

Programación

Programación Funcional

(lambda - streams)

Unidad 15

Jesús Alberto Martínez
versión 0.2





Reconocimiento – NoComercial – CompartirIgual (by-nc-sa): No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original.
Basado en los apuntes de WirzJava y del CEEDCV




Nomenclatura

A lo largo de este tema se utilizarán distintos símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:

 Importante

 Atención

 Interesante

Unidad 15. Programación funcional

- 1 Introducción..... 3
- 2 Interfaces Funcionales..... 3
 - 2.1 Predicate, Consumer, Function y Supplier..... 6
- 3 Funciones Lambda..... 8
 - 3.1 Predicate, Consumer, Function y Supplier..... 10
 - 3.2 Referencias a métodos..... 12
- 4 API Stream..... 13
 - 4.1 Tratar Ficheros como Streams..... 16

1 Introducción

La forma de programar que hemos seguido a lo largo de este manual se denomina “programación imperativa” ya que se basa en detallar todos los pasos ordenados para la resolución de un problema.

En ciertas ocasiones podremos emplear una forma de programar más orientada a lo que queremos obtener y no al detalle de pasos para obtenerlo, ese estilo de programación se llama “declarativa”.

Formalmente, la programación declarativa se define, en contraposición a la programación imperativa, como un paradigma de programación basado en el desarrollo de programas especificando o “declarando” un conjunto de condiciones, proposiciones, afirmaciones, restricciones, ecuaciones o transformaciones que describen el problema y detallan su solución. La solución es obtenida mediante mecanismos internos de control, sin especificar exactamente cómo encontrarla.

Un ejemplo de lenguaje puramente declarativo que conocéis bien es SQL. La programación funcional es un tipo de programación declarativa, basada en el uso de funciones, dotándolas de mayor versatilidad. Java, desde su versión 8, permite la programación funcional empleando funciones Lambda (a través de las Interfaces Funcionales) y el API Stream.

Como ejemplo de lo que vamos a aprender en este tema, supongamos el siguiente enunciado: Crea un conjunto (Set) a partir de una lista llamada ‘números’, que contenga el resultado de elevar al cuadrado todos sus elementos que sean pares. En programación imperativa necesitaríamos un “for” para recorrer cada elemento de la colección, dentro del bucle un “if” para verificar la condición de si el número es par o no, y en caso afirmativo calcular el cuadrado y a continuación guardar dicho valor en un conjunto. Todo ello, con programación funcional se reducirá a:

```
Set cuadradosPares = numeros.stream()
    .filter(x -> x%2==0)
    .map(x->x*x)
    .collect(Collectors.toSet());
```

Pero para llegar entender este código basado en Streams , tenemos que ver previamente lo que son las interfaces funcionales y las funciones Lambda.

2 Interfaces Funcionales

Las interfaces definían métodos que suponían un “un compromiso” que debían cumplir las clases que la implementaban. Adicionalmente podían tener métodos estáticos, métodos por defecto y métodos privados.

Una interfaz funcional solo puede tener un método abstracto, pudiendo tener métodos por defecto, privados o estáticos. En realidad, se podría matizar esta definición; una interfaz funcional puede tener

más de un método abstracto, pero todos menos uno deben ser sobrescritura de métodos de la clases Object (por ejemplo: toString(), equals() , etc.).

La anotación @FunctionalInterface no es obligatoria, pero comprueba en tiempo de compilación si se cumplen las condiciones que comentamos. Este sería un ejemplo:

```
@FunctionalInterface
interface ICalculadora {
    public double calcular (int a, int b);
}
```

Como ya sabemos de capítulos anteriores, necesitaremos crear una clase que implemente la interfaz y por tanto los métodos definidos en ella (salvo los métodos default, estáticos o privados):

```
class Sumador implements ICalculadora {
    @Override
    public double calcular(int i, int j) { return (double) i+j;}
}
```

Podríamos crear más clases que implementasen esa interfaz, como un Multiplicador que desarrollaría el código del método de forma distinta.

Finalmente, deberemos crear instancias de la definida e invocar a su método:

```
ICalculadora s = new Sumador(); //también: Sumador s = new Sumador();
System.out.printf("%f\n",s.calcular(3,5));
```

También vimos que las interfaces se podían implementar con una clase anónima, que en un solo paso y sin crear la clase explícitamente, crea la instancia de la clase y sobrescribe el método abstracto definido en la interfaz:

```
ICalculadora s = new ICalculadora () { //no creamos la clase Sumador
    @Override
    public double calcular(int i, int j) {
        return (double) i+j;
    }
};
System.out.printf("%f\n",s.calcular(3,5));
```

Un ejemplo de interfaz funcional es Comparator, vista también en capítulos anteriores. Tiene un solo método abstracto: compare (Object o1, Object o2). Esta interfaz la usábamos para ordenar colecciones por distintos atributos. El método Collections.sort () recibe como primer parámetro la lista a ordenar y como segundo parámetro una instancia de una clase que implemente Comparator.

Teníamos una clase Peli:

```
class Peli {
    public String nombre;
    public int año;
    public Peli(String n, int a) {
        this.nombre = n;
        this.año = a;
    }
}
```

Y una lista de Peli:

```
List<Peli> lista = new ArrayList<>();
lista.add(new Peli("Episode 7: The Force Awakens", 2015));
lista.add(new Peli("Episode 4: A New Hope", 1977));
lista.add(new Peli("Episode 1: The Phantom Menace", 1999));
```

Podíamos ordenar la lista por el atributo que deseásemos. Para ello creábamos una clase que implementase Comparator y su método compare (). Este método tenía que devolver entero positivo si el primer parámetro era mayor que el segundo según el criterio de ordenación deseado, entero negativo si el segundo parámetro era mayor, o cero si ambos parámetros eran iguales.

Versión con clase anónima:

```
Collections.sort(lista, new Comparator() {
    @Override
    public int compare(Object o1, Object o2) {
        Peli p1 = (Peli) o1;
        Peli p2 = (Peli) o2;
        return p1.año - p2.año;
    }
});
```

Versión sin clase anónima:

```
class ComparaAño implements Comparator {
    public int compare(Object o1, Object o2) {
        Peli p1 = (Peli) o1;
        Peli p2 = (Peli) o2;
        return p1.año - p2.año;
    }
}
ComparaAño compAño = new ComparaAño();
Collections.sort(lista, compAño);
```

Comprobamos como el usar clases anónimas reduce la cantidad de código.

Otro aspecto interesante de las interfaces (no solo de las funcionales) era que podíamos definir las sobre tipos genéricos, con lo que aún podemos reducir más el código, omitiendo los castings:

```
Collections.sort(lista, new Comparator < Peli > () {
    @Override
    public int compare(Peli p1, Peli p2) {
        return p1.año - p2.año;
    }
});
for (Peli p: lista) System.out.println(p.nombre + ":" + p.año);
```

⚡ Es importante entender todo lo anterior para seguir avanzando en este tema.

2.1 Predicate, Consumer, Function y Supplier

Existen de forma predefinida una serie de interfaces funcionales que permiten realizar un gran abanico de operaciones y que serán fundamentales para trabajar con Streams como veremos más adelante y encajarán perfectamente con el uso de expresiones Lambda.

Esas interfaces funcionales son las siguientes: Predicados, Consumidores, Funciones y Proveedores.

Int. Funcional	Método abstracto	Descripción
Predicate	boolean test (T t)	Devuelve true como resultado de evaluar el parámetro pasado. Tiene otros métodos (default) para combinar con éste: or (), and (), negate ().
Consumer	void accept (T t)	Sirve para “consumir” los datos recibidos, por ejemplo, mostrarlos por pantalla. Tiene el método default: andThen() para encadenar con otro consumer.
Function (*)	R apply (T t)	Sirve para transformar un objeto. Recibe un parámetro de un tipo y devuelve otro de un tipo diferente. Tiene los métodos default: andThen(), compose() y identity().
Supplier	T get ()	No recibe parámetros y sirve para obtener objetos. Hay interfaces especializados como IntSupplier, LongSupplier, DoubleSupplier, BooleanSupplier.

(*) Existe BiFunction, para usar si necesitamos recibir dos parámetros

Vamos a ver unos ejemplos. Para su uso es necesario incorporar los siguientes import:

```
import java.util.function.Predicate;
import java.util.function.Consumer;
import java.util.function.Function;
import java.util.function.Supplier;
```

Aunque ahora no le veamos mucha utilidad, luego, con las expresiones Lambda y la API Stream veremos cómo el uso de estas interfaces funcionales simplifica mucho el código.

Ejemplo Predicate:

```
Predicate < String > cadLarga = new Predicate < > () {
    @Override
    public boolean test(String s) {
        if (s.length() > 8) return true;
        return false;
    }
};
if (cadLarga.test("123456789")) System.out.println("Es larga");
else System.out.println("No Es larga");
```

Ejemplo Consumer:

```
Consumer < String > cadena = new Consumer < > () {
    @Override
    public void accept(String s) {
        System.out.println(s);
    }
};
cadena.accept("123456789");
```

Ejemplo Function:

```
Function < Integer, Long > cuadrado = new Function < > () {
    @Override
    public Long apply(Integer a) {
        return (long) a * a;
    }
};
System.out.println(cuadrado.apply(10));
```

Ejemplo Supplier:

```
Random random = new Random();
Supplier < Integer > aleatorio = new Supplier < > () {
    Random random = new Random();
    @Override
    public Integer get() {
        Random random = new Random();
        int n = random.nextInt(10);
        return n;
    }
};
System.out.println(aleatorio.get());
```

3 Funciones Lambda

A continuación, veremos lo que son las funciones Lambda, pero ya podemos adelantar que allá donde haya una interfaz funcional, podremos emplear una expresión Lambda para definir su método abstracto, de una forma más intuitiva y con menos código que con la programación imperativa.

Las expresiones lambda son funciones anónimas, sin nombre cuya sintaxis básica se detalla a continuación:

```
( parámetros ) -> { cuerpo de la función }
```

El operador lambda (->) separa la declaración de parámetros de la declaración del cuerpo de la función.

Parámetros:

- Cuando se tiene un solo parámetro no es necesario utilizar los paréntesis.
- Cuando no se tienen parámetros, o cuando se tienen dos o más, es necesario utilizar paréntesis.
- Los argumentos de una función Lambda pueden ser declarados explícitamente o a su vez pueden ser inferidos por el compilador de acuerdo al contexto. Veremos más adelante como hace esto. Ejemplos:
 - (int x) -> {cuerpo}
 - (int x, int y,...) -> {cuerpo}.
 - x -> {cuerpo}
 - () -> {cuerpo}.

Cuerpo de lambda:

- Cuando el cuerpo de la expresión lambda tiene una única línea no es necesario utilizar las llaves y no necesitan especificar la cláusula return en el caso de que deban devolver valores.
- Se hace un return implícito del valor calculado en esa única línea. Cuando el cuerpo de la expresión lambda tiene más de una línea se hace necesario utilizar las llaves y es necesario incluir la cláusula return en el caso de que la función deba devolver un valor. Ejemplos:
 - `z -> z + 2`
 - `() -> System.out.println(" Mensaje 1")`
 - `n -> {System.out.print(n + " ");}`
 - `(int longitud, int altura) -> { return altura * longitud; }`
 - `(String x) -> {`
 `String retorno = x;`
 `retorno = retorno.concat(" ***");`
 `return retorno; }`
 - `(int a, int b) -> { return a + b; }`
 - `() -> { return 3.1415 }`

Usando la función Lambda:

Las funciones Lambda no se pueden invocar directamente en el código, sino que precisan estar definidas como implementación del método abstracto de una interfaz funcional. Así pues, en el ejemplo creado en el apartado anterior:

```
@FunctionalInterface
interface ICalculadora {
    public double calcular (int a, int b);
}
```

Podríamos implementar (sobrescribir) el método calcular() mediante una Lambda. Así lo habíamos hecho sin Lambda, con una clase anónima:

```
ICalculadora s = new ICalculadora () {    //no creamos la clase Sumador
    @Override
    public double calcular(int i, int j) {
        return (double) i+j;
    }
}
```

```
};
```

Y así lo haríamos con la función Lambda en clase anónima:

```
ICalculadora s = (x,y) -> x+y;          //return implícito
```

En cualquiera de los dos casos, hay que invocar al método:

```
System.out.printf("%f%n", s.calcular(3, 5));
```

Vemos como no es necesario hacer new ni especificar el método que sobrescribe, ya que solo hay uno. Tampoco es necesario indicar los parámetros porque se infieren de la definición de la interfaz.

El otro ejemplo visto de interfaz funcional, de Comparator:

```
Collections.sort(lista, new Comparator <Peli> () {
    @Override
    public int compare(Peli p1, Peli p2) {
        return p1.año - p2.año;
    }
});
```

Quedaría así con expresiones Lambda:

```
Collections.sort(lista, (p1, p2) -> p1.año - p2.año);
```

3.1 Predicate, Consumer, Function y Supplier

Como acabamos de comentar, las funciones Lambda están vinculadas a una interfaz funcional y vimos que Java tiene predefinidas una serie de ellas (Predicate, Consumer, Function y Supplier) que nos permiten realizar muchas tareas.

Vamos a reescribir los ejemplos de estas interfaces vistas en el apartado anterior, ahora empleando funciones Lambda.

Predicate sin Lambda

```
Predicate <String> cadLarga = new Predicate <>() {
    @Override
    public boolean test(String s) {
        if (s.length() > 8) return true;
        return false;
    }
};
```

Predicate con Lambda

```
Predicate<String> cadLarga = (s) -> s.length() > 8;
```

Consumer sin Lambda

```
Consumer <String> cadena = new Consumer <>() {  
    @Override  
    public void accept (String s) {  
        System.out.println(s);  
    }  
};
```

Consumer con Lambda

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

Function sin Lambda

```
Function <Integer, Long> cuadrado = new Function <>() {  
    @Override  
    public Long apply (Integer a) {  
        return (long)a * a; }  
};
```

Function con Lambda

```
Function<Integer, Long> cuadrado = a -> (long) a * a;
```

Supplier sin Lambda

```
Random random = new Random();  
Supplier <Integer> aleatorio = new Supplier <> () {  
    Random random = new Random();  
    @Override  
    public Integer get () {  
        Random random = new Random();  
        int n = random.nextInt(10);  
        return n; }  
};
```

Supplier con Lambda

```
Random random = new Random();
```

```
Supplier <Integer> aleatorio = () -> random.nextInt(10);
```

3.2 Referencias a métodos

Con las referencias a métodos no solo se puede utilizar expresiones lambda para implementar la interfaz funcional sino que se puede hacer referencia a los métodos del objeto utilizando el operador double colon, dos puntos dobles, :: y sustituyen a una expresión lambda.

Nr	Method Reference Type	Method Reference	Lambda expression
1	Static method	String::valueOf	(int i) -> String.valueOf(i)
2	Instance method of a particular object	s::substring	(int beg, int end) -> s.substring(beg, end)
3	Instance method of an arbitrary object	String::equals	(String s1, String s2) -> s1.equals(s2)
		JLabel::getIcon	(JLabel lb) -> lb.getIcon()
4	Constructor	String::new	() -> new String()

Es muy típico ver la referencia de métodos en las colecciones, en su método forEach, que recibe una interfaz funcional representada mediante esta referencia de métodos.

⚡ No confundir con el bucle for each que llevamos utilizando todo el curso

```
ArrayList1.forEach(System.out::println);
```

También resulta muy visual la referencia al método estático comparing () de la interfaz funcional Comparator que devuelve un comparador implementando el método compare () sobre el atributo pasado.

```
class Pelı {
    private String nombre;
    private int año;
    public Pelı(String n, int a) {
        this.nombre = n;
        this.año = a;
    }

    public int getAño() {
        return año;
    }
}
```

```

    }
    public String getNombre() {
        return nombre;
    }
    @Override
    public String toString() {
        return nombre + " (" + año + ")";
    }
}
List < Peli > lista = new ArrayList < > ();
lista.add(new Peli("Episode 7: The Force Awakens", 2015));
lista.add(new Peli("Episode 4: A New Hope", 1977));
lista.add(new Peli("Episode 1: The Phantom Menace", 1999));
Collections.sort(lista, Comparator.comparing(Peli::getAño));
lista.forEach(System.out::println);

```

4 API Stream

A través del API Stream podemos trabajar sobre colecciones de una manera limpia y clara, evitando bucles y algoritmos que ralentizan los programas y hacen complejo entender su funcionalidad.

Existen 3 partes que componen un Stream que de manera general serían:

1. Un Stream funciona a partir de una lista o colección, que también se la conoce como la fuente de donde obtienen información. El método `stream()` convertirá la colección en Stream para comenzar su tratamiento.
2. Operaciones intermedias que actúan sobre el Stream. Estas son solo algunas:
 - `map`: Obtiene un nuevo Stream resultado de aplicarle una función a cada elemento (en general una expresión Lambda que implementa la interfaz funcional `Function`).
 - `filter`: selecciona elementos según la expresión pasada como parámetro que será una implementación de la interfaz funcional `Predicate`, generalmente mediante una expresión Lambda.
 - `sorted`: ordena el Stream. Lo hace por el criterio por defecto de la clase (`Comparable`) o con el comparador pasado como parámetro, generalmente como expresión Lambda.
 - `skip (n)`: elimina de Stream los 'n' primeros de elementos del Stream.
 - `limit (n)`: se queda solo con los 'n' primeros elementos del Stream, eliminando los sobrantes.
3. Operaciones terminales. Son la última operación que se hace con el Stream. Ejemplos:
 - `collect`: Obtiene una colección con el resultado de los procesos previos aplicados al Stream.

- `forEach`: Itera sobre cada elemento de Stream (el método `peek()` valdría para iterar sobre el Stream como operación intermedia, no terminal).
- `reduce`: Permite realizar un cálculo sobre los elementos del Stream, utilizando un operador binario para definir la operación. Tiene muchas posibilidades, por ejemplo:
 - `reduce(0, (a,b) -> a+b)` acumula, sería como `a+=b`, empezando con `a=0`.
 - `reduce(Integer::sum)` hace lo mismo que la anterior, pero devuelve un `Optional`. Esta es una clase que almacena un valor, gestionando los valores nulos. Para obtener su valor hay que ejecutar su método `get` (luego lo veremos en los ejemplos)
 - `reduce(Integer::min)` Análogo al anterior, pero con el mínimo.
 - `reduce(Integer::max)` Análogo al anterior, pero con el máximo.
- `count`: Obtiene un `long` con la cantidad de elementos del Stream una vez procesado.
- `sum`: Obtiene la suma de los valores, previamente debemos hacer un `mapToInt(Integer::intValue)`

Para entender todas estas operaciones lo mejor es ver ejemplos en funcionamiento. Partiremos en todos los casos de una lista con los 10 primeros enteros.

```
List <Integer> numeros = Arrays.asList(1,2,3,4,5,6,7,8,9,10);
```

Ejemplo 1: Mostrar los elementos del ArrayList inicial elevados al cuadrado.

```
numeros.stream()
    .map(x->x*x)
    .forEach(y->System.out.println(y));
```

O también:

```
numeros.stream()
    .map(x->x*x)
    .forEach(System.out::println);
```

A veces, se presenta todo en una sola línea, pero se entiende peor:

```
numeros.stream().map(x->x*x).forEach(System.out::println);
```

Ejemplo 2: Obtener una nueva lista con elementos de la lista inicial elevados al cuadrado.

```
List cuadrados = numeros.stream()
    .map(x->x*x)
```

```
.collect(Collectors.toList());
```

Ejemplo 3: Obtener un Set con elementos de la lista inicial que sean pares elevados al cuadrado:

```
Set cuadradosPares = numeros.stream()
    .filter(x -> x%2==0)
    .map(x->x*x)
    .collect(Collectors.toSet());
```

Ejemplo 4: Mostrar el cuadrado de los elementos de las posiciones 4 a 8, ambas incluidas (empezando en cero).

```
numeros.stream()
    .skip(4)
    .limit(5)
    .map(x->x*x)
    .forEach(System.out::println);
```

Ejemplo 5: Obtener la suma de los elementos impares:

Este ejercicio podemos hacerlo de varias formas:

```
int res1 = numeros.stream()
    .filter(x -> x%2!=0)
    .reduce(0, (a, b) -> a + b);
System.out.println(res1);
```

O también:

```
int res2 = lista.stream()
    .map(z -> z.getMeGusta())
    .mapToInt(Integer::intValue)
    .sum();
System.out.println(res2);
```

Y también con esta última versión:

```
Optional res3 = lista.stream()
    .map(z -> z.getMeGusta())
    .reduce(Integer::sum);
System.out.println(res3.get());
```

Ejemplo 6: A partir de un ArrayList de String, hacer una lista con los que empiecen por "A":

```
List<String> textos = Arrays.asList("Alfa", "Bravo", "Charlie", "Aback");
textos.stream()
    .filter(s->s.startsWith("A"))
    .forEach(System.out::println);
```

Ejemplo 7: A partir de una lista de "Peli" mostrar los títulos de las películas posteriores al año 1998, ordenadas alfabéticamente.

```
lista.stream()
    .filter(x -> x.getAño() > 1998)
    .map(x -> x.getNombre())
    .sorted()
    .forEach(System.out::println);
```

Ejemplo 8: A partir de una lista de "Peli" mostrar el título de la película más antigua:


```
Optional<Peli> var = lista.stream()
    .min( (a,b)->a.getAño()-b.getAño());
System.out.println(var.get().getNombre());
```

4.1 Tratar Ficheros como Streams

La programación funcional se puede aplicar también al tratamiento de ficheros. Podemos convertir el contenido de un fichero en un Stream y luego aplicarle todas las operaciones que acabamos de ver.

El siguiente ejemplo, leería un fichero de texto línea a línea y mostraría por pantalla en mayúsculas aquellas líneas de menos de 10 caracteres:

```
String fichero = "fichero.txt";
try (Stream<String> stream = Files.lines(Paths.get(fichero))) {
    stream.filter(x -> x.length() < 10)
        .map(String::toUpperCase)
        .forEach(System.out::println);
} catch (IOException e) {e.printStackTrace();}
```


 El mundo de los Streams y la programación funcional es muy amplio, con muchas más posibilidades de las vistas en esta introducción.