Resumen — Expresiones Lambda y Java Streams

CFGS DAM · IES de Teis · Preparación de examen

1. Expresiones Lambda

Una expresión lambda es un bloque corto de código (función anónima) que toma parámetros y devuelve un valor, similar a un método pero sin nombre. Introducidas en Java 8, son clave para Streams y eventos GUI. Sintaxis: **(parámetros) -> { cuerpo }**. Con un único parámetro puedes omitir paréntesis; con un cuerpo de una línea, puedes omitir llaves y *return* cuando aplica.

Tipos más comunes (interfaces funcionales):

- **Consumidores**: reciben parámetros y no devuelven nada (p. ej., *Consumer<T>*).
- **Proveedores**: no reciben parámetros y devuelven resultado (p. ej., *Supplier<T>*).
- **Funciones**: reciben y devuelven (*Function<T,R>*).
- **Predicados**: devuelven booleano (*Predicate<T>*).
- **Operadores**: devuelven el mismo tipo que reciben (*UnaryOperator<T>*).

2. Java Streams

Un **Stream** encapsula una fuente de datos (colecciones, arrays o canales de E/S) y permite operar sobre sus elementos con lambdas de forma declarativa y encadenada. Stream API aparece en Java 8.

Cómo crear un stream:

```
// Desde Collection
List<String> lista = Arrays.asList("Uno", "Dos", "Tres");
Stream<String> s1 = lista.stream();

// Directamente con Stream.of(...)
Stream<String> s2 = Stream.of("Juan", "Francisco", "Ana", "María");

// Desde un array
String[] nombres = {"Juan", "Francisco", "Ana", "María"};
Stream<String> s3 = Stream.of(nombres);
```

Características clave:

- No es una estructura de datos; consume desde Collections, Arrays o I/O.
- No modifica los datos originales: produce resultados a partir de ellos.
- Las **operaciones intermedias** son perezosas (*lazy*) y devuelven otro Stream (encadenables).
- Una **operación terminal** cierra el Stream y produce el resultado final.

2.4.1. Operaciones intermedias

- **map**(f): transforma cada elemento con una función.
- **filter**(p): deja pasar elementos que cumplan el predicado.
- **sorted**([comparator]): ordena el stream.
- **flatMap**(f): aplana streams anidados (colecciones → elementos).
- **distinct**(): elimina duplicados (según equals).
- **peek**(c): inspección/depuración sin modificar el stream.

```
List<String> frutas = Arrays.asList("manzana","pera","uva","plátano","kiwi","pera","manzana"

// map
List<String> mayus = frutas.stream().map(String::toUpperCase).toList();

// filter
List<String> largas = frutas.stream().filter(f -> f.length() > 5).toList();

// sorted
List<String> ordenadas = frutas.stream().sorted().toList();

// flatMap (frutas -> letras)
List<String> letras = frutas.stream()
    .flatMap(f -> Arrays.stream(f.split("")))
    .toList();

// distinct
List<String> sinDup = frutas.stream().distinct().toList();

// peek (debug)
List<String> debug = frutas.stream()
    .filter(f -> f.contains("a"))
    .peek(f -> System.out.println("pasa filtro: " + f))
    .map(String::toUpperCase)
    .peek(f -> System.out.println("en mayúsculas: " + f))
    .toList();
```

2.4.2. Operaciones terminales

- **collect**(collector): reúne en colección u otra estructura.
- **forEach**(c): aplica una acción a cada elemento.
- **reduce**(op): acumula en un solo valor.
- **count**(): cuenta elementos.
- **findFirst**(): obtiene el primer elemento (Optional).
- **allMatch**(p) / **anyMatch**(p): comprobaciones booleanas.

```
List<String> frutas = Arrays.asList("manzana","pera","uva","plátano","kiwi","pera","manzana"

// collect
List<String> res = frutas.stream()
    .map(String::toUpperCase)
    .distinct()
    .collect(Collectors.toList());

// forEach
frutas.stream().forEach(System.out::println);

// reduce (concatenar)
String todas = frutas.stream().reduce((a,b) -> a + ", " + b).orElse("Lista vacía");

// count
long n = frutas.stream().count();

// findFirst
Optional<String> primera = frutas.stream().findFirst();

// allMatch / anyMatch
boolean todasConA = frutas.stream().allMatch(f -> f.contains("a"));
```

```
boolean algunaLarga = frutas.stream().anyMatch(f -> f.length() > 6);
```

3. Streams y ficheros (Java NIO)

Usa **Files.lines(Path)** para un Stream de líneas con try-with-resources:

```
public class LeerArchivo {
    public static void main(String[] args) {
        Path p = Path.of("res/palabras.txt");
        try (Stream<String> lineas = Files.lines(p)) {
            lineas.forEach(System.out::println);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Cheat■Sheet de examen (recordatorios rápidos)

- **Pipeline típico**: stream() → filter → map → sorted → collect.
- **map vs flatMap**: map 1→1; flatMap 1→muchos (y aplana).
- **lazy**: nada se ejecuta hasta la operación terminal.
- **inmutabilidad**: el origen no cambia.
- **Optional**: usa orElse / ifPresent con findFirst().
- **distinct()** usa equals: define equals/hashCode en tus clases.
- **peek()** es para depurar; evita efectos laterales.

Consejo: primero decide *qué* resultado quieres (colección, agregado o comprobación) y luego compón el pipeline con las intermedias adecuadas antes de la terminal correspondiente.

Checklist de estudio

1. Sintaxis de lambda y firmas de Consumer, Supplier, Function, Predicate, Operator.
2. Crear streams desde Collection, arrays y Stream.of(...).
3. Practicar intermedias: map, filter, sorted, flatMap, distinct, peek.
4. Practicar terminales: collect, forEach, reduce, count, findFirst, allMatch, anyMatch.
5. Ejercicio con Files.lines(Path) + try■with■resources.

¡Éxitos en el examen! ■