{codemotion}
MADRID · NOV 27-28 · 2015

Java 8 en Accion

Denis Simon Soneira
Aurea Muñoz Hernandez
Katia Aresti Gonzalez

Quiénes somos

aureamunioz@gmail.com

denis.simon@gmail.com

La programación funcional

Programación imperativa

```
public int moreThanTwentyYearsOld()
{
    int moreThanTwenty = 0;
    for (Person person : persons) {
        if (person.age > 20) {
            moreThanTwenty++;
        }
    }
    return moreThanTwenty;
}
```

La programación imperativa describe las acciones que, modificando el estado de la máquina, para obtener el resultado.

Basada en la Maquina de Turing.

Programación declarativa

```
select count(*)
from Person p
where p.age > 20
```

La programación consiste en sentencias que describen la lógica de computación.

No hay flujos.

Dentro de los tipos de programación declarativa está la funcional.

Programación funcional

```
def sum(xs: List[int]): int = xs match {
        case Nil => 0
        case head :: tail => head + sum(tail)
}
```

En un lenguaje funcional puro :

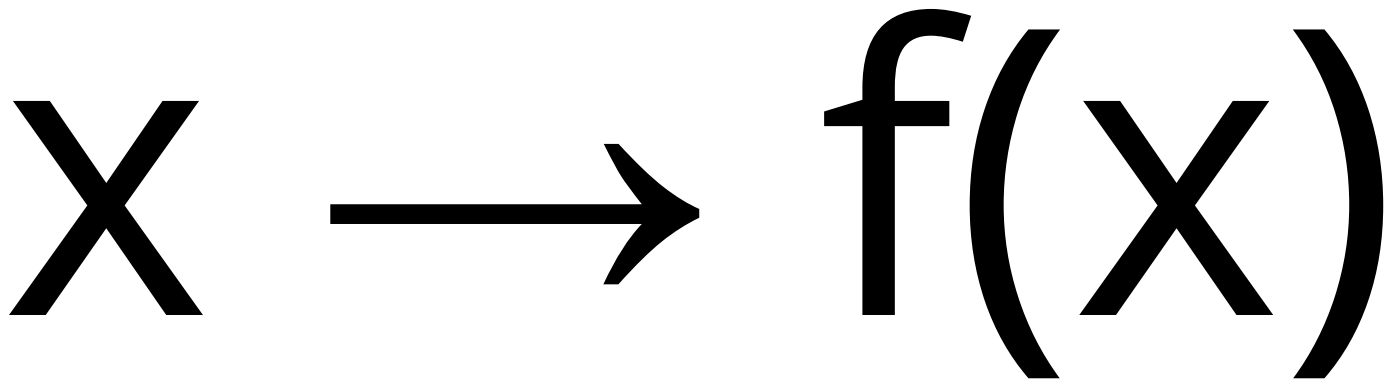
1. Todas las funciones son puras. No hay cambios de estado.
2. La recursividad es omnipresente.
3. Basada en el calculo Lambda. Funciones de orden superior.

Programación funcional en Java 8

```
public int moreThanTwentyYearsOld()
{
    int moreThanTwenty = 0;
    for (Person person : persons) {
        if (person.age > 20) {
            moreThanTwenty++;
        }
    }
    return moreThanTwenty;
}
```

```
public int moreThanTwentyYearsOld()
{
    return persons.stream()
      .filter(p -> p.getAge() > 20)
      .count();
}
```

Las expresiones Lambda (λ)



Las expresiones Lambda

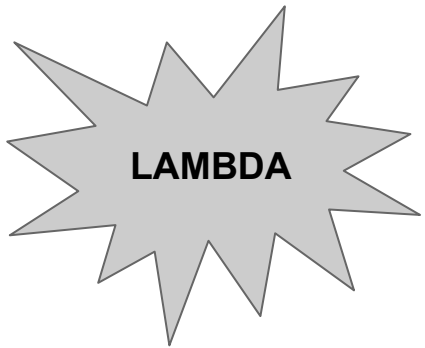
Una expresión lambda es una instancia de una “Functional interface”.

```
@FunctionalInterface
public interface X<T,R> {
    R method(T value);
}
```

Las interfaces funcionales definen un método abstracto único y pueden estar anotadas con **@FunctionalInterface**

Transformación en lambda

```
Function<Person, Integer> mapper =  new Function<Person, Integer>() {
    @Override
    public Integer apply(Person person) {
        return person.getAge();
    }
};
```




```
Function<Person, Integer> mapper = (person) -> person.getAge();
```

Tipos de expresiones lambda

(parameters) -> expression

```
()->{}
() -> "Raoul"
```

(parameters)->{statements;}

```
(i,j) -> {return i+j;}
```

Referencia al método

```
Person::getAge
```

Streams

Un stream es una secuencia de elementos tomados de una fuente (Collection) que soporta operaciones de procesamiento de datos.

Streams

```
public List<String> getNamesOfLowCaloriesDishes() {
    List<String> lowCaloriesDishesNames = new ArrayList<>();
    for (Dish dish : menu) {
        if (dish.getCalories() < 400) {
            lowCaloriesDishesNames.add(dish.getName() );
        }
    }
    return lowCaloriesDishesNames;
}
```

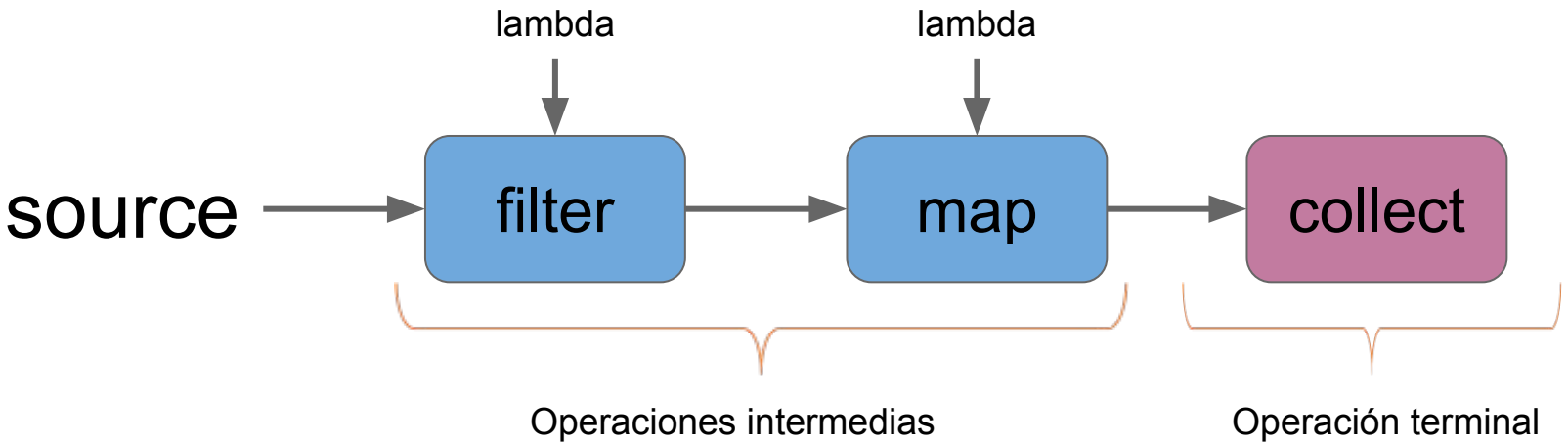
Antes (java 7)

```
public List<String> getNamesOfLowCaloriesDishes() {
    return menu.stream().filter(d->d.getCalories()<400)
               .map(Dish::getName).collect(Collectors.toList() );
}
```

Después (java 8)

Streams

Para ejecutar operaciones sobre un stream, éstas se efectúan en un **Stream Pipeline**



Streams vs Collection

Collections

- Almacenamiento de datos
- Modificación de datos (añadir, suprimir)
- Iteración externa

Streams

- Transformación de datos
- Evaluación lazy
- No están delimitados
- No son reutilizables
- Iteración interna

Default methods

Nueva característica de Java 8 cuyo propósito es facilitar la evolución de APIs existentes garantizando la retro compatibilidad.

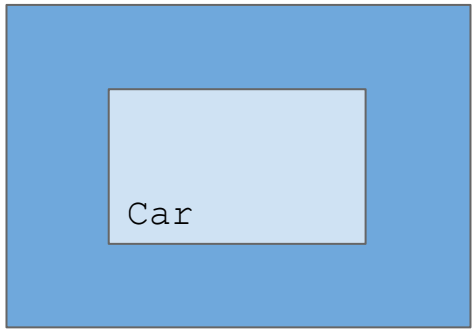
```
public interface Collection<E> extends Iterable<E> {
…
    default Stream<E> stream() {
        return StreamSupport.stream(spliterator(), false);
    }
}
```

Defaults methods

```
public interface A{
     default void hello(){
           System.out.println("Hello from A");
      }
}
public interface B extends A{
     default void hello(){
          System.out.println("Hello from B");
     }
}
public class C implements B,A{
     public void main (String...args){
       new C().hello();
     }
}
```

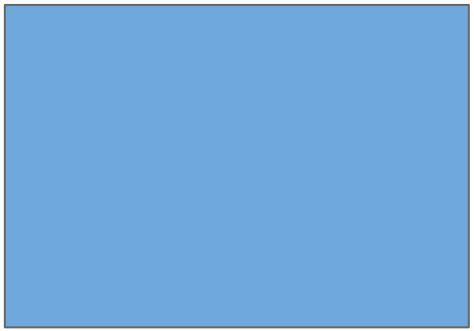
Optional (mejor alternativa a null)

`Optional<Car>`



Contiene un objeto de tipo Car

`Optional<Car>`



Está vacío

Nuevas API Date y Time

```
Date date = new Date(114, 2, 18);

//Printing this date produces
Tue Mar 18 00:00:00 CET 2014
```

Antes (java 1)

```
LocalDate date = LocalDate.of(2014, 3, 18);
int year = date.getYear();
Month month = date.getMonth();
int day = date.getDayOfMonth();
```

2014

MARCH

18

Después (java 8)

Ejercicios Prácticos : prerrequisitos

1. JDK 8

   ```
   http://www.oracle.com/technetwork/java/javase/downloads
   ```

2. GIT

   ```
   https://git-scm.com/downloads
   ```

3. Maven 3

   ```
   https://maven.apache.org
   ```

4. IDE JAVA (Eclipse, IntelliJ, ...)
5. Clone del repositorio GIT :

   ```
   git clone https://github.com/2nis6mon/dojo-java8.git dojo-java8
   ```

6. Abrir el nuevo repositorio “dojo-java8” en tu IDE
7. Ejecuta los tests del proyecto. Si todos estan en VERDE estas LISTO para EMPEZAR

Ejercicios Practicos 1 al 5

Checkout la rama « java8-X » (Donde X es el numero del ejercicio)

git checkout –b java8-X remotes/origin/java8-X

Seguir las instrucciones en las clases del package (src/main/java/)

org.dojo.java8.exerciseX

Una vez finalizado el ejercicio commit las modificaciones antes de pasar al siguiente

git commit -am "java8-X solution"

Puedes encontrar las soluciones de cada ejercicio en una rama dedicada « java8-X-solution »

git checkout –b java8-X-solution remotes/origin/java8-X-solution