Anexo - Java Streams

CFGS DAM - IES de Teis

1. Expresiones Lambda

1.1. ¿Qué son?

Una **expresión lambda**, también llamada función lambda, función literal o función anónima, es una subrutina definida que no está enlazada a un identificador.

Son muy comunes en los lenguajes de programación funcional.

En **Java** se implementaron con la **versión 8**. Muy usadas en el manejo de *streams* y los eventos de las interfaces gráficas.

Una expresión lambda es un bloque corto de código que toma parámetros y devuelve un valor, similar a un método, pero sin necesidad de nombre.

Sintaxis: `(parámetros) -> {cuerpo-lambda}`

- Cuando se tiene un solo parámetro no es necesario utilizar los paréntesis.
- Cuando el cuerpo tiene una sola línea no hace falta usar las llaves ni poner return en case de devolver valores.

1.2. Tipos de expresiones Lambda

- Consumidores: reciben parámetros y no devuelven nada.
- Proveedores: no reciben parámetros y devuelven un resultado.
- Funciones: reciben parámetros y devuelven un resultado.
- Predicados: reciben parámetros y devuelven valor lógico.
- Operadores: reciben un parámetro y devuelven un valor del mismo tipo.

2. Java Streams

2.1. ¿Qué son?

Los streams son clases que encapsulan colecciones o arrays y permiten que éstos soporten operaciones utilizando lambdas. De esa forma se simplifica operar sobre los

elementos de toda la colección o sobre la propia colección en sí. Stream API se introduce en Java 8.

2.2. Cómo crear un stream

Los streams se pueden generar llamando al método stream() de cualquiera de las implementaciones de la interfaz java.util.Collection.

```
List<Animal> listaAnimales = new ArrrayList<>();
listaAnimales.add("Perro");
listaAnimales.add("Gato");
listaAnimales.add("Hamster");
Stream<Animal> streamAnimales = listaAnimales.stream();
List<String> lista = Arrays.asList("Uno", "Dos", "Tres"));
Stream<String> streamDeLista = lista.stream();
```

También se pueden crear a partir de objetos de forma individual, sin partir de ninguna colección ya existente:

```
Stream<String> streamNombres = Stream.of("Juan", "Francisco", "Ana",
"María");
```

Y también se puede crear un stream a partir de un array:

```
String[] nombres = {"Juan", "Francisco", "Ana", "María"};
Stream<String> streamNombres = Stream.of(nombres);
```

2.3. Características de los streams

The features of Java streams are mentioned below:

- A Stream is not a data structure; it just takes input from Collections, Arrays or I/O channels.
- Streams do not modify the original data; they only produce results using their methods.
- Intermediate operations (like filter, map, etc.) are lazy and return another Stream, so you can chain them together.

- A terminal operation (like collect, forEach, count) ends the stream and gives the final result.

Las características de los flujos de Java se mencionan a continuación:

- Un stream **no es una estructura de datos**. Simplemente recibe la entrada de colecciones, matrices o canales de E/S.
- Los flujos **no modifican los datos originales**. Solo producen resultados utilizando sus métodos.
- Las operaciones intermedias (como filter, map, etc.) son lazy (perezosas) y devuelven otro stream, por lo que se pueden encadenar.
- Las operaciones terminales (como collect, forEach, count) finalizan el stream y proporcionan el resultado final.

2.4. Tipos de operaciones en Streams

2.4.1. Operaciones intermedias

Las operaciones intermedias tienen las siguientes características:

- Los métodos están encadenados.
- Las operaciones intermedias transforman un stream en otro.
- Permiten el concepto de filtrado, donde un método filtra los datos y los pasa a otro método después de procesarlos.

Las operaciones intermedias más importantes son:

1. **map**(): devuelve un stream con los resultados de aplicar la función dada a los elementos de este flujo.
2. **filter**(): selecciona elementos según el predicado pasado como argumento.
3. **sorted**(): ordena el stream. Puede aceptar un Comparator para especificar cómo llevar

a cabo la ordenación.

4. **flatMap**(): aplana un flujo de colecciones en un único flujo de elementos.

5. **distinct**(): elimina elementos duplicados. Devuelve un stream con los elementos distintos (según Object.equals(Object)).

6. **peek**(): realiza una acción sobre cada elemento sin modificar el stream. Devuelve un stream que consta de los elementos de este stream y, además, realiza la acción proporcionada en cada elemento a medida que se consumen elementos del stream resultante. Útil para depurar.

Ejemplo que demuestra el uso de las operaciones intermedias previas:

```
import java.util.*;
import java.util.stream.Collectors;

public class StreamSobreUnMismoArray {
    public static void main(String[] args) {

        // Nuestro array base
        List<String> frutas = new ArrayList<>(
                Arrays.asList("manzana", "pera", "uva", "plátano", "kiwi",
"pera", "manzana", "cereza")
        );

        System.out.println("Lista original: " + frutas);

        // 1) map(): transformar las frutas en mayúsculas
        List<String> enMayusculas = frutas.stream()
                .map(String::toUpperCase)
                .toList();
        System.out.println("\n1) map() -> Mayúsculas: " + enMayusculas);

        // 2) filter(): quedarnos solo con las frutas de más de 5 letras
        List<String> largas = frutas.stream()
                .filter(f -> f.length() > 5)
                .toList();
        System.out.println("\n2) filter() -> Más de 5 letras: " + largas);

        // 3) sorted(): ordenar alfabéticamente
```

```
        List<String> ordenadas = frutas.stream()
                .sorted()
                .toList();
        System.out.println("\n3) sorted() -> Orden alfabético: " +
ordenadas);

        // 4) flatMap(): ejemplo con frases -> separar frutas en letras
        List<String> letras = frutas.stream()
                .flatMap(f -> Arrays.stream(f.split(""))) // cada fruta se
convierte en letras
                .toList();
        System.out.println("\n4) flatMap() -> Letras individuales: " +
letras);

        // 5) distinct(): eliminar duplicados
        List<String> sinDuplicados = frutas.stream()
                .distinct()
                .toList();
        System.out.println("\n5) distinct() -> Sin duplicados: " +
sinDuplicados);

        // 6) peek(): mostrar elementos que pasan por el stream (debug)
        List<String> debug = frutas.stream()
                .filter(f -> f.contains("a"))
                .peek(f -> System.out.println("peek -> pasa filtro: " + f))
                .map(String::toUpperCase)
                .peek(f -> System.out.println("peek -> en mayúsculas: " +
f))
                .toList();
        System.out.println("\n6) peek() -> Resultado final: " + debug);
    }
}
```

2.4.2. Operaciones terminales

Las operaciones terminales devuelven el resultado. Estas operaciones no se procesan más ya que solo devuelven un valor de resultado final.

1. **collect**(): El método "collect" devuelve el resultado de las operaciones intermedias realizadas en el stream.

2. **forEach**(): El método "forEach" itera cada elemento del stream.

3. **reduce**(): El método "reduce" reduce los elementos de un flujo a un único valor. El método "reduce" toma un operador binario como parámetro.

4. **count**(): Devuelve el número de elementos en el flujo.

5. **findFirst**(): Devuelve el primer elemento del flujo, si está presente.

6. **allMatch**(): Comprueba si todos los elementos del flujo coinciden con un predicado dado.

7. **anyMatch**(): Comprueba si algún elemento del flujo coincide con un predicado dado. Nota: Las operaciones intermedias se ejecutan según el concepto de evaluación diferida (lazy), que garantiza que cada método devuelva un valor fijo (operación terminal) antes de pasar al siguiente método.

Ejemplo utilizando operaciones terminales:

```
import java.util.*;
import java.util.stream.Collectors;

public class TerminalOperationsExample {
    public static void main(String[] args) {

        // Nuestro array base
        List<String> frutas = new ArrayList<>(
                Arrays.asList("manzana", "pera", "uva", "plátano", "kiwi",
"pera", "manzana", "cereza")
        );

        System.out.println("Lista original: " + frutas);

        // 1) collect(): recolectar el stream en una lista sin duplicados y
en mayúsculas
        List<String> frutasMayusSinDuplicados = frutas.stream()
```

```
                .map(String::toUpperCase)
                .distinct()
                .collect(Collectors.toList());
        System.out.println("\n1) collect() -> Mayúsculas y sin duplicados:
" + frutasMayusSinDuplicados);

        // 2) forEach(): imprimir cada elemento directamente del stream
        System.out.println("\n2) forEach() -> Imprimir frutas:");
        frutas.stream().forEach(System.out::println);

        // 3) reduce(): concatenar todas las frutas en una sola cadena
separada por comas
        String todasJuntas = frutas.stream()
                .reduce((a, b) -> a + ", " + b)
                .orElse("Lista vacía");
        System.out.println("\n3) reduce() -> Concatenadas: " +
todasJuntas);

        // 4) count(): contar cuántas frutas hay
        long cantidad = frutas.stream().count();
        System.out.println("\n4) count() -> Cantidad de frutas: " +
cantidad);

        // 5) findFirst(): obtener la primera fruta
        Optional<String> primera = frutas.stream().findFirst();
        System.out.println("\n5) findFirst() -> Primera fruta: " +
primera.orElse("No hay frutas"));

        // 6) allMatch(): ¿todas las frutas contienen la letra 'a'?
        boolean todasConA = frutas.stream().allMatch(f -> f.contains("a"));
        System.out.println("\n6) allMatch() -> ¿Todas contienen 'a'? " +
todasConA);

        // 7) anyMatch(): ¿alguna fruta tiene más de 6 letras?
        boolean algunaLarga = frutas.stream().anyMatch(f -> f.length() >
6);
        System.out.println("\n7) anyMatch() -> ¿Alguna tiene más de 6
letras? " + algunaLarga);
    }
}
```

3. Ejemplo del uso de streams en ficheros

La clase Files de Java NIO nos permite generar un Stream de un archivo de texto a través del método lines().

```
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();
        }
    }
}
```

Documentación

- https://www.geeksforgeeks.org/java/stream-in-java/
- https://java.codeandcoke.com/apuntes:streams
- Apuntes de Programación de Rafael del Castillo
- ChatGPT