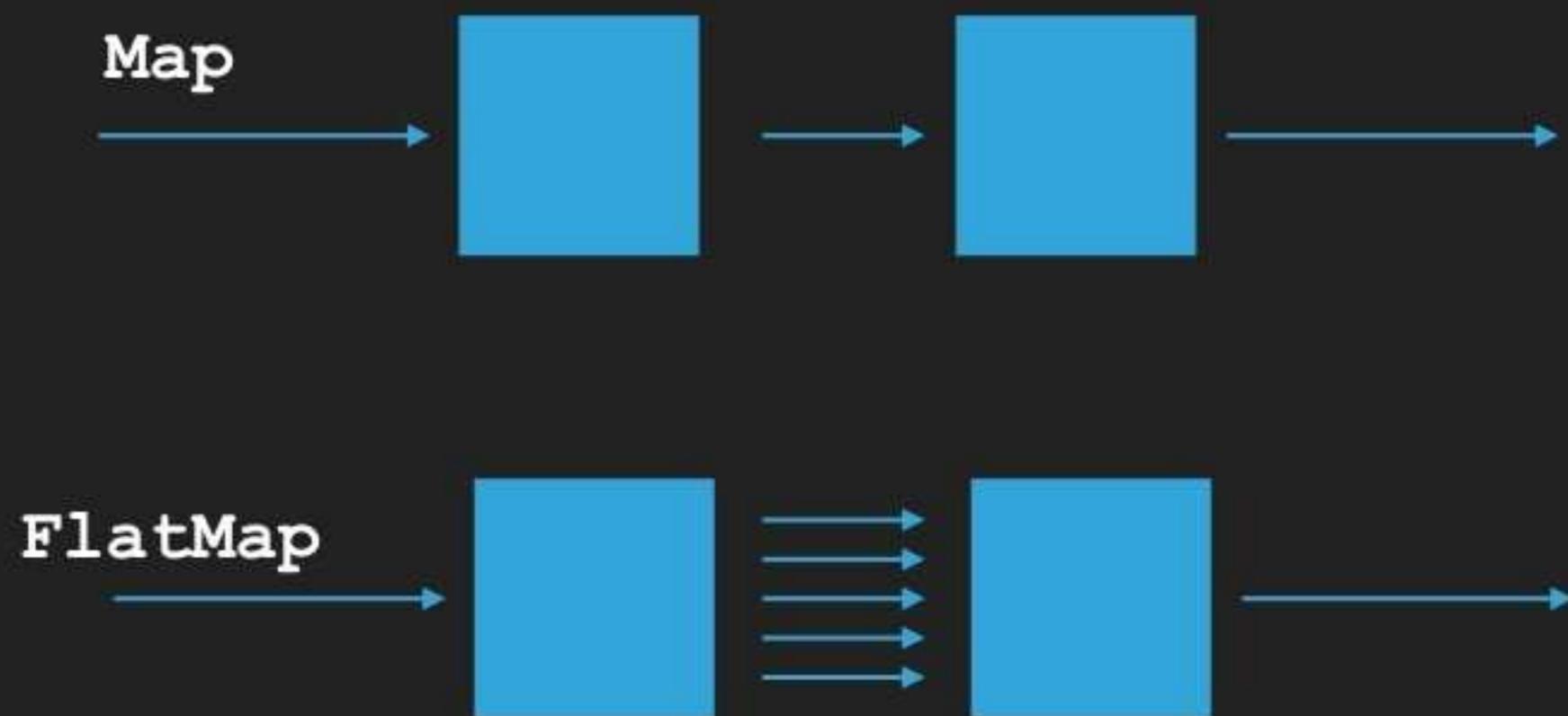
OPERACIONES INTERMEDIAS - ORDEN

- ▶ **distinct()**
 - ▶ Regresa un stream que contiene sólo los elementos que son diferentes aplicando un `Object.equals(Object)`
- ▶ **sorted()**
 - ▶ Regresa un stream ordenado según el orden natural de los objetos contenidos
- ▶ **sorted(Comparator<? super T> comparator)**
 - ▶ Regresa un stream ordenado según el `Comparator` proporcionado

OPERACIONES INTERMEDIAS - MAPEO

- ▶ **map(Function<? super T, ? extends R> mapper)**
 - ▶ Regresa un stream que contiene los resultados de aplicar la función proporcionada a cada elemento
- ▶ **flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)**
 - ▶ Regresa un stream que reemplaza los elementos del stream actual por los elementos del stream mapeado
- ▶ **orders.flatMap(order -> order.getLineItems().stream())**

OPERACIONES INTERMEDIAS - MAPEO



OPERACIONES TERMINALES - MATCH

- ▶ **allMatch (Predicate<? super T> predicate)**
 - ▶ true si todos los elementos pasan la prueba del **Predicate**
- ▶ **anyMatch (Predicate<? super T> predicate)**
 - ▶ true si alguno de los elementos pasa la prueba del **Predicate**
- ▶ **noneMatch (Predicate<? super T> predicate)**
 - ▶ true si ninguno de los elementos pasa la prueba del **Predicate**

OPERACIONES TERMINALES - NUMÉRICOS

- ▶ **max(Comparator<? super T> comparator)**
 - ▶ Optional del elemento mayor según comparator
- ▶ **min(Comparator<? super T> comparator)**
 - ▶ Optional del elemento menor según comparator
- ▶ **count()**
 - ▶ Cantidad de elementos en el stream

OPERACIONES TERMINALES - MÁS NUMÉRICOS

- ▶ Para los Streams de Primitivos IntStream, LongStream, DoubleStream
- ▶ **average ()**
 - ▶ Obtiene el promedio de los elementos
 - ▶ Regresa un Optional pues la lista puede estar vacía
- ▶ **sum ()**
 - ▶ Obtiene la suma de los elementos

OPERACIONES TERMINALES - REDUCTORES

- ▶ **reduce (BinaryOperator<T> accumulator)**
 - ▶ Reducción de elementos de este stream usando una función de acumulación asociativa
 - ▶ Este acumulador toma un resultado parcial y el siguiente elemento y regresa un resultado parcial
 - ▶ Regresa un Optional

REDUCTORES

- ▶ Encontrar la longitud de la línea mas larga de un archivo

```
Path input = Paths.get("lines.txt");

int longestLineLength = Files.lines(input)
    .mapToInt(String::length)
    .max()
    .getAsInt();
```

REDUCTORES

- ▶ Encontrar la longitud de la línea mas larga de un archivo

REDUCTORES

- ▶ Encontrar la longitud de la línea mas larga de un archivo

```
String longest = Files.lines(input)
    .sort((x, y) -> y.length() - x.length())
    .findFirst()
    .get();
```

REDUCTORES

- ▶ Encontrar la longitud de la línea mas larga de un archivo

```
String longest = Files.lines(input)
    .sort((x, y) -> y.length() - x.length())
    .findFirst()
    .get();
```

¿Y si el archivo es muy grande?

REDUCTORES - ALTERNATIVAS

- ▶ Usar un while y una variable que contenga el resultado
 - ▶ simple, pero no es thread safe
- ▶ Usar recursión
 - ▶ No tiene una variable mutable, por lo que es thread safe
 - ▶ En un archivo largo podemos tener un stack overflow
- ▶ Usar reduce(BinaryOperator<T> accumulator)
 - ▶ El accumulator obtiene un resultado parcial y el siguiente elemento
 - ▶ Como con recursión, pero sin el stack overflow

REDUCTORES

- ▶ Encontrar la línea mas larga de un archivo

```
String longestLine = Files.lines(input)
    .reduce((x, y) -> {
        if (x.length() > y.length())
            return x;
        return y; })
    .get();
```

REDUCTORES

- ▶ Encontrar la línea mas larga de un archivo

```
String longestLine = Files.lines(input)
    .reduce((x, y) -> {
        if (x.length() > y.length())
            return x;
        return y; })
    .get();
```

La x mantiene el estado,
en este caso guarda la linea más larga

OPERACIONES TERMINALES - COLECTORES

- ▶ **collect(Collector<? super T,A,R> collector)**
 - ▶ Realiza una reducción mutable de los elementos del stream
 - ▶ El Contenedor del resultado puede ser un List, Map, String, etc
- ▶ **toArray()**
 - ▶ Regresa un Array con los elementos del Stream

COLLECTORS

- ▶ La clase `Collectors` implementa muchos colectores como:
- ▶ `toList()`
 - ▶ `List<String> list = people.stream()
 .map(Person::getName)
 .collect(Collectors.toList());`
- ▶ `toCollection(Supplier<C> collectionFactory)`
 - ▶ `Set<String> set = people.stream()
 .map(Person::getName).collect(
 Collectors.toCollection(TreeSet::new));`

COLLECTORS - A UN MAPA

- ▶ `toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper)`
- ▶ **Map<String, Student> studentIdToStudent**
`students.stream()
.collect(toMap(Student::getId,
Functions.identity()));`
- ▶ `toMap(Function<? super T,? extends K> keyMapper, Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction)`
- ▶ **Map<String, String> phoneBook =**
`people.stream().collect(toMap(Person::getName,
Person::getAddress, (s, a) -> s + ", " + a));`
- ▶ Para mapas concurrentes usar `toConcurrentMap(...)`

COLLECTORS - AGRUPACIONES

- ▶ `groupingBy(Function<? super T,? extends K> classifier)`
- ▶ `Map<Department, List<Employee>> byDept = employees.stream().collect(Collectors.groupingBy(Employee::getDepartment));`
- ▶ `groupingBy(Function<? super T,? extends K> classifier, Collector<? super T,A,D> downstream)`
- ▶ `Map<City, Set<String>> namesByCity = people.stream().collect(groupingBy(Person::getCity, mapping(Person::getLastName, toSet())));`
- ▶ Para agrupaciones concurrentes usar `groupingByConcurrent(...)`

COLLECTORS - UNIENDO CADENAS

- ▶ Otra forma de colecciónar resultados es concatenarlos en cadenas
 - ▶ `joining()`
 - ▶ `joining(CharSequence delimiter)`
 - ▶ `joining(CharSequence delimiter, CharSequence prefix, CharSequence suffix)`

```
String joined = things.stream()  
    .map(Object::toString)  
    .collect(Collectors.joining(", "));
```

OPERACIONES TERMINALES - ITERACIÓN

- ▶ Stream cuenta con métodos para iterar el stream resultante, pero dependiendo de lo que se esté haciendo muy probablemente alguno de los colectores anteriores lo pueda realizar
- ▶ **forEach(Consumer<? super T> action)**
 - ▶ Realiza la action para cada elemento en el stream
- ▶ **forEachOrdered(Consumer<? super T> action)**
 - ▶ Realiza la action para cada elemento en el stream garantizando el orden si se ejecuta en paralelo

PROCESAMIENTO EN PARALELO

- ▶ Una "ventaja" del uso de streams es su facilidad para convertirlas a procesos paralelos
- ▶ Para convertir un stream a procesamiento en paralelo sólo hay que agregar parallel() al pipeline o usar parallelStream() al generar el stream
- ▶ Basado en el framework Fork-Join
- ▶ Considerarlo si, y sólo si, las operaciones intermedias o terminales a realizar son computacionalmente pesadas y se tienen miles de elementos

EN RESUMEN

- ▶ En Java 8 contamos con nuevas herramientas
 - ▶ Lambdas - Funciones independientes para realizar tareas muy específicas
 - ▶ Optional - Forma de manejar la incertidumbre
 - ▶ Streams - Forma de procesar colecciones o secuencias de valores