

# Java 1.8.

¿Puede la programación orientada a objetos ser funcional?

Jose Divasón, Jesús Aransay

Seminario de Informática Mirian Andrés

Universidad de La Rioja

21/03/2017

# Motivación

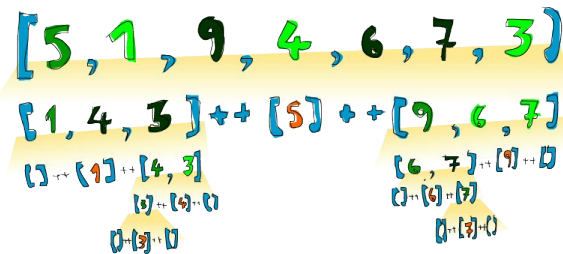
Algunas ventajas de la programación funcional:

- Currificación
- Funciones como parámetros
- Clases de tipos
- Evaluación perezosa
- Sin efectos laterales

# Motivación:

```
int divide(int *array, int start, int end) {
    int left;
    int right;
    int pivot;
    int temp;
    pivot = array[start];
    left = start;
    right = end;
    while (left < right) {
        while (array[right] > pivot) {
            right--;
        }
        while ((left < right) && (array[left] <= pivot)) {
            left++;
        }
        if (left < right) {
            temp = array[left];
            array[left] = array[right];
            array[right] = temp;
        }
    }
    temp = array[right];
    array[right] = array[start];
    array[start] = temp;
    return right;}
}
```

```
void quicksort(int *array, int start, int end)
{
    int pivot;
    if (start < end) {
        pivot = divide(array, start, end);
        quicksort(array, start, pivot - 1);
        quicksort(array, pivot + 1, end);
    }
}
```



quicksort :: (Ord a) => [a] -> [a]

quicksort [] = []

quicksort (p:xs) =

quicksort [a | a <- xs, a <= p] ++ [p] ++ quicksort [a | a <- xs, a > p]

# Motivación

Java 1.8 permite escribir código como:

```
// Toma dos parámetros enteros y devuelve su suma  
(int x, int y) -> x + y
```

```
// Toma dos parámetros enteros y devuelve su suma  
(x, y) -> x + y
```

```
// Toma un parámetro y lo muestra por la salida estándar  
msg -> System.out.println(msg)
```

# Motivación

Java 1.8 permite escribir código como:

```
List numbers = Arrays.asList(1, 2, 3, 4, 5);
```

```
int sum = numbers.stream()
```

```
    .filter(n -> n % 2 == 1)
```

```
    .map(n -> n * n)
```

```
    .reduce(0, Integer::sum);
```

# Motivación

¿Qué elementos de la programación funcional “tradicional” pueden ser utilizados en Java 1.8?

¿Cómo han sido encajados dentro del paradigma de Orientación a Objetos que implementa Java?

# Índice

- Java lambdas
- Java Streams
- Conclusiones

# Java lambdas

¿Cuál es el tipo de las siguientes expresiones en Java?

`(String str) -> str.length()`

`person -> (if (person.age > 40) then return “Very old man...”;`

`else return “enjoy the moment”;`)



# Java lambdas

En el caso de la primera expresión ((String str) -> str.length()) podría ser...

```
@FunctionalInterface
```

```
interface StringToIntMapper {
```

```
    int map (String str);
```

```
}
```

# Java lambdas

¿Y en el caso de la segunda expresión? Las cosas se complican...porque no hay tipado explícito

Para poder entenderlo mejor, debemos distinguir en Java dos tipos de expresiones:

- “Standalone expressions”
- “Poly expressions”

# Java lambdas: “standalone expressions”

Las siguientes expresiones en Java son ejemplos de “standalone expressions”:

```
new String (“Hello”)
```

```
“Hello”
```

```
new ArrayList<String>();
```

Podemos conocer su tipo (también el compilador puede) sin necesidad de contexto

# Java lambdas: “poly expressions”

Los siguientes son ejemplos de “poly expressions”:

```
new ArrayList<>();
```

La expresión tendrá distintos tipos, dependiendo del contexto en que la encontremos:

```
List<Long> idList = new ArrayList<>();
```

```
List<String> nameList = new ArrayList<>();
```

En el primer contexto tiene tipo “ArrayList<Long>”, en el segundo tiene tipo “ArrayList<String>”

# Java lambdas: “poly expressions”

De hecho, todas las expresiones lambda en Java son “poly expressions”

Su tipo será inferido siempre en un contexto determinado (aunque no sea necesario, como en el ejemplo “(String str) -> str.length()”)

De este modo, la expresión “(x, y) -> x + y” es una “poly expression” cuyo tipo será, dependiendo del contexto:

- `@FunctionalInterface interface IntIntToIntMapper { int map(int, int); }`
- `@FunctionalInterface interface DoubleDoubleToDoubleMapper { double map(double, double); }`
- `@FunctionalInterface interface StringStringToStringMapper { String map(String, String); }`

# Java lambdas

Las expresiones lambdas son una “versión reducida” de las clases anónimas

En realidad, se pueden describir como “azúcar sintáctico” para clases anónimas que

- no tengan estado y
- contengan un solo método

# Java lambdas

// Clase anónima (sin estado y con un solo método)

```
class StringToIntMapper mapper = new StringToIntMapper() {  
  
    @Override  
  
    public int map(String str) {  
  
        return str.length();  
  
    }  
  
};
```

// Expresión lambda

```
StringToIntMapper mapper = (String str) -> str.length();
```

# Java lambdas: ¿y si omitimos los tipos?

// Los tipos de los parámetros son declarados

```
(int x, int y) -> { return x + y; }
```

// Los tipos de los parámetros son omitidos

```
(x, y) -> { return x + y; }
```

// Error de compilación

```
(int x, y) -> { return x + y; }
```

Conclusión: o tipamos explícitamente todo (menos el tipo devuelto), o no tipamos nada



# Java lambdas: más azúcar sintáctico

Distintas opciones de uso y declaración:

```
// Declara el parámetro de tipo
```

```
(String msg) -> { System.out.println(msg); }
```

```
// Omite el parámetro de tipo
```

```
(msg) -> { System.out.println(msg); }
```

```
// Omite el parámetro de tipo y los paréntesis
```

```
msg -> { System.out.println(msg); }
```

# Java lambdas: más azúcar sintáctico

Otras formas válidas de definir expresiones lambda:

```
(final int x, final int y) -> { return x + y; }
```

```
(int x, final int y) -> { return x + y; }
```

// forma no válida

```
(final x, final y) -> { return x + y; }
```

# Java lambdas: más azúcar sintáctico

El cuerpo de una expresión lambda puede ser:

- un bloque

```
(int x, int y) -> { return x + y; }
```

- una expresión; en ese caso, la expresión se evalúa y es el valor de retorno

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

# Java lambdas: más azúcar sintáctico

También podemos trabajar con expresiones más elaboradas a través de los bloques

```
(Person p) -> p.getAge() > 21 && p.getWeight() > 100 && "Chris".equals(p.getName())
```

```
(oldState, newState) -> {  
    System.out.println("Old state: " + oldState);  
    System.out.println("New state: " + newState);  
}
```

# Java lambdas: ¿y los tipos...?

Tal y como hemos dicho, las expresiones lambda son instancias de interfaces funcionales. ¿Interfaces qué...?

Una interfaz funcional es simplemente una interfaz que tiene exactamente un método abstracto (los métodos default, static y los públicos heredados de Object no cuentan)

Java contiene en su librería múltiples casos de SAM (Single Abstract Method) interfaces:

```
@FunctionalInterface
interface Comparator<T>
{
    int compare (T o1, T o2);
}
```

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

# Java lambdas: ¿y los tipos...?

Recordatorio de Prog. III:

¿Qué métodos pueden ser llamados sobre el siguiente objeto? (EstudianteComparator es una clase que implementa Comparator)

```
Comparator c = new EstudianteComparator ();
```

```
Estudiante e1 = new Estudiante(...), e2 = new Estudiante(...);
```

```
c.¿?;
```

# Java lambdas: ¿y los tipos...?

Una interfaz funcional por tanto contendrá el método propio de su interfaz, así como copias abstractas de todos los métodos de la clase “Object”

Una expresión lambda puede ser usada en cualquier sitio que se espera una instancia de una clase “Single Abstract Method” (ni siquiera de una @FunctionalInterface, que es un tipo particular de SAM)

```
Runnable r1 = () -> System.out.println("My Runnable");
```

Nota: en este caso, Runnable sí es una @FunctionalInterface

# Java lambdas: ¿y los tipos...?

Entonces, ¿para qué emplea Java la anotación `@FunctionalInterface`?

Las interfaces SAM que llevan la anotación `@FunctionalInterface` son las que Java espera o recomienda que implementes por medio de lambdas



# Java lambdas: ¿y los tipos...?

Contraejemplo:

```
interface Comparable<T>{  
    int compareTo (T o); // Compara “this” con el parámetro “o”  
}
```

```
Comparable<Person> c1 = (Person p) -> return 0;
```

```
// Compila, no es “peligroso”, pero probablemente no tiene mucho sentido...
```

```
// Generalmente necesitaremos usar “this” para comparar
```

# Java lambdas: ¿y los tipos...?

Solución natural (aunque lo anterior compilaba): usar “Comparator<T>”

```
Comparator<Human> ch = (Human h1, Human h2)
```

```
-> h1.getName().compareTo(h2.getName());
```

# Java lambdas: target typing

Para que una expresión lambda sea asignable a una interfaz T:

1. T *debería* ser una interfaz funcional
2. La expresión lambda tiene que tener el mismo número de parámetros que el método abstracto de T y los tipos deben ser compatibles (en caso necesario, siempre que sea posible el compilador podrá inferir los tipos a partir del método abstracto).
3. El tipo del valor de retorno de la lambda debe ser compatible con el del método abstracto de T.
4. Si el cuerpo de la lambda lanza (throw) alguna excepción, estas deben ser compatibles con las que lanza el método abstracto de T.

# Java lambdas: ¿y los tipos...?

Si tenemos estas dos interfaces funcionales:

```
@FunctionalInterface
public interface Adder {
    double add(double n1, double n2);
}
```

```
@FunctionalInterface
public interface Joiner {
    String join(String s1, String s2);
}
```

`(x, y) -> x + y;` **NO**

`Adder adder = (x, y) -> x + y;` **OK**

`Adder adder = (x, y) -> x - y;` **OK**

`Adder adder = (double x, double y) -> x + y;` **OK**



`Joiner joiner = (x, y) -> x + y;` **OK**


# Java lambdas: ¿y los tipos...?


```
public class Ejemplo {  
  
    public void test(Adder adder)  
    {  
        double x = 190.90;  
        double y = 8.50;  
        double sum = adder.add(x, y);  
        System.out.println(x + " + " + y + " =  
        " + sum);  
    }  
  
    public void test(Joiner joiner)  
    {  
        String s1 = "Hello";  
        String s2 = "World";  
        String s3 = joiner.join(s1,s2);  
        System.out.println("\\" + s1 + "\" + \"\\" + \"\\"  
        + s2 + "\" = \"\\" + s3 + "\"");  
    }  
}
```


Creamos un objeto de Ejemplo:  
Ejemplo util = new Ejemplo();

¿Y ahora, esto compila?

util.test((double x, double y) -> x + y);   
util.test((double x, double y) -> x - y); 

util.test((Adder)(x, y) -> x + y); 

Adder adder = (x, y) -> x + y;  
util.test(adder); 

util.test((x, y) -> x + y); 

# Java lambdas: interfaces funcionales

Nombre de la Interfaz	Método	Descripción
Function<T,R>	R apply(T t)	Representa una función que coge un parámetro de tipo T y devuelve un resultado de tipo R
BiFunction<T,U,R>	R apply(T t, U u)	Representa una función que coge dos parámetros de tipos T y U, y devuelve un resultado de tipo R
Predicate<T>	boolean test(T t)	Dado un parámetro de tipo T, comprueba algo y devuelve un booleano
BiPredicate<T,U>	boolean test(T t, U u)	Predicado de dos parámetros
Consumer<T>	void accept(T t)	Representa una operación que toma un parámetro, opera con él para hacer algo y no devuelve un resultado
BiConsumer<T,U>	void accept(T t, U u)	Análogo al anterior con dos parámetros
Supplier<T>	T get()	Devuelve un elemento de tipo T
UnaryOperator<T>	T apply(T t)	Hereda de Function<T,T>. Representa una función que coge un parámetro y devuelve un resultado del mismo tipo
BinaryOperator<T>	T apply(T t1, T t2)	Hereda de BiFunction<T,T,T>. Representa una función que coge dos parámetros del mismo tipo y devuelve un resultado del mismo tipo también

Lista completa: <https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html>

# Java lambdas: ejemplos

```
// Function square = x -> x*x; //NO SE PUEDE
```

```
// Las siguientes tres son equivalentes:
```

```
Function<Integer,Integer> square = x -> x*x;
```

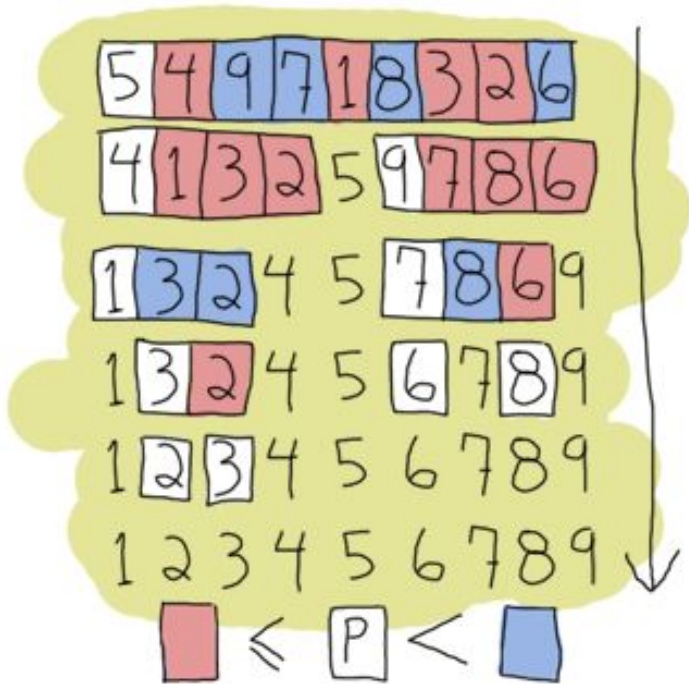
```
UnaryOperator<Integer> square2 = x -> x*x;
```

```
IntFunction square3 = x -> x*x;
```

```
System.out.println(square.apply(4));
```

# Java lambdas: ¿Funciones como parámetros?

Ejercicio: ordenar crecientemente un vector de enteros.



¿Y si ahora queremos ordenar un vector de reales?

¿Y ordenarlo decrecientemente?

¿Y ordenarlo alfabéticamente?



# Java lambdas: ¿Funciones como parámetros?

1: Crear una lambda que instancia de la interface Comparator sobre tipos genéricos, para que Java en tiempo de ejecución determine el tipo:

```
Comparator<Comparable> comp = ((a, b) -> a.compareTo(b));
```

2: Pasar dicha instancia a la función de ordenación:

```
List<Integer> listaInt = new ArrayList<>();
```

```
// Poblar la lista ...
```

```
listaInt.sort(comp);
```

# Java lambdas: ¿Funciones como parámetros?

- Si quisiésemos en orden inverso: `listaInt.sort(comp.reversed());`
- Si quisiésemos ordenar una lista de doubles (o de Strings, o de cualquier cosa Comparable), también funciona! Java se encarga de inferir tipos en tiempo de ejecución.

```
List<Double> listaDouble = new ArrayList<>();
```

```
// Poblar la lista ...
```

```
listaDouble.sort(comp);
```

- Para mostrar por pantalla: `listaDouble.forEach(x->System.out.println(x));`

# Java lambdas: ¿Funciones como parámetros?

```
Empleado Eloy = new Empleado("Eloy", 25000, "Famoso corredor", 50);
```

```
Empleado Felix = new Empleado("Félix", 5000, "Currante serio y madrugador", 31);
```

```
Empleado Julio = new Empleado("Julio", 75000, "Rector", 52);
```

```
Empleado Laure = new Empleado("Laure", 100000, "Amante de los ordenadores", 52);
```

Podemos ordenar de forma muy fácil por cualquiera de los campos:

```
listaEmpleados.sort(Comparator.comparing(Empleado::getNombre));
```

```
listaEmpleados.sort(Comparator.comparing(Empleado::getSueldo));
```

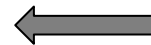
```
listaEmpleados.sort(Comparator.comparing(Empleado::getEdad));
```

# Java lambdas: variable capture

Como en el caso de clases anónimas, una expresión lambda puede acceder a variables locales “de fuera” en dos casos:

1. Si son finales
2. Si no son finales, pero se han inicializado solo una vez.

```
public Printer test() {  
    final String msg = "Hello";  
    Printer printer = msg1 -> System.out.println(msg + " " + msg1);  
    return printer;  
}
```



OK

```
public Printer test() {  
    String msg = "Hello";  
    Printer printer = msg1 -> System.out.println(msg + " " + msg1);  
    msg = "How are you?";  
    return printer;  
}
```



**Error de  
compilación**

# Java lambdas: variable capture

```
public class ExperimentoScope {  
    private String valor = "Hello from the outside";  
  
    public String DevuelveString() {  
        Ejemplo claseInterna = new Ejemplo() {  
            String value = "Valor en la clase interna";  
  
            public String metodo(String string) {  
                return this.value;  
            }  
        };  
        String resultIC = claseInterna.metodo("");  
  
        Ejemplo fooLambda = parameter -> {  
            String value = "Valor en la lambda";  
            return this.valor;  
        };  
        String resultLambda = fooLambda.metodo("");  
  
        return "Resultado IC = " + resultIC + "\r\n" +  
            "Resultado Lambda = " + resultLambda;  
    }  
    public static void main(String[] args) {  
        ExperimentoScope exp = new ExperimentoScope();  
        System.out.println(exp.DevuelveString());  
    }  
}
```

Resultado IC = Valor en la clase interna  
Resultado Lambda = Hello from the outside

# Java lambdas: ¿Cómo hacer una lambda serializable?

```
public class Main {  
    public static void main(String[] argv) {  
        Serializable ser = (Serializable & Calculator) (x,y)-> x + y;  
    }  
}
```

```
@FunctionalInterface  
interface Calculator{  
    long calculate(long x, long y);  
}
```

# Java lambdas: algunas desventajas...

Extraído de <http://blog.takipi.com/the-dark-side-of-lambda-expressions-in-java-8/>

```
// simple check against empty strings
public static int check (String s) {
    if (s.equals("")) { throw new IllegalArgumentException(); }
        else { return s.length(); }
}

//map names to lengths
List lengths = new ArrayList();
for (String name: Arrays.asList(args)) { lengths.add(check(name)); }
```

Traza de la excepción (código sin usar lambdas, Java 6 y Java 7):  
1 at Prueba.check(Prueba.java:19)  
2 at Prueba.main (Prueba.java:34)

# Java lambdas: algunas desventajas...

Extraído de <http://blog.takipi.com/the-dark-side-of-lambda-expressions-in-java-8/>

```
public static int check (String s) {  
    if (s.equals("")) { throw new IllegalArgumentException(); }  
        else { return s.length(); }  
}  
Stream lengths = names.stream().map(name -> check(name));
```

Traza de la excepción (Java 8):

```
01 at Prueba.check(Prueba.java:19)  
02 at Prueba.lambda$0(Prueba.java:37)  
03 at Prueba$$Lambda$1/821270929.apply(Unknown Source)  
04 at java.util.stream.ReferencePipeline$3$1.accept(ReferencePipeline.java:193)  
05 at java.util.Spliterators$ArraySpliterator.forEachRemaining(Spliterators.java:948)  
06 at java.util.stream.AbstractPipeline.copyInto(AbstractPipeline.java:512)  
07 at java.util.stream.AbstractPipeline.wrapAndCopyInto(AbstractPipeline.java:502)  
08 at java.util.stream.ReduceOps$ReduceOp.evaluateSequential(ReduceOps.java:708)  
09 at java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:234)  
10 at java.util.stream.LongPipeline.reduce(LongPipeline.java:438)  
11 at java.util.stream.LongPipeline.sum(LongPipeline.java:396)  
12 at java.util.stream.ReferencePipeline.count(ReferencePipeline.java:526)  
13 at Prueba.main(Prueba.java:39)
```



# Java lambdas: conclusiones

- No se pueden usar expresiones lambda solas, se necesita una interfaz funcional
- Las lambdas facilitan crear instancias de interfaces funcionales, sobre todo en comparación con clases anónimas
- Sintaxis concisa
- No tiene todo el poder de la programación funcional, aunque se pueden modelar alguna de sus ventajas, como el paso de funciones como parámetros (por medio de lambdas e instancias de interfaces funcionales).

# Java Streams

La vida antes (Java 1.7) de Streams:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
```

```
int sum = 0;
```

```
for (int n : numbers) {
```

```
    if (n % 2 == 1) {
```

```
        int square = n * n;
```

```
        sum = sum + square; }
```

```
}
```

# Java Streams

La vida después (Java 1.8 - ...) de Streams:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
```

```
int sum = numbers.stream()
```

```
    .filter(n -> n % 2 == 1)
```

```
    .map(n -> n * n)
```

```
    .reduce(0, Integer::sum);
```

# Java Streams

¿Y qué es una Stream...? En programación funcional se definen como

- “Streams are infinite lists”
- Definición Haskell: “data Stream a = Cons a (Stream a) deriving (Eq, Ord)”

<https://hackage.haskell.org/package/Stream-0.4.7.2/docs/src/Data-Stream.html>

- Isabelle/HOLCF:

“domain (unsafe) 'a stream = scon (ft::'a) (lazy rt::'"a stream") (infixr "&&" 65)”

# Java Streams

¿Y cómo se construye una Stream en programación funcional?

Siguiendo la definición de Streams, como una lista infinita...

<https://hackage.haskell.org/package/Stream-0.4.7.2/docs/Data-Stream.html#g:4>

# Java Streams

En jerga Java, una Stream es una interface (o mejor aún, una familia de interfaces)

<https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html>

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

“A sequence of elements from a source that support data processing operations”

Raoul-Gabriel Urma

# Java Streams

¿De dónde puedo sacar una Stream en Java...? (1 / 6)

- Desde una Stream
  - Por concatenación de dos Streams  
(<https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#concat-java.util.stream.Stream-java.util.stream.Stream->)
  - Por selección de elementos, por ejemplo de los distintos  
(<https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#distinct-->)

# Java Streams

¿De dónde puedo sacar una Stream en Java...? (2 / 6)

- Desde una Collection

(<https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html#stream-->)

- y por tanto desde List, Set, HashSet, Queue, Stack, Vector...



# Java Streams

¿De dónde puedo sacar una Stream en Java...? (3 / 6)

- Desde un Array:

<https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#stream-T:A->

# Java Streams

¿De dónde puedo sacar una Stream en Java...? (4 / 6)

- Desde una función iteradora o generadora (detalles más adelante)

# Java Streams

¿De dónde puedo sacar una Stream en Java...? (5 / 6)

- Desde un canal de I/O; por ejemplo, desde un fichero:

<https://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html#lines-java.nio.file.Path->

# Java Streams

¿De dónde puedo sacar una stream en Java...? (6 / 6)

- Desde un canal de I/O; por ejemplo, desde un fichero:

```
import java.io.IOException, java.nio.file.Files, java.nio.file.Path, java.nio.file.Paths,  
java.util.stream.Stream;
```

```
public class ReadFileStream {  
    public static void main(String [] args) {  
        Path path = Paths.get ("test.txt");  
        try ( Stream<String> lines = Files.lines(path) ) {  
            lines.forEach(s -> System.out.println(s));  
        }  
        catch ( IOException ex ) { }  
    }  
}
```

# Java Streams

Entonces, ¿una Stream en Java es una lista infinita?

No exclusivamente, Java es un más “permisivo” que los lenguajes funcionales “puros” y define una Stream como:

“*A sequence* of elements supporting sequential and parallel aggregate operations”

[https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.htm](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html)

l

# Java Streams

Propiedades (más) relevantes:

- no son almacenadas
- pueden representar una secuencia infinita de elementos
- están diseñadas para realizar iteraciones internas (implícitas)
- pueden ser procesadas en paralelo “automáticamente”
- soportan programación funcional
- soportan operaciones “lazy”
- pueden ser ordenadas o desordenadas
- no pueden ser reusadas

# Java Streams: no son almacenadas

- Un stream es “extraído” de una fuente de datos y procesado, pero no almacenado:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
```

```
int sum = numbers.stream()
```

```
    ....;
```

# Java Streams: pueden representar una secuencia infinita de datos

- Se pueden generar streams a partir de funciones iteradoras, y por tanto dar lugar a un stream infinito:

```
Stream<Long> longNumbers = Stream.iterate(1L, n -> n + 1);
```

Especificación del método (atención a los parámetros):

<https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#iterate-T-java.util.function.UnaryOperator->

*Hay un tipo específico “LongStream” que sería más adecuado para este caso de uso; lo evitamos por simplicidad*



# Java Streams: pueden representar una secuencia infinita de datos

- Se pueden generar streams a partir de funciones generadoras, y por tanto dar lugar a un stream infinito:

```
Stream.generate(Math::random)  
            ....;
```

Especificación del método (atención a los parámetros):

<https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#generate-java.util.function.Supplier->

# Java Streams: están diseñadas para realizar iteraciones internas

- ¿Dónde está el iterador externo (for, forEach...)? Las iteraciones sobre streams se realizan de forma interna:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
```

```
int sum = numbers.stream()
```

```
    .filter(n -> n % 2 == 1)
```

```
    .map(n -> n * n)
```

```
    .reduce(0, Integer::sum);
```

# Java Streams: están diseñadas para realizar iteraciones internas

- Ventaja adicional: los iteradores externos (for, while) producen generalmente código secuencial (no paralelizable, al menos de forma sencilla)
- Java ofrece un framework “Fork / Join” (en la JDK desde Java 1.7) para dividir tareas en subtareas recursivamente y ejecutarlas en paralelo; su uso directo no es trivial  
(<https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html>)

# Java Streams: están diseñadas para realizar iteraciones internas

En Java 1.8 podemos encontrar usos concretos del framework “Fork / Join” de manera transparente al usuario:

- Sobre Array:

<https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#parallelSort-T:A->

- Sobre Streams: `int sum = numbers.parallelStream()`

```
.filter(n -> n % 2 == 1)
```

```
.map(n -> n * n)
```

```
.reduce(0, Integer::sum);
```

# Java Streams: están diseñadas para realizar iteraciones internas

Streams también ofrece la posibilidad de crear iteradores explícitos, que nunca debería haber motivo para usar:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
```

```
Iterator iterator = numbers.stream().iterator();
```

```
while(iterator.hasNext()) {
```

```
    int n = iterator.next();
```

```
    ...
```

```
}
```

# Java Streams: soportan programación funcional

Una stream soporta dos tipos de operaciones:

- Operaciones intermedias
- Operaciones terminales

¿Cómo identificarías cada una de las siguientes?

```
int sum = numbers.parallelStream()
```

```
.filter(n -> n % 2 == 1)
```

```
.map(n -> n * n)
```

```
.reduce(0, Integer::sum);
```

# Java Streams: soportan programación funcional

Las operaciones sobre una Stream se combinan generando “stream pipelines”

Un “stream pipeline” consiste en:

- Una fuente de datos (Collection, Array, función generadora, canal de I/O)
- Cero o múltiples operaciones intermedias (distinct, sort, map, filter...)
- Una operación terminal (forEach, toArray, reduce, min, max, count, anyMatch...)

# Java Streams: soportan programación funcional

En realidad, cada operación de Stream lleva “asociado” su tipo:

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

- Las operaciones intermedias se gestionan de manera “perezosa” (solo cuando se llame a una operación terminal, o “temprana”, se procesarán sus elementos)
- Las operaciones terminales fuerzan el procesado (y también el fin) de la stream



# Java Streams: soportan programación funcional

Además, las operaciones intermedias se subdividen en:

- Operaciones intermedias “sin estado”: map, filter... (no necesitan conocer el estado de toda la stream)
- Operaciones intermedias “de estado completo”: sort, distinct... (necesitan conocer el estado de toda la stream)

Las primeras solo requieren procesar la stream una única vez... así que ofrecen mejores resultados al paralelizar

# Java Streams: soportan programación funcional

Las operaciones terminales también se subdividen en:

- Operaciones terminales (min, max, count, forEach...)...
- y operaciones terminales “cortocircuitadas” (anyMatch, allMatch, noneMatch...)

# Java Streams: pueden ser ordenadas o desordenadas, secuenciales o paralelas

- Una stream ordenada preservará el orden de sus elementos
- Una stream puede pasar de ordenada a desordenada (y viceversa)
- La interfaz “BaseStream” dispone de métodos “unordered()”, “sequential()” y “parallel()”

# Java Streams: no son reusables

- Una vez que hayamos invocado a cualquier operación terminal sobre una stream, la stream no puede ser reusada (ha sido “consumida”)
- Si la intentamos reusar, lanzará una `IllegalStateException`

# Java Streams: más difícil todavía...

¿Qué devolverá la siguiente “stream operation”?

```
¿...? = Stream.of("Ken", "Ellen", "Li")
```

```
.sorted ()
```

```
.filter (s -> s.length () > 10)
```

```
.findFirst();
```

# Java Streams: más difícil todavía...

Un objeto de tipo Optional...

```
Optional<String> os = Stream.of("Ken", "Ellen", "Li")
```

```
.sorted ()
```

```
.filter (s -> s.length () > 10)
```

```
.findFirst();
```

... cuyo valor es un “contenedor” (como un Array, List, etc) con 0 (None) ó 1 elementos

# Java Streams: más difícil todavía...

```
Optional<String> os = Stream.of("Ken", "Ellen", "Li")
```

```
    .sorted ()
```

```
    .filter (s -> s.length () > 10)
```

```
    .findFirst();
```

```
if (os.isPresent()) { System.out.println ("El primer nombre largo es " + os.get()); }
```

```
else { System.out.println ("La stream está vacía"); }
```

# Java Streams: más difícil todavía...

El tipo Optional es también un intento de “acabar” con las “populares”  
NullPointerException

Más detalles sobre Optional types en Java:

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



# Java Streams: algo para matemáticos...

¿Y si queremos calcular el número primo que ocupa la posición 1797778 ...?

```
Optional<Long> primo1797778 = Stream.iterate(1L, n -> n + 1)
    .filter (Pruebas::isPrime)
    .skip(1797778)
    .findFirst();

System.out.println ( "El primo en la posición 1797778 es " + primo1797778.get() );
```

El primo en la posición 1797778 es 28967683

# Java Streams: algo para matemáticos...

Cuidado, lo siguiente ya (tras unos cuantos segundos) no funciona...

```
Optional<Long> primo1797778 = Stream.iterate(1L, n -> n + 1)
```

```
.sorted()
```

```
.filter (Pruebas::isPrime)
```

```
.skip(1797778)
```

```
.findFirst());
```

*Exception in thread "main" java.lang.OutOfMemoryError: Java heap space*

# Java Streams: algo para matemáticos...

Aunque lo siguiente sí (la stream es evaluada de manera “lazy”)...

```
Stream<Long> naturalNumbers = Stream.iterate(1L, n -> n + 1);
```

```
Optional<Long> primo1797778 = naturalNumbers
```

```
.filter (Pruebas::isPrime)
```

```
.skip(1797778)
```

```
.findFirst();
```

```
System.out.println ("El primo en la posición 1797778 es " + primo1797778.get());
```

# Java Streams: agrupando datos (ejemplos)

```
class Person{  
    ...  
    public static List persons() {  
        Person ken = new Person(1, "Ken", Gender.MALE, LocalDate.of(1970, Month.MAY, 4), 6000.0);  
        ...  
        List<Persons> persons = Arrays.asList(ken, jeff, donna, chris, laynie, lee);  
        return persons; }  
}
```

# Java Streams: agrupando datos (ejemplos)

Método forEach

```
Person.persons().stream()
```

```
.filter( Person::isFemale )
```

```
.forEach( p -> p.setIncome( p.getIncome() * 1.1 ) );
```

# Java Streams: agrupando datos (ejemplos)

Método reduce (dos parámetros)

```
double sum = Person.persons()
```

```
    .stream()
```

```
    .reduce( 0.0, Double::sum );
```

# Java Streams: agrupando datos (ejemplos)

Método reduce (dos parámetros)

```
double sum = Person.persons()
```

```
    .stream()
```

```
    .reduce( 0.0, Double::sum );
```

*(Fallo de compilación...)*

# Java Streams: agrupando datos (ejemplos)

Método reduce (dos parámetros)

```
double sum = Person.persons()
```

```
    .stream()
```

```
    .map(Person::getIncome)
```

```
    .reduce(0.0, Double::sum);
```

Ahora sí, correcto...pero cuidado...



# Java Streams: agrupando datos (ejemplos)

Método reduce (dos parámetros)

```
double sum = Person.persons()
```

```
    .stream()
```

```
    .map(Person::getIncome)
```

```
    .reduce(0.0, Double::sum);
```

Restricciones “teóricas” no comprobadas por el compilador (“Road to perdition...”):

- el primer parámetro debe ser identidad de la segunda operación
- el segundo parámetro debe ser asociativo

# Java Streams: agrupando datos (ejemplos)

Método reduce (¿tres parámetros?)

```
double sum = Person.persons()
```

```
    .stream()
```

```
    .reduce(0.0, (partialSum, person) -> partialSum + person.getIncome(), Double::sum);
```

El segundo argumento ahora es una “BiFunction”, no un “BiOperator”...

¿Y el tercer argumento de reduce(..., ... , ...)?

# Java Streams: agrupando datos (ejemplos)

Método reduce (¿tres parámetros?)

```
double sum = Person.persons()
```

```
    .parallelStream()
```

```
    .reduce(0.0, (partialSum, person) -> partialSum + person.getIncome(), Double::sum);
```

El tercer parámetro se usa para *combinar* los resultados de los diferentes hilos:

<https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#reduce-U-java.util.function.BiFunction-java.util.function.BinaryOperator->

# Java Streams: creando nuevas agrupaciones

Método collect (tres parámetros); sin parametrizar...

```
List names = Person.persons()
```

```
    .stream()
```

```
    .map(Person::getName)
```

```
    .collect(ArrayList::new, ArrayList::add, ArrayList::addAll);
```

# Java Streams: creando nuevas agrupaciones

Método collect (tres parámetros); con parámetro de tipos...

```
List<String> names = Person.persons()  
  
    .stream()  
  
    .map(Person::getName)  
  
    .collect(ArrayList::new, ArrayList::add, ArrayList::addAll);
```

# Java Streams: creando nuevas agrupaciones

Método collect (tres parámetros); con parámetros de tipos...

```
List<String> names = Person.persons()  
  
    .stream()  
  
    .map(Person::getName)  
  
    .collect(ArrayList<String>::new, ArrayList<String>::add, ArrayList<String>::addAll);
```

# Java Streams: creando nuevas agrupaciones

Método collect (tres parámetros); pero no con diamante!!!

```
List<String> names = li
```

```
    .stream()
```

```
    .map(Person::getName)
```

```
    .collect(ArrayList<>::new, ArrayList<>::add, ArrayList<>::addAll);
```

Fallo de compilación!!!

# Java Streams: creando nuevas agrupaciones

También se puede hacer más fácil; clase Collector (métodos toList, toSet...)

```
List<String> names = Person.persons()  
  
    .stream()  
  
    .map(Person::getName)  
  
    .collect(Collectors.toList());
```



# Java Streams: creando nuevas agrupaciones

Clase Collector (también toMap...)

```
Map<String,Gender> ntg = Person.persons()
```

```
    .stream()
```

```
    .collect(Collectors.toMap(Person::getName, Person::getGender));
```

# Java Streams: creando nuevas agrupaciones

El siguiente enlace contiene ejemplos de algunas de las operaciones adicionales que se pueden hacer por medio de Collectors:

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

# Java Streams: ¿y quién paraleliza...?

La fragmentación de Streams para poder paralelizar se hace a través de la interface “SplitIterator” (y el método “trySplit”):

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

Reglas generales:

- Si el tamaño de la Stream es conocido, se hacen dos partes de igual tamaño (y se “recurre”...)
- Si el tamaño es desconocido, el “split” se hace, por ejemplo, por un tamaño fijo

La práctica puede no ser tan satisfactoria (mirar Demo...)

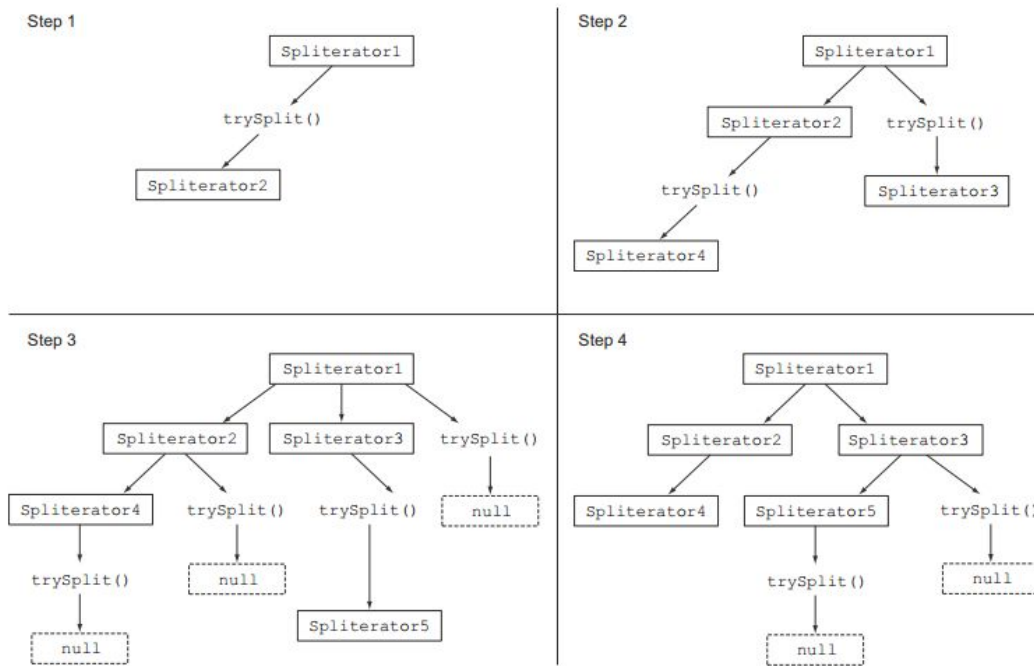
# Java Streams: ¿y quién paraleliza...?

SplitIterator se encarga tanto de :

- Recorrer elementos (como Iterator para Collection):
  - Individualmente (“tryAdvance(): boolean”)
  - En bloques (“forEachRemaining(): void”)
- Particionar elementos: “trySplit(): SplitIterator<T>”. Calcula la longitud del SplitIterator actual, y devuelve un nuevo SplitIterator con la primera mitad

# Java Streams: ¿y quién paraleliza...?

- El nuevo Splititerator recorre sobre la primera mitad, hasta que devuelve “null”, al encontrar un fragmento de trabajo que no puede particionar más



# Java Streams: ¿y quién paraleliza...?

- Con los “fragmentos” (Spliterator) de trabajo creados, el framework “fork/join” se encarga de distribuir los trabajos para los diferentes hilos (“fork”) y de recuperarlos para completar el resultado final (“join”)

# Java Streams: ¿y sobre la concurrencia?

- Al principio de la charla dijimos que las expresiones lambda solo pueden acceder a:
  - atributos finales
  - atributos efectivamente finales

para evitar problemas al paralelizar en múltiples hilos

Como el atributo final (o efectivamente final) no cambia de valor, todos los hilos comparten un único valor

# Java Streams: ¿y sobre la concurrencia?

¿Qué resultados obtenemos al ejecutar el siguiente código?

```
final long[] result = new long[1];

for (int i = 0; i < 8; i++) {

    result [0] = 0;

    LongStream.range(0, 1000).parallel()

                                                .forEach(n -> result[0] = (result[0] + n) * n);

    System.out.println("parallel: " + result[0]);

}
```



# Java Streams: ¿y sobre la concurrencia?

Por ejemplo, los siguientes...

parallel: 4978113844326184457

parallel: 5358490659060193433

parallel: -4385875300773810675

parallel: -772351014370484819

parallel: 228875638259599668

parallel: 6452004553751078580

parallel: 5726083396462237492

parallel: -8157228369161431244

# Java Streams: ¿y sobre la concurrencia?

¿Cuál ha sido el problema...?

El array usado es final, pero las componentes del array no...

(en realidad, en Java podemos hacer constante una referencia a un objeto, pero no el objeto apuntado por la misma)

Por tanto, podemos “saltarnos” el requisito de que la variable usada en la lambda expresión (“result [0]”, en lugar de “result”) sea constante;

- cada hilo la modifica en un orden y con unos valores diferentes, dando lugar a resultados no deterministas

# Java Streams: ¿y sobre la concurrencia?

¿Cómo podríamos solucionarlo?

En realidad, poner efectos laterales en una expresión lambda debería estar prohibido

# Java Streams: ¿y sobre la concurrencia?

Otras opciones... Hacerlo secuencial:

```
final long[] result = new long[1];
```

```
for (int i = 0; i < 10; i++) {
```

```
    result [0] = 0;
```

```
    LongStream.range(0, 1000)
```

```
                .forEach(n -> result[0] = (result[0] + n) * n);
```

```
    System.out.println("serial: " + result[0]);
```

```
}
```

# Java Streams: ¿y sobre la concurrencia?

Otras opciones... Usar forEachOrdered:

```
final long[] result = new long[1];
```

```
for (int i = 0; i < 10; i++) {
```

```
    result [0] = 0;
```

```
    LongStream.range(0, 1000).parallel()
```

```
        .forEachOrdered(n -> result[0] = (result[0] + n) * n);
```

```
    System.out.println("parallel ordered: " + result[0]);
```

```
}
```

<https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#forEachOrdered-java.util.function.Consumer->

# Java Streams: ¿y sobre la concurrencia?

Opciones erróneas... no vale con usar “sorted()” (“forEach” no preserva el orden)

```
final long[] result = new long[1];
```

```
for (int i = 0; i < 10; i++) {
```

```
    result [0] = 0;
```

```
    LongStream.range(0, 1000).parallel().sorted()
```

```
        .forEach(n -> result[0] = (result[0] + n) * n);
```

```
    System.out.println("parallel sorted: " + result[0]);
```

```
}
```

# Java Streams: ¿y sobre la concurrencia?

Opciones erróneas... ni si intentamos “colar” una variable de tipo básico como “final” (o efectivamente “final”)

Lo siguiente ni siquiera compila (tampoco quitando “final”):

```
final long result = 0;

for (int i = 0; i < 10; i++) {
    LongStream.range(0, 1000)
                .forEach(n -> result = (result + n) * n);

    System.out.println("serial: " + result);
}
```

# Java lambdas y streams: ¿limitaciones?

- Una de ellas, code debugging (ya mencionado)
- La siguiente serie de posts identifica algunas otras debilidades, especialmente desde el punto de vista de la comparación con la Programación Funcional

- <https://dzone.com/articles/whats-wrong-java-8-currying-vs>
- <https://dzone.com/articles/whats-wrong-java-8-part-ii>
- <https://dzone.com/articles/whats-wrong-java-8-part-iii>
- <https://dzone.com/articles/whats-wrong-java-8-part-iv>
- <https://dzone.com/articles/whats-wrong-java-8-part-v>
- <https://dzone.com/articles/whats-wrong-java-8-part-vi>
- <https://dzone.com/articles/whats-wrong-java-8-part-vii>



# Java lambdas y streams: ¿limitaciones?

Los siguientes artículos también apuntan algunas limitaciones relevantes de la tecnología

- <http://blog.takipi.com/the-dark-side-of-lambda-expressions-in-java-8/>
- <http://blog.takipi.com/new-parallelism-apis-in-java-8-behind-the-glitz-and-glamour/>

# Java lambdas y streams: demo

- Fichero Java 8 en Eclipse

# Java lambdas y streams: conclusiones

- Java 8 ofrece herramientas para aumentar la sencillez (¿la calidad?) del código que generamos
- Permite trabajar con funciones como parámetros...siempre que haya una interface detrás...
- Permite trabajar con “streams” de datos de forma perezosa
- Permite paralelizar, por medio de Fork/Join, el procesado de “streams”

Para realizar esta presentación hemos tomado parte de la estructura y de los ejemplos del texto “Beginning Java 8 Language Features” de Kishori Sharan (Apress, 2014); lo puedes encontrar en la BibUR en <https://link.springer.com/book/10.1007%2F978-1-4302-6659-4>