1. Refuerzo de Java

Parte III. Programación funcional

Programación de Servicios y Procesos

Arturo Bernal
Nacho Iborra
Javier Carrasco

Tabla de contenido

7. ¿Qué es la programación funcional?

La programación funcional es un paradigma de programación (es decir, una forma de implementar programas) que se enfoca en la evaluación de funciones matemáticas. Se basa en cálculos lambda, un sistema nacido en 1930 para evaluar la definición, aplicación y recursividad de funciones.

El paradigma funcional es **declarativo**, lo que significa que, el código se hace con declaraciones en lugar de sentencias. En otras palabras, se le dice al programa cómo están las cosas, en lugar de cómo resolver un problema. En el lado opuesto se encuentran los lenguajes **imperativos**, que se basan en comandos en el código fuente, como asignaciones, que le dicen al programa cómo resolver el problema.

7.1. Algunas características de la programación funcional

La programación funcional se basa en algunos conceptos fundamentales como:

- **Transparencia referencial**: la salida de una función depende solo de sus argumentos, por lo que si llama muchas veces con los mismos argumentos, siempre obtendremos el mismo resultado. Entonces, la programación funcional no tiene efectos secundarios que puedan modificar algo fuera de la función.
- **Inmutabilidad de datos**: para asegurarse que no se produzcan efectos secundarios al llamar a funciones, una de las reglas más importantes del paradigma de la programación funcional es que los datos no pueden ser mutables.
- **Composición de funciones**: los lenguajes funcionales tratan las funciones como datos, por lo que se puede aplicar composición de funciones, esto es, se puede encadenar la salida de una función con la entrada de la siguiente función.
- **Funciones de primer orden**: son funciones de nivel superior, permiten otras funciones como parámetros. Son muy populares en lenguajes como *Javascript*, se pueden definir *"llamadas de devolución"* (*callbacks*) para responder a eventos o tareas asincrónicas, pero también son un aspecto importante de los lenguajes funcionales.

Los lenguajes imperativos pueden producir efectos secundarios en elementos externos. También tienen funciones, al igual que la programación funcional, pero en este caso las funciones no son definiciones matemáticas, sino un grupo de instrucciones que se pueden llamar desde diferentes partes del programa. Teniendo en cuenta los posibles efectos secundarios, es posible obtener resultados diferentes al llamar a una función muchas veces con los mismos argumentos. Observa el siguiente código escrito en C:

```c
int externalValue = 1;
int aFunction(int param) {
    externalValue++;
    return externalValue + param;
}
```

Si se llama a la función *aFunction(1)*, devolverá 3. Pero si se la vuelve a llamar con los mismos argumentos (*aFunction(1)*) devolverá 4 … y así sucesivamente. Este tipo de situación está prohibida en un paradigma funcional, aunque algunos lenguajes funcionales permiten cometer este "crimen".

7.2. Lenguajes funcionales

Si buscas un lenguaje funcional, debes distinguir entre:

- Lenguajes funcionales ***Pure***, esto son, lenguajes que fueron creados para seguir este paradigma, y ningún otro. En este grupo se encuentran lenguajes como *Haskell* o *Miranda*. El primero es el utilizado por Facebook para lidiar con *big data* y análisis.
- Lenguajes ***Hybrid***, esto son, lenguajes que aceptan varios paradigmas, aunque les va bien con enfoques funcionales. En este grupo se puede hablar de *Scala*, *Clojure* o *Lisp*.
- Otros lenguajes, que inicialmente eran imperativos, pero han modificado su sintaxis y estructura para aceptar algunas características funcionales. Este es el caso de algunos de los lenguajes más importantes de estos años, como *Java*, *C#*, *Javascript* …

7.3. Un ejemplo introductorio

Solo para tener una primera experiencia con el paradigma funcional, observa cómo resolver un problema típico. Se utilizará un lenguaje popular, como Java, para ilustrarlo. Se dispone de una lista de objetos *Person*, cada uno con su propio nombre y edad. Se quiere obtener solo las personas que son adultas (es decir, su edad es mayor o igual a 18), se haría de esta manera con la programación tradicional e imperativa:

```
List<Person> adultPeople = new ArrayList<>();

for (int i = 0; i < people.size(); i++) {
    if (people.get(i).getAge() >= 18)
        adultPeople.add(people.get(i));
}
```

Lo que se hace es explorar la lista original, y añadir cada persona mayor de 18 años. Pero, si se aprovechan algunas de las nuevas características que proporciona Java 8 en cuanto a programación funcional, se puede resolver el mismo problema así:

```
List<Persona> adultPeople = people.stream()
        .filter(p -> p.getAge() >= 18)
        .collect(Collectors.toList());
```

Como puedes ver, el código es más compacto, menos propenso a errores y (una vez que te acostumbras a la sintaxis), más comprensible. También puedes ver cómo funciona la composición de funciones: la salida del método *stream()* es la entrada para el método *filter()*, y la salida de este método, la entrada para *collect()*.

7.4. Programación funcional en Java

En cuanto a Java, ha añadido algunas características de programación funcional, especialmente desde la versión 8, que permiten operar con datos (casi) inmutables, y encadenar operaciones a través de la composición de funciones. Hablamos principalmente de expresiones lambda y flujos (*streams*).

8. Expresiones lambda

Desde Java 8, las expresiones lambda, también llamadas funciones anónimas en muchos lenguajes, son una forma fácil y rápida de implementar un método de interfaz sin tener que crear una nueva clase para hacerlo.

8.1. Un primer ejemplo: java.io.FileFilter

Observa un ejemplo. Se dispone de esta **interfaz funcional** (interfaz con un solo método a implementar) disponible en el *core* de Java para manejar archivos: *java.io.FileFilter*.

```
public interface FileFilter {
    boolean accept(File file);
}
```

Se implementará esta interfaz para aceptar solo archivos Java (es decir, archivos con la extensión .java). Se verá cómo hacer esto en diferentes versiones de Java, para que puedas comparar la evolución de estos patrones de programación.

8.1.1. Implementación antes de Java 7: clase normal

Antes de Java 7, se tenía que crear una nueva clase para cada nueva implementación que se quería definir. Por ejemplo:

```
public class JavaFileFilter implements FileFilter {
    @Override
    public boolean accept(File file) {
        return file.getName().endsWith(".java");
    }
}
```

Y luego se utilizará así:

```
File dir = new File("/home/javier");
File[] javaFiles = dir.listFiles(new JavaFileFilter());
```

8.1.2. Implementación en Java 7: clase anónima

Además de usar clases normales, desde Java 7 se pueden crear **clases anónimas** implementando interfaces, o extendiendo clases abstractas, cuyos métodos se implementarán cuando se cree el objeto (se podrían hacer diferentes implementaciones cada vez).

```
File dir = new File("/home/javier");

File[] javaFiles = dir.listFiles(new FileFilter() {
    @Override
    public boolean accept(File file) {
        return file.getName().endsWith(".java");
    }
});
```

8.1.3. Implementación desde Java 8: expresiones lambda

Además de las clases normales y anónimas, desde Java 8, cuando estés implementando una interfaz con un solo método (interfaz funcional), podrás hacerlo con menos código gracias a las expresiones lambda:

```
File dir = new File("/home/javier");
File[] javaFiles = dir.listFiles((File file) -> file.getName().endsWith(".java"));
```

No es necesario especificar que es una interfaz *FileFilter* lo que se está implementando, ya que el compilador sabe que el método ***listFiles()*** necesita un objeto *FileFilter* derivado como argumento. No es necesario utilizar la palabra ***return*** porque el compilador la asumirá. Incluso se puede omitir el tipo de parámetro ya que el compilador puede verlo en la definición de la interfaz. Entonces, la expresión lambda puede ser aún más simple, así:

```
File[] javaFiles = dir.listFiles(file -> file.getName().endsWith(".java"));
```

8.2. Otro ejemplo: java.util.Comparator

El interfaz *Comparator* del paquete *java.util* es otra interfaz funcional. Tiene un solo método llamado *compare* que toma dos objetos como parámetros y los compara devolviendo un número entero que indica qué objeto es mayor. Oberva cómo implementar este comparador para comparar dos objetos *String* según su longitud.

8.2.1. Implementación antes de Java 7

Si estás usando Java 6 o versiones anteriores, necesitarás definir una clase normal que implemente la interfaz, y luego crear un objeto de esta clase y usarlo siempre que se necesite comparar las cadenas. Por ejemplo:

```
public class MyStringComparator implements Comparator<String> {
    @Override
    public int compare(String s1, String s2) {
        return Integer.compare(s1.length(), s2.length());
    }
}

// MAIN
List<String> list = Arrays.asList("Hello", "Hi", "Goodbye", "Farewell", "Bye");

MyStringComparator msc = new MyStringComparator();
…
Collections.sort(list, msc);
```

8.2.2. Implementación en Java 7

Si utilizas Java 7, también podrás usar una clase anónima para implementar la interfaz, de esta manera:

```
// MAIN
List<String> list = Arrays.asList("Hello", "Hi", "Goodbye", "Farewell", "Bye");

Comparator<String> comp = new Comparator<String>() {
    @Override
    public int compare(String s1, String s2) {
        return Integer.compare(s1.length(), s2.length());
    }
};

Collections.sort(list, comp);
```

8.2.3. Implementación en Java 8

Finalmente, si se está utilizando Java 8, también podrás usar una expresión lambda. En este caso, el método a implementar tiene dos parámetros, por lo que se definen ambos en el paréntesis de la expresión lambda:

```
Comparator<String> lComp = (s1,s2) -> Integer.compare(s1.length(), s2.length());

List<String> list = Arrays.asList("Hello", "Hi", "Goodbye", "Farewell", "Bye");

Collections.sort(list, lComp);
```

8.3. Un ejemplo más: java.lang.Runnable

Utilizarás esta interfaz en la Unidad 2, cuando se hable de sub-procesos y programación multi-proceso. Solo tiene un método para implementar, sin parámetros ni tipo de retorno. Se utiliza en objetos *Thread* para crearlos con un parámetro de tipo *Runnable* (se verán estos conceptos en profundidad en la Unidad 2).

8.3.1. Implementación antes de Java 7

En versiones anteriores a Java 7, como es habitual, se necesitará definir una clase que implemente la interfaz y luego usar un objeto de esta clase:

```
public class MyRunnable implements Runnable {
    @Override
    public void run() {
        for(int i = 0; i < 3; i++) {
            System.out.println("This is thread: " +
                Thread.currentThread().getName());
        }
    }
}

// MAIN
Runnable run = new MyRunnable()
Thread t = new Thread(run);
t.start();
```

8.3.2. Implementación en Java 7

En Java 7, se puede definir una clase anónima siempre que se quiera implementar la interfaz:

```
// MAIN
Runnable run = new Runnable() {
    @Override
    public void run() {
        for(int i = 0; i < 3; i++) {
            System.out.println("This is thread: " +
                Thread.currentThread().getName());
        }
    }
};

Thread t = new Thread(run);
t.start();
```

8.3.3. Implementación en Java 8

Si se utiliza Java 8, también se podrá usar una expresión lambda. En este caso, como el método no tiene parámetros, se dejan los paréntesis de la expresión lambda vacíos (deben escribirse los paréntesis de todos modos):

```
Runnable lambdaRun = () -> {
    for(int i = 0; i < 3; i++) {
        System.out.println("This is thread: " +
            Thread.currentThread().getName());
    }
};

Thread t = new Thread(lambdaRun);
t.start();
```

Deberás tener en cuenta que, en este ejemplo, el código necesita más de una sentencia, por lo que se deben usar las llaves {…} después de la flecha de la expresión lambda.

8.4. Conclusiones

Cuando se quiere utilizar una expresión lambda, solo hay que centrarse en el método implementado y comprobar:

- Los parámetros de entrada del método.
- El valor devuelto (si lo hubiera).

Después, en el lugar de la aplicación donde quieras usar la expresión lambda, se definirá un objeto de la interfaz dada. Los parámetros de entrada deben colocarse entre paréntesis (a menos que solo haya un parámetro), a continuación, se escribe una flecha (->) seguida del código del método (entre {…} si tiene más de una instrucción).

Recuerda la forma en que se ha creado la expresión lambda para *Comparator* (dos parámetros, un *String* devuelto):

```
Comparator<String> lComp = (s1,s2) -> Integer.compare(s1.length(), s2.length());
```

y la forma en que se ha creado la expresión lambda para *Runnable* (sin parámetros ni tipo de retorno):

```
Runnable lambdaRun = () -> {
    for(int i = 0; i < 3; i++) {
        System.out.println("This is thread: " +
            Thread.currentThread().getName());
    }
};
```

8.4.1. Algunos conceptos más que tener en cuenta

Existe una forma de acortar, aún más, una expresión lambda. Cuando tiene una llamada a un método que necesita los mismos parámetros que la expresión lambda y en el mismo orden, se puede escribir una referencia a ese método, omitiendo incluso los parámetros, usando esta sintaxis especial:

```
// Normal version
Comparator<Integer> comp = (i1, i2) -> Integer.compare(i1, i2);
// Shorter version
Comparator<Integer> comp2 = Integer::compare;

// Normal version
Consumer<String> con = s -> System.out.println(s);
// Shorter version
Consumer<String> con2 = System.out::println;
```

Se mencionó al principio de esta sección que las expresiones lambda también se denominan funciones anónimas. Si se utiliza una clase anónima para implementar una interfaz, entonces el compilador crea un objeto completo para manejar sus métodos. Sin embargo, cuando se trabaja con expresiones lambda, el compilador no crea objetos completos, sino que crea un tipo de objeto especial y más ligero, por lo que suele ser una mejor opción.

Ejercicio 6

Crea un proyecto llamado ***ListFilter*** que incluya lo siguiente:

- Una clase llamada ***Student*** que tenga estas propiedades (*getters*/*setters* cuando sea necesario):
  - *Name*
  - *Age*
  - Lista de temas (*subjects*, tipo *String*)
- En la clase principal, el método *main* y un método estático llamado:

```
    List<Student> filterStudents(List<Student> srcList, Predicate<Student> predicate)
```

El método ***filterStudents()*** recibirá una lista de estudiantes y devolverá otra lista solo con los elementos que cumplan la condición definida en *Predicate*. Un predicado es una interfaz funcional que necesita implementar un método (***boolean test(T t)***). Impleméntalo utilizando expresiones lambda.

En el método principal tendrás que crear una lista de al menos 8 estudiantes, y luego, usando el método *filterStudents()*, genera otras 3 listas que solo contengan estudiantes que:

1. Tengan más de 20 años.
2. Están inscritos en la asignatura “Programación”.
3. Su nombre contiene “Peter”.

9. Streams

Un flujo (*stream*) es un nuevo concepto diseñado para procesar grandes (o pequeñas) cantidades de datos, utilizando para ello un nuevo nivel de abstracción en combinación con expresiones lambda que, pueden ser fácilmente *“paralelizables”* para aprovechar todos los núcleos de CPU disponibles, sin tener que hacerlo manualmente (dividir el cálculo y crear nuevos hilos).

Un *stream*:

- Es un objeto especial en el que se pueden definir operaciones (usando expresiones lambda, generalmente).
- No contiene ningún dato (actúa como intermediario).
- No modifica los datos que procesa. El compilador le permite hacerlo, pero no debería hacer, puede producir un comportamiento impredecible.
- Procesa todos los datos de una operación y los pasa a la siguiente operación. No hace nada hasta que se llama a las operaciones ***final*** o **terminal** (se verá qué es una operación final).
- Se comporta, de alguna manera, similar a SQL.

Los flujos (*streams*) son un concepto realmente poderoso (y quizás difícil de entender) de Java 8. Son útiles cuando se quiere filtrar y procesar una lista de gran tamaño o una cantidad de datos importante.

9.1. Stream vs Collection

En primer lugar, por problemas de compatibilidad, el *framework Collection* no se ha modificado para que funcione como lo hacen los s*treams*, por lo que son cosas diferentes. La mayoría de las veces, los *streams* se generan a partir de colecciones y en ocasiones, generan una nueva colección como resultado, pero no son iguales.

Con *Collection*, el programador debe iterar y operar con todos sus valores manualmente, y si quiere *"paralelizar"* la operación, debe crear los hilos necesarios y dividir el problema por sí mismo. Con un flujo, solo tiene que definir las operaciones que se realizarán con todos los datos, y automáticamente iterará y aplicará las operaciones definidas, además, también dividirá el problema y generará sub-procesos cuando utilice flujos paralelos.

9.2. Operaciones Intermediary o lazy

Pueden hacerse dos tipos de operaciones con *streams*: una de ellas son las operaciones ***Intermediary***. Con estas operaciones no se ejecuta nada, devuelven otro *stream*, para que se pueda encadenar tantas como se quiera. Las operaciones *Intermediary* más típicas son los filtros y mapeos.

9.2.1. Filtros

Una operación de filtrado es una operación intermedia. Toma un predicado como argumento, lo que significa que aceptará datos que coincidan con la condición y rechazará los que no. Por ejemplo, si se tiene una lista de objetos *Person*:

```
List<Person> persons = new ArrayList<>(10);
persons.add(new Person(16, "Peter"));
persons.add(new Person(22, "Mary"));
persons.add(new Person(43, "John"));
persons.add(new Person(70, "Amy"));
```

Se puede obtener un flujo de esa lista y filtrar aquellos objetos cuya edad sea mayor de 18 años. Luego, se puede explorar el flujo e imprimir los objetos seleccionados:

```
Stream<Person> stream = persons.stream();
// Intermediary
Stream<Person> stream2 = stream.filter(p -> p.getAge() >= 18);
// Final (processes all)
stream2.forEach(p -> System.out.println(p.toString()));

// A shorter way to do exactly the same
persons.stream().filter(p -> p.getAge() > 18).forEach(System.out::println);
```

El parámetro utilizado en el método *filter()* es un predicado, una interfaz funcional proporcionada por Java 8. Tiene un método llamado *test()* que devuelve un valor booleano que indica si una prueba determinada es verdadera o falsa. En este caso, se utiliza una expresión lambda para implementar este predicado, y la prueba que se debe pasar es que cada persona *p* seleccionada debe tener más al menos 18 años (`p.getAge() >= 18`).

9.2.2. Operaciones de mapeo

Las operaciones de mapeo también son operaciones intermediarias. Usan una interfaz *Function* (otra interfaz funcional traída por Java 8), tomando un elemento (*input*) y devuelve una salida diferente. Por ejemplo, puede utilizarse para extraer las edades de las personas del ejemplo anterior:

```
persons.stream()
    .filter(p -> p.getAge() >= 18)
    .map(p -> p.getAge())
    .forEach(System.out::println); // Prints only ages
```

9.3. Operaciones finales

Otro tipo de operaciones que pueden hacerse con los *streams* son las operaciones finales. Estas cierran el flujo y obtienen un tipo de objeto (por ejemplo, *Collection*) o un valor final (por ejemplo, un promedio o máximo). Una cosa importante a tener en cuenta es que, un *stream* solo puede tener **una operación final**. Una vez que ha procesado los datos, la transmisión no se puede reutilizar, por lo que se debe crear otro *stream*.

9.3.1. Operaciones de reducción

Las reducciones devuelven un elemento que puede ser de cualquier tipo. A veces, no devuelven nada (por ejemplo, el elemento mínimo de una lista vacía), y eso sería un problema si se asigna ese resultado a una variable. Se puede usar la clase ***Optional*** para administrar este tipo de situaciones.

Tipos de reducciones:

- Returns Optionals
  - *max(), min()*
  - *findAny(), findFirst()*
- Returns long
  - *count()*
- Returns boolean
  - *allMatch(), noneMatch(), anyMatch()*
- Other (el valor de retorno depende de la operación realizada)
  - *reduce()*

Observa un ejemplo: si quieres sumar las edades de todos los objetos *Person* mayores de 18 años, se podría usar el método *reduce()*:

```
int sumAges = persons.stream()
    .filter(p -> p.getAge() > 18)
    .map(p -> p.getAge())
    .reduce(0, (a,b) -> a + b);
```

El primer argumento del método *reduce()* es el caso base, el valor que se tomará si la lista está vacía. Después, para cada edad encontrada en la lista filtrada, se añade a este caso base.

Otras operaciones como *max()*, no necesitan tomar un valor base, por lo que devolverán un ***Optional*** (en caso de que la lista esté vacía, no haya resultado). Aquí se obtiene la edad máxima de todos los objetos filtrados:

```
Optional<Integer> maxAge = persons.stream()
    .filter(p -> p.getAge() > 18)
    .map(p -> p.getAge())
    .reduce(Integer::max);

// Will print Optional[70]
System.out.println(maxAge);
// Better
System.out.println(maxAge.isPresent()?maxAge.get():"No max age");
```

Con *mapToInt()*, *mapToDouble()*, etc … los valores se convierten en tipos primitivos y pueden hacerse más operaciones y más simples. Como calcular el promedio de todas las edades (mayores de 18 años):

```
OptionalDouble avgAge = persons.stream()
    .filter(p -> p.getAge() > 18)
    .mapToInt(p -> p.getAge()).average();

System.out.println(avgAge.isPresent()?"Avg: " + avgAge.getAsDouble():"No ages");
```

Otro ejemplo: se pregunta si hay algún objeto *Person* menor de 18 años:

```
if(persons.stream().anyMatch(p -> p.getAge() < 18)) {
    System.out.println("There are children in the collection!");
}
```

9.3.2. Operaciones de recolección

Este tipo de operaciones también son definitivas. En lugar de devolver un objeto, un valor primitivo o nada, normalmente devuelven una *Collection*, o a veces un *String* (operación de unión). El método utilizado para estas operaciones es ***collect()***.

El uso más básico es pasar un parámetro, un objeto ***Collector***, que se puede crear a partir de la clase ***Collectors***. En este ejemplo se devuelve un *string* con todos los nombres de las personas unidos por comas:

```
String names = persons.stream()
    .filter(p -> p.getAge() >= 18)
    .map(p -> p.getName())
    .collect(Collectors.joining(",", "Adults: ", ""));

System.out.println(names); // Adults: Mary,John,Amy
```

Se pueden generar nuevas listas:

```
List<Person> older = persons.stream()
    .filter(p -> p.getAge() >= 18)
    .collect(Collectors.toList());
```

Se puede devolver un mapa usando *Collectors.groupingBy(key, calculated value)*. Por ejemplo, en el siguiente ejemplo, se obtiene un mapa con todas las edades y la cantidad de personas que tiene cada edad:

```
Map<Integer, Long> ages = persons.stream()
    .filter(p -> p.getAge() >= 18)
    .collect(Collectors.groupingBy(
            p -> p.getAge(),
            Collectors.counting()
        )
    );

System.out.println(ages.toString()); // {70=1, 22=1, 43=1}
```

Con un solo parámetro, *groupingBy* devuelve una lista de elementos agrupados por cada clave.

```
Map<Integer, List<Person>> ages = persons.stream()
    .filter(p -> p.getAge() >= 18)
    .collect(
        Collectors.groupingBy(
            p -> p.getAge()
        )
    );

System.out.println(ages.toString());
// {70=[Name: Amy, age: 70], 22=[Name: Mary, age: 22], 43=[Name: John, age: 43]}
```

También puede optarse por generar una lista diferente pasando una función como esta:

```
Map<Integer, List<String>> ages = persons.stream()
    .filter(p -> p.getAge() >= 18)
    .collect(
        Collectors.groupingBy(
            p -> p.getAge(),
            Collectors.mapping(
                p -> p.getName(), // Insert only the name
                Collectors.toList()
            )
        )
    );

System.out.println(ages.toString()); // {70=[Amy], 22=[Mary], 43=[John]}
```

9.4. Conclusión

En resumen, se verá un ejemplo de cómo hicimos el mismo tipo de operaciones con Java 7 y la diferencia con el uso de *streams* y expresiones lambda en Java 8:

```
// Store the names of adult people sorted by age. JAVA 7
List<Person> older = new ArrayList<>();
for(Person p : persons) {
    if(p.getAge() >= 18)
        older.add(p);
}

Collections.sort(older, new Comparator<Person>() {
    public int compare(Person p1, Person p2) {
        return Integer.compare(p1.getAge(), p2.getAge());
    }
});

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

```java
for(Person p : older)
    namesOlder.add(p.getName());

// Store the names of adult people sorted by age. JAVA 8
List<String> namesOlder = persons.stream()
    .filter(p -> p.getAge() >= 18)
    .sorted(Comparator.comparing(Person::getAge))
    .map(Person::getName)
    .collect(Collectors.toList());

// Could use: .sorted((p1, p2) -> Integer.compare(p1.getAge(),
// p2.getAge())) and .map(p -> p.getName())
```

Ejercicio 7

Usando el proyecto ***ListFilter*** del ejercicio anterior, añade algunas cosas nuevas:

1. En la clase *Main*, añade un nuevo método estático llamado ***getOldestNames (List<Student> list)*** que devuelva ***List<String>***. En el interior, deberás crear un flujo a partir de la lista recibida como parámetro y generar una lista con los nombres de las tres personas más viejas de la lista (utiliza *limit(3)* después de ordenar). Pruébalo y muestra los resultados en consola para ver si funciona bien.
2. Crea otro método estático clase principal llamado ***getAllSubjects (List<Student> list)*** que devuelva un ***Set<String>***. En este método, deberás crear un *stream* a partir de la lista de estudiantes pasada y generar una lista (*Set*) que contenga todas las materias en las que haya al menos un estudiante (o más) de la lista, ordenados alfabéticamente. Pruébelo pasando alguna lista y mostrando los resultados en la consola.