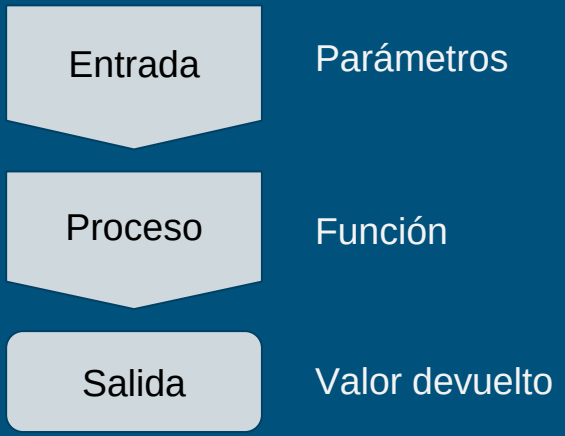
Programación funcional en Java 8

Guillermo Siles Bonilla
Francisco Molina Sánchez

Programación funcional

- Es un paradigma de **programación declarativa** basado en las funciones matemáticas
- El valor devuelto por una función depende **exclusivamente** de los argumentos de entrada
  - **Transparencia referencial** de sus variables



Novedades en Java 8

1. Interfaces funcionales
2. Expresiones Lambda
3. Stream API
4. Nuevo motor de Javascript Nashorn

1.Interfaces funcionales

- Una interfaz es funcional si declara **solamente un método abstracto**
- Pueden tener varios **métodos predefinidos**
- Es opcional usar la anotación ***@FunctionalInterface***
- Pueden ser instanciadas por las **expresiones lambda**

2.Expresiones Lambda

- Código más **compacto y limpio**
- Inferencia de tipos
- Podemos **pasar funciones** como **parámetros**
- Métodos anónimos
- Introduce la **programación funcional** en Java 8
- Gran sinergía con **Streams API**

2.Expresiones Lambda. Ejemplos básicos

Forma básica arg -> body	**Forma extendida** (**T1** arg1, **T2** arg2) -> { body }
x -> x*2	(int x)->{return x*2;}
(x,y)->x+y	(int x,int y)->{return x+y;}
()->3	()->{return 3;}
	(Integer x)-> { Integer y; y = x + 1; return y; }

2.Expresiones Lambda. Ejemplos básicos

Forma general	Forma reducida
x -> x.toString()	Object::toString
x -> String.valueOf(x)	String::valueOf
() -> new ArrayList<>()	ArrayList::new

Ej.1: Interfaz Funcional y Expresiones Lambda

```java
public interface InterfazFuncional{
    public void sumar(int v1,int v2);
}
```

Clase anónima interna

```java
public class ClaseInternaAnonima {
    public static void main(String[] args) {
        InterfazFuncional x = new InterfazFuncional() {
            @Override
            public void sumar(int v1, int v2) {
                int res=v1+v2;
                System.out.println(res);
            }
        };
        x.sumar(1, 2);
    }
}
```

```
<terminated> ClaseInternaAnonima
3
```

Ej.1: Interfaz Funcional y Expresiones Lambda

```java
@FunctionalInterface
public interface InterfazFuncional{
    public void sumar(int v1,int v2);

    public default int doble(int n) {
        return 2*n;
    }

    public default int triple(int n) {
        return 3*n;
    }
}
```

Métodos por defecto

- ✓ Nos permiten ampliar nuestro código sin modificar el código antiguo

Ej.1: Interfaz Funcional y Expresiones Lambda

```java
@FunctionalInterface
public interface InterfazFuncional{
    public void sumar(int v1,int v2);

    public default int doble(int n) {
        return 2*n;
    }
```

Expresión lambda

```java
public class Lambda {
    public static void main(String[] args) {
        InterfazFuncional x = (int v1,int v2) -> {
                    int res= v1+v2;
                    System.out.println(res);
        };
        x.sumar(1, 2);

        System.out.println(x.doble(2));
    }
}
```

```
<terminated> Lambda [Java Application]
3
4
```

Ej.2: Filtrar Archivos


```
File directorioActual = new File(System.getProperty("user.dir"));

String[] archivos= directorioActual.list();
String[] archivosFiltrados= directorioActual.list(new FiltrarArchivosTexto());
```

Devuelve todos los ficheros

FilenameFilter filtro

```
public class FiltrarArchivosTexto implements FilenameFilter{
    @Override
    public boolean accept(File directorio, String nombre) {
        return nombre.endsWith(".txt");
    }
}
```

Devuelve TRUE si el nombre del fichero debe ser almacenado en el array de String

```
<terminated> FiltrarArchivos

Sin filtrar:
.classpath
.git
.gitattributes
.gitignore
.project
.settings
bin
Prueba - copia.txt
Prueba.txt
src
```

```
Filtrando ".txt":
Prueba - copia.txt
Prueba.txt
```

Ej.2: Filtrar Archivos

```
File directorioActual = new File(System.getProperty("user.dir"));

String[] archivos = directorioActual.list((directorio,nombre)->
                    nombre.endsWith(".txt"));
```

Expresión lambda

filtro

Devuelve TRUE si el nombre del fichero debe ser almacenado en el array de String


```
<terminated> FiltrarArchivosLambda
Prueba - copia.txt
Prueba.txt
```

2.Interfaces Funcionales Básicas

- import java.util.function;

Interface	Función	Tipo devuelto
UnaryOperator<T>	apply(T a)	T
BinaryOperator<T>	apply(T a,T b)	T
Function<T,S>	apply(T a)	S
BiFunction<T,U,S>	apply(T a,U b)	S
Consumer<T>	accept(T a)	void
Predicate<T>	test(T a)	Boolean (Condición)
Supplier<T>	get()	T (Collections, Streams)

Ej.3: Interfaces Funcionales Básicas

Function<T,S>	apply(T a)	S

```
Function<String,Integer> hashCodeL = (contraseña)-> contraseña.hashCode();
System.out.println("\nEl hashcode es: "+ hashCodeL.apply("contraseña"));
```

```
El hashcode es: 624705475
```

Predicate<T>	test(T a)	Boolean (Condición)

```
Predicate<Double> verificar = (nota) -> nota >= 5;
System.out.println("\n¿Aprobado?: " + verificar.test(7.0));
```

```
¿Aprobado?: true
```

Ej.3: Interfaces Funcionales Básicas

Consumer<T>	accept(T a)	void

```
Consumer<String> imprimir = cadena -> System.out.println(cadena);
imprimir.accept("\nEsto es un String.");
```

```
Esto es un String.
```

```
lista = Arrays.asList("\nEjemplo 1","Ejemplo 2","Ejemplo 3");
lista.forEach(imprimir);
```

Consumer

```
Ejemplo 1
Ejemplo 2
Ejemplo 3
```

Ej.4: Funciones de orden superior

- Algún argumento de entrada es una función o devuelven una función.

```
public static Integer procesar(Function<String,Integer> operacion,String cadena) {
    return operacion.apply(cadena);
}
```

Función como argumento de entrada

```
List<String> lista= Arrays.asList("Ejemplo 1","Ejemplo 2","Ejemplo 3");

lista.forEach(cadena -> System.out.println(
                    procesar(x -> x.hashCode() ,cadena)));
```

Expresión lambda

(Parámetro de entrada de la función procesar)

```
<terminated> F
-1730076361
-1730076360
-1730076359
```

Ej.5: Devolviendo una función

```
enum TipoEvaluacion {Continua, ExamenFinal};

public static BiFunction<Double, Double, Double> calcularNotaFinal
(TipoEvaluacion tipo) {

    switch (tipo) {
    case Continua:
        return (parcial, exFinal) -> parcial*0.2 + exFinal*0.8;
    case ExamenFinal:
        return (parcial,exFinal) -> exFinal;
    default:
        return null;
    }
}
```

Función como argumento de retorno

Función devuelta

```
<terminated> DevolverFuncion
Nota final: 7,30
```

```
System.out.println("Nota final: "+ String.format("%.2f",
        calcularNotaFinal(TipoEvaluacion.Continua).apply(4.5,8.0)));
```

Devuelve un objeto de tipo BiFunction

✓ Ventaja: podemos usar la expresión lambda en diversos contextos

Ej.6: Asignando una expresión lambda o función

```
public static BiFunction<Double, Double, Double> calcularNotaFinal
(TipoEvaluacion tipo) {
```

```
BiFunction<Double, Double, Double> variable =
        calcularNotaFinal(TipoEvaluacion.ExamenFinal);
```

Asignamos la función

```
System.out.println("Nota final: "+String.format("%.2f",
        variable.apply(4.5,7.0)));
```

Devuelve un objeto de tipo BiFunction

- ✓ Nos permite usar un código más compacto

```
<terminated> DevolverFuncion
Nota final: 7,00
```

Ej.7: Currificación: Funciones parciales

Sin currificación:

```
BiFunction<Integer, Integer, Boolean> mayorQue = (x, y) -> x>y ;
System.out.println("Sin currificacion: " +mayorQue.apply(2, 3)+" \n");
```

```
Sin currificacion: false
```

Con currificación:

```
Function<Integer, Function<Integer,Boolean>> mayorQueC = x -> y -> x>y ;
Function<Integer,Boolean> funcIntermedia;
```

Argumentos de entrada

```
funcIntermedia = mayorQueC.apply(2);
```

← Primer argumento de entrada

```
System.out.println(funcIntermedia.apply(3));
```

← Segundo argumento de entrada

```
Con Currificacion:
Esto es una función intermedia ej7.Currificacion$$Lambda$3/303563356@87aac27
false
```

Ej.7: Currificación: Funciones parciales

```
Function<Integer, Function<Integer,Boolean>> mayorQueC = x -> y -> x>y ;
```

```
List<Integer> lista = Arrays.asList(1,2,3,4,5,6,7,8,9,10);
System.out.println("\n¿Es mayor que 4?");
lista.forEach(x-> System.out.println(x+" "+mayorQueC.apply(x).apply(4)));
```

```
¿Es mayor que 4?
1 false
2 false
3 false
4 false
5 true
6 true
7 true
8 true
9 true
10 true
```

Pasamos como primer parámetro cada elemento de la lista

Segundo parámetro de entrada de la función parcial

Ej.8: Composición de funciones

```
Function<Integer, Integer> f = x -> x + 2;
Function<Integer, Integer> g = x -> x * 5;
Function<Integer, Integer> c1,c2;
```

f(x) = x + 2
g(x) = x * 5

```
c1 = f.compose(g);
c2 = f.andThen(g);
```

f º g (x*5)+2
g º f (x+2)*5

```
c1.apply(5)
c2.apply(5)
```

c1(5) = 27
c2(5) = 35

Salida:

```
c1(5) = 27
c2(5) = 35
```

3. Streams

- Nada que ver con los clásicos InputStream / OutputStream .
- ¡Son mónadas! -> Anidan funciones del mismo tipo.
- Un stream representa un conjunto de elementos sobre los que se realizan distintas operaciones.
- Estas operaciones pueden ser terminales o no terminales.
- Los streams pueden ser secuenciales o paralelos (lo veremos más adelante).
- Se recorren una sola vez.

Streams / Ejemplos

- Creamos una lista
- Aplicamos stream con una serie de funciones.
- Filter, sorted y map son no terminales.
- forEach es terminal.

```java
public class Ejemplo1 {
    public static void main(String [] args){
        List<String> l = Arrays.asList("Guille", "Fran", "Daniel", "Gabriel");
        l.stream()
        .filter(lalista -> lalista.startsWith("G"))
        .sorted()
        .map(String::toUpperCase)
        .forEach(System.out::println);
    }
}
```

```
<terminated> Ejemplo1 [Java Application] C:\Program Files\Java\jre1.8.0_66\bin\javaw.exe
GABRIEL
GUILLE
```

Streams / funciones básicas

- Map. No terminal. Aplica una función a todos los elementos.
- Foreach. Terminal. Parecida a Map.
- Filter. No terminal. Aplica un filtro.
- Anymatch. Terminal. Devuelve true o false si algún elemento cumple el predicado.
- Get. No terminal. Se usa para reutilizar streams (lo veremos más adelante).

Streams / lazy

- Las funciones no terminales son perezosas.
- Se dividen en dos categorías:
- Sin estado. No necesitan guardar el estado de operaciones anteriores. Ej: map, filter…
- Con estado. Necesitan guardar el estado de operaciones anteriores. Ej: sorted, distinct...

```java
public class Ejemplo2 {
    // Ejemplo de funciones lazy.
    public static void main(String [] args){
        List<String> l = Arrays.asList("Guille", "Gabriel");
        l.stream()
        .filter(s -> {
            System.out.println("filter: " + s);
            return true;
        });
    }
}
```

```
<terminated> Ejemplo2 [Java Application] C:\Program Files\Java\jre1.8.0_66\bin\javaw.exe
```

Streams / lazy

¿Cuál es la salida si añadimos una función terminal?

```
public class Ejemplo3 {
    // Ejemplo de funciones lazy
    public static void main(String [] args){
        List<String> l = Arrays.asList("Guille", "Gabriel");
        l.stream()
        .filter(s -> {
            System.out.println("filter: " + s);
            return true;
        })
        .forEach(s -> System.out.println("forEach: " + s));
    }
}
```

Streams / lazy

¿Cuál es la salida si añadimos una función terminal?

Al ser comportamiento perezoso, el contenido de filter se ejecuta solo cuando va a ejecutarse forEach.

```java
public class Ejemplo3 {
    // Ejemplo de funciones lazy
    public static void main(String [] args){
        List<String> l = Arrays.asList("Guille", "Gabriel");
        l.stream()
        .filter(s -> {
            System.out.println("filter: " + s);
            return true;
        })
        .forEach(s -> System.out.println("forEach: " + s));
    }
}
```

```
<terminated> Ejemplo3 [Java Application] C:\Program Files\Java\jr
filter: Guille
forEach: Guille
filter: Gabriel
forEach: Gabriel
```

Streams / datos

- Los datos int, double y long tienen clases de stream propias.
- Poseen funciones extra, como sum, average…
- Hay funciones map para cambiar de un tipo a otro.

```
public class Ejemplo4 {
    public static void main(String [] args){
        List<String> l = Arrays.asList("1. Guille", "2. Gabriel");
        l.stream()
        .map(s -> s.substring(0, 1))
        .mapToInt(Integer::parseInt)
        .max()
        .ifPresent(System.out::println);
    }
}
```

```
<terminated> Ejemplo4 [Java Application] C:\Program Files\Java\jre1.8.0_66\
2
```

Streams / orden

El orden de las funciones no terminales sí altera el producto.

```
// Caso no eficiente
l.stream()
.map(s -> {
    System.out.println("Map.");
    return s.toUpperCase();
})
.filter(s -> {
    System.out.println("Filter.");
    return s.length() == 5;
})
.forEach(System.out::println);
```

```
// Caso eficiente
l.stream()
.filter(s -> {
    System.out.println("Filter.");
    return s.length() == 5;
})
.map(s -> {
    System.out.println("Map.");
    return s.toUpperCase();
})
.forEach(System.out::println);
```

Streams / reutilización

- Cuando llegamos a una función terminal, el stream no se puede reutilizar.
- Sin embargo, hay una solución.

```
public class Ejemplo6 {
    // Reutilización
    public static void main(String [] args){
        IntStream stream = IntStream.of(1, 2);
        stream.forEach(System.out::println);
        stream.forEach(System.out::println);
    }
}
```

```
<terminated> Ejemplo6 [Java Application] C:\Program Files\Java\jre1.8.0_66\bin\javaw.exe
1
2
Exception in thread "main" java.lang.IllegalStateException:
        at java.util.stream.AbstractPipeline.sourceStageSpli
        at java.util.stream.IntPipeline$Head.forEach(Unknowr
        at streams.Ejemplo6.main(Ejemplo6.java:11)
```

Streams / reutilización

- Para solucionar este problemas usamos la función get.
- Cada vez que usamos la función get, obtenemos un nuevo stream al que aplicaremos la función terminal.

```
Supplier<Stream<String>> streamSupplier =
    () -> Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> s.startsWith("a"));

streamSupplier.get().forEach(System.out::println);
streamSupplier.get().forEach(System.out::println);
```

```
<terminated> Ejemplo6 [Java Application] C:\Program Files\Java\
a2
a2
```

Streams / accidentes

Hay varios errores básicos que se pueden cometer cuando se comienza a trabajar con streams.

- Crear streams infinitos.
- Equivocarse al elegir el orden de las funciones.
- Querer reusar el stream.
- Olvidarse de la función terminal.

Streams / accidentes

- En el primero olvidamos limitar el iterate.
- En el segundo nunca alcanzamos el limite de 10.
- El tercero es similar al segundo pero en paralelo, podríamos terminar teniendo que reiniciar la máquina.

```
public class Ejemplo7 {
    // Accidentes
    public static void main(String [] args){


        IntStream.iterate(0, i -> i + 1)
        .forEach(System.out::println);

        IntStream.iterate(0, i -> ( i + 1 ) % 2)
        .distinct()
        .limit(10)
        .forEach(System.out::println);

        IntStream.iterate(0, i -> ( i + 1 ) % 2)
        .parallel()
        .distinct()
        .limit(10)
        .forEach(System.out::println);
    }
}
```

Streams / operaciones avanzadas

Collect es una operación que usamos para a partir de un stream crear nuevas colecciones.

En el ejemplo, si no añadimos collect el código no compila.

```
public class Ejemplo8 {

    // Collect

    public static void main(String [] args){

        List<String> nombres = Arrays.asList("Pepe", "Juan", "Guille", "Ana");
        List<String> nombresf = nombres.stream()
                                .filter(s -> s.startsWith("G"))
                                .collect(Collectors.toList());
        System.out.println(nombresf.toString());

        List<Integer> salarios = Arrays.asList(1000,980,1200,5500,1100);
        List<Integer> salariosord = salarios.stream()
                                    .sorted()
                                    .collect(Collectors.toList());
        System.out.println(salariosord.toString());

    }

}
```

Streams / operaciones avanzadas

¡Reduce es nuestro fold!
Los hay de tres tipos.

- Una sola función de acumulación.
- Valor base y función de acumulación.
- Valor base, función de acumulación y función de combinación.

```
public static void main(String [] args){
    List<String> nombres = Arrays.asList("Ana", "Guille", "David", "Fran");
    nombres
    .stream()
    .reduce((n1, n2) -> n1.compareTo(n2) <= 0 ? n1 : n2)
    .ifPresent(System.out::println);

    String todos = nombres
    .stream()
    .reduce("\nLaura", (n1, n2) -> (n1 + n2));
    System.out.println(todos);

    List<Person> p = Arrays.asList(new Person(19), new Person(20), new Person(49));
    p.stream()
    .reduce(0, (sum, person) -> sum += person.getAge(), (sum1, sum2) -> sum1 + sum2);
}
```

Streams / stream en paralelo

- Las colecciones permiten stream en paralelo.
- Podemos modificar el número de hilos que usamos.
- Parallelstream() en lugar de stream()
- En la siguiente diapositiva tenemos un ejemplo de parallelstream en el que podemos ver el uso del cpu.

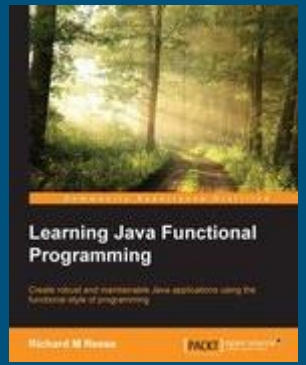
Streams / stream en paralelo

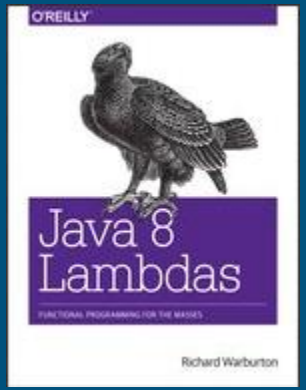
```java
public static void main(String [] args){
    Arrays.asList("a1", "a2", "b1", "c2", "c1")
    .parallelStream()
    .filter(s -> {
        System.out.format("filter: %s [%s]\n",
            s, Thread.currentThread().getName());
        return true;
    })
    .map(s -> {
        System.out.format("map: %s [%s]\n",
            s, Thread.currentThread().getName());
        return s.toUpperCase();
    })
    .forEach(s -> System.out.format("forEach: %s [%s]\n",
        s, Thread.currentThread().getName()));
}
```

```
<terminated> Ejemplo10 [Java Application] C:\Program Files\Java\jre1.8.0_66\
filter: b1 [main]
filter: c1 [ForkJoinPool.commonPool-worker-3]
map: c1 [ForkJoinPool.commonPool-worker-3]
filter: a1 [ForkJoinPool.commonPool-worker-2]
filter: a2 [ForkJoinPool.commonPool-worker-1]
map: a2 [ForkJoinPool.commonPool-worker-1]
map: a1 [ForkJoinPool.commonPool-worker-2]
forEach: A1 [ForkJoinPool.commonPool-worker-2]
forEach: C1 [ForkJoinPool.commonPool-worker-3]
map: b1 [main]
forEach: B1 [main]
filter: c2 [ForkJoinPool.commonPool-worker-2]
map: c2 [ForkJoinPool.commonPool-worker-2]
forEach: A2 [ForkJoinPool.commonPool-worker-1]
forEach: C2 [ForkJoinPool.commonPool-worker-2]
```

Bibliografía

- Learning Java Functional Programming (Richard M Reese, 2015)
- Java 8 in Action: Lambdas, streams, and functional-style programming (Raoul-Gabriel Urma, Mario Fusco, and Alan Mycroft, 2014)
- Java 8 Lambdas (Richard Warburton, 2014)





Referencias

- Programación funcional (Wikipedia)
- Expresiones Lambda Java 8 (Youtube)
- Java 8: Interfaces Funcionales (Youtube)
- Lambdas y Stream API | Alexis Lopez | Jespañol (Youtube)
- Lambdas y API Stream - Apuntes de Java (Slideshare)
- Trying Out Lambda Expressions in the Eclipse IDE (Oracle)

Referencias

- Tutorial stream
- Otro tutorial de stream
- Api stream java
- Errores comunes en java8
- Curso Java – Expresiones Lambda (Wiseratel)