Apunte 10 - Programación funcional y API de Streams

Introducción a la programación funcional

Al desarrollar software se identifican paradigmas de programación que gobiernan el enfoque con el que se encara el problema, se diseña una solución y finalmente se redacta el código fuente. Entre los paradigmas más utilizados se destacan el paradigma estructurado y el paradigma orientado a objetos.

Otro paradigma de uso habitual es el paradigma funcional. El mismo tiene una base matemática bastante compleja y por tal motivo existen pocos lenguajes denominados puros, es decir, que permiten únicamente desarrollar con este paradigma. Sin embargo, actualmente todos los lenguajes de uso cotidiano (tales como Java, C#, Python o Javascript) incluyen ciertos elementos de esta metodología.

A pesar de que estos lenguajes están mayormente concentrados en los paradigmas estructurado y orientado a objetos, ofrecen algunas características de programación funcional. Esto se logra mediante la posibilidad de trabajar con un nuevo tipo de dato denominado "función".

El origen de esta perspectiva puede parecer desconcertante, pero se la puede analizar desde el punto de vista de las variables. Cuando se utiliza la programación estructurada, los programas poseen variables donde se almacenan valores. En esta modalidad, cada variable tiene la capacidad de almacenar un único valor de alguno de los tipos de datos ofrecidos por el lenguaje, tales como números, cadenas, booleanos, etc. En cambio, en la programación orientada a objetos, las variables pueden contener objetos, los cuales pueden tener múltiples atributos y métodos. Aunque es válido argumentar que las variables solo contienen referencias a objetos, en esencia, sigue siendo cierto que cada variable almacena un objeto.

En la programación funcional, las variables almacenan código fuente, es decir, conjuntos de instrucciones del lenguaje. Por lo tanto, si una variable se declara como una función, es posible ejecutar el código que esta contiene. Y dado que son variables pueden cambiar su contenido, por lo tanto, si en algún momento se asigna un código diferente a la variable y se ejecuta su contenido nuevamente, se llevará a cabo una ejecución distinta. En conclusión, siempre se ejecutará el conjunto de instrucciones que esté almacenado en la variable en ese momento específico.

Funciones de orden superior

Los lenguajes que ofrecen programación funcional, incluso en su forma más básica, nos brindan la posibilidad de tener variables cuyo contenido sea una función. También podemos tener parámetros de tipo función, lo cual implica que un método puede ejecutar una función sin necesidad de saber con exactitud su comportamiento. A pesar de parecer bastante confuso, este desacoplamiento es muy utilizado gracias a las interfaces de Java.

De la misma manera, un método puede retornar una función para que sea almacenada desde la llamada al mismo.

Los métodos que poseen parámetros o retorno de tipo función se denominan métodos o funciones de orden superior. Esta noción tiene una gran relevancia matemática y es muy útil en diversas situaciones de programación.

Interfaces funcionales

La principal dificultad que se evidencia en un lenguaje orientado a objetos y fuertemente tipado como es Java radica en que su sintaxis no es flexible para un dato de tipo función. Al ser orientado a objetos, se debe trabajar con clases, objetos y métodos.

Java logra ofrecer variables de tipo función mediante el concepto de las interfaces funcionales. Una interfaz funcional es una interfaz que posee un único método, la cual puede estar opcionalmente anotada como `@FunctionalInterface`.

Esta anotación no impide su uso para programación funcional, pero previene del potencial error de que se le agregue un segundo método. En ese caso, al disponer de más de un método deja de considerarse una interfaz funcional y todas las clases que la implementen dejan de compilar correctamente. Si la interfaz está anotada como funcional, el agregado de un segundo método es detectado por el compilador como un error en la interfaz en lugar de marcar como erróneas las clases implementadoras.

Caso 1: Calculadora con orientación a objetos

Se presenta a continuación una implementación de una calculadora básica utilizando POO para crear una calculadora básica. Los requerimientos iniciales de la misma se resumen en que el usuario pueda indicar una operación y los dos operandos numéricos con los que se desea calcular. Se desea que se puedan agregar nuevas operaciones sin afectar el código existente.

Para ello se define una interfaz llamada "Operacion" con un método "calcular" el cual toma dos operandos y devuelve el resultado. Esta interfaz representará las operaciones básicas de la calculadora.

```
public interface Operacion {
    float calcular(float a, float b);
}
```

A continuación, por cada operación que la calculadora ofrece se agrega una clase que implemente la interfaz Operacion. Cada clase proporciona su propia implementación del método "calcular".

```
public class Suma implements Operacion {
    @Override
    public float calcular(float a, float b) {
        return a + b;
    }
}

public class Resta implements Operacion {
    @Override
    public float calcular(float a, float b) {
```

```
        return a - b;
    }
}

public class Multiplicacion implements Operacion {
    @Override
    public float calcular(float a, float b) {
        return a * b;
    }
}

public class Division implements Operacion {
    @Override
    public float calcular(float a, float b) {
        if (b != 0) {
            return a / b;
        } else {
            throw new ArithmeticException("No se puede dividir entre
cero.");
        }
    }
}
```

Finalmente en la clase principal se crea una instancia de cada una de las operaciones y se las almacena en alguna estructura de datos, tal como un arreglo.

```
import java.util.Scanner;

public class Calculadora {

    public static void main(String[] args) {

        Operacion[] operaciones;

        operaciones = new Operacion[]{
                new Suma(),
                new Resta(),
                new Multiplicacion(),
                new Division()
        };

        Scanner scanner = new Scanner(System.in);
        float a, b;
        int opcion;
        System.out.println("Calculadora básica");
        System.out.println("Seleccione una operación:");
        System.out.println("1. Suma");
        System.out.println("2. Resta");
        System.out.println("3. Multiplicación");
        System.out.println("4. División");
        System.out.println("0. Salir");
```

```
        opcion = scanner.nextInt();
        while (opcion != 0) {
            if (opcion >= 1 && opcion <= 4) {
                System.out.print("Ingrese el primer número: ");
                a = scanner.nextFloat();
                System.out.print("Ingrese el segundo número: ");
                b = scanner.nextFloat();
                float resultado = operaciones[opcion - 1].calcular(a, b);
                System.out.println("El resultado es: " + resultado);
            } else {
                System.out.println("Opción inválida. Intente
nuevamente.");
            }
            System.out.println();
            System.out.println("Seleccione una operación:");
            opcion = scanner.nextInt();
        }
    }

}
```

Esta implementación a pesar de cumplir con todos los requerimientos impone la dificultad de crear una clase por cada operación con el único objetivo de brindar una sobreescritura al método calcular.

Funciones lambda

Las extensiones funcionales de Java permiten implementar interfaces funcionales sin requerir la creación explícita de una nueva clase mediante las expresiones lambda.

Dado que una interfaz funcional posee un único método, resulta redundante programar una nueva clase con una sobreescritura. Si se declara una variable de referencia a dicha interfaz, es inevitable que debe referenciar a una nueva implementación que sobreescriba es único método. Las expresiones lambda ofrecen la posibilidad de escribir únicamente el bloque de código del método directamente en el lugar de la declaracion de la variable, el cual es considerado automáticamente como la sobreescritura del método.

Una expresión lambda tiene la siguiente sintaxis

```
    (parámetros) -> { cuerpo }
```

De esta manera, las operaciones de la calculadora pueden ser reescritas de una manera mucho más concisa y sin programar una clase nueva por cada una, siendo cada una de las expresiones lambda una sobreescritura del método calcular:

```
    Operacion suma = (float a, float b) -> { return a + b; };
    Operacion resta = (float a, float b) -> { return a - b; };
    Operacion multiplicacion = (float a, float b) -> { return a * b; };
    Operacion division = (float a, float b) -> {
        if (b != 0) {
```

```
            return a / b;
        } else {
            throw new ArithmeticException("No se puede dividir entre
cero.");
        }
    };
    operaciones = new Operacion[] { suma, resta, multiplicacion, division
};
```

Las expresiones lambda pueden omitir los paréntesis en el caso de poseer un único parámetro. Y de la misma manera si su cuerpo consiste en una única instrucción return pueden ser omitidas las llaves y la palabra clave return.

Entonces las funciones suma, resta y multiplicación pueden ser aún más simples:

```
    Operacion suma = (float a, float b) -> a + b;
    Operacion resta = (float a, float b) -> a - b;
    Operacion multiplicacion = (float a, float b) -> a * b;
```

Finalmente para utilizar las funciones se invoca a su método calcular():

```
    float resultado = suma.calcular(34,87);
```

Interfaces funcionales provistas

La API de funciones de java ofrece una amplia cantidad de interfaces funcionales para los casos más habituales. Se considera una práctica no recomendada la creación de interfaces funcionales nuevas si ya existe en la API una equivalente.

Las diversas interfaces funcionales que provee el paquete java.util.function se diferencian entre sí únicamente por la cantidad y tipo de parámetros y el tipo de retorno. Existe una cantidad bastante grande de interfaces funcionales provistas, para diferentes combinaciones de tipos de parámetros y de retorno. La más destacas son:

Nombre de la interfaz	Parámetros	Tipo de retorno	Uso
`Supplier<T>`	Ninguno	`T`	Generador de valores, se suele usar con Stream.generate() (más adelante)
`Consumer<T>`	`T`	`void`	Recibe un valor pero no retorna nada, se suele utilizar para el método forEach()
`Predicate<T>`	`T`	`boolean`	Dado un valor retorna un boolean, se suele utilizar con filter()

Nombre de la interfaz	Parámetros	Tipo de retorno	Uso
`Function<T, R>`	`T`	`R`	Dado un objeto retorna uno diferente, probablemente de otro tipo, se usa con map()
`UnaryOperator<T>`	`T`	`T`	También para usar con map, cuando el tipo del retorno es el mismo que el parámetro

En el momento de necesitar una variable de tipo función, únicamente se debe analizar los tipos de los parámetros y del retorno y con ellos seleccionar una de las interfaces funcionales de la tabla.

Para el caso de la calculadora, el método calcular recibe dos parámetros float y retorna otro float. Por lo tanto las interfaces más adecuadas son `BinaryOperator<Float>` o `DoubleBinaryOperator`

De esta manera, para el caso de la calculadora tampoco es requerida la declaración de la interfaz Operacion, siendo adecuado crear las variables como:

```
    DoubleBinaryOperator suma = (double a, double b) -> a + b;
    DoubleBinaryOperator resta = (double a, double b) -> a - b;
    DoubleBinaryOperator multiplicacion = (double a, double b) -> a * b;
    DoubleBinaryOperator division = (double a, double b) -> {
        if (b != 0) {
            return a / b;
        } else {
            throw new ArithmeticException("No se puede dividir entre
cero.");
        }
    };
```

Y dado que estas variables son utilizadas únicamente para llenar el arreglo de operaciones, también se las puede omitir y agregar el bloque de cada una al llenado del arreglo:

```
    DoubleBinaryOperator []operaciones = {
        (double a, double b) -> a + b,
        (double a, double b) -> a - b,
        (double a, double b) -> a * b,
        (double a, double b) -> {
            if (b != 0) {
                return a / b;
            } else {
                throw new ArithmeticException("No se puede dividir entre
cero.");
            }
        }
    };
```

Ejemplos

- Calculadora con clases
- Calculadora con interfaz funcional propia
- Calculadora con interfaz funcional provista

Stream API

La Stream API o API de flujos permite manipular y procesar conjuntos de datos como una secuencia o flujo. A diferencia de las colecciones que mantienen todos los datos almacenados en memoria y que pueden ser recorridos mediante ciclos, los flujos plantean un concepto en cierto modo opuesto. Un stream es una secuencia de datos que van "transcurriendo", sin estar necesariamente almacenados todos juntos. Y sobre esa secuencia el programador introduce operaciones que manipulen cada dato individualmente, eliminandolo de la responsabilidad de recorrerlos.

Puede imaginarse a un flujo como una cascada, en la cual cada gota de agua es cada uno de los datos. Al caer el agua desde la cascada puede cada gota individualmente ser interceptada y desviada por las piedras o ramas existentes, pero finalmente todas las gotas finalizan cayendo en una laguna. Siguiendo esta analogía, las operaciones que se insertan en el flujo son las interrupciones que las gotas encuentran en su caída.

Obtención de flujos

La API permite crear flujos desde diversos orígenes:

- Desde colecciones: la interfaz Collection ofrece un método stream() que instancia un nuevo flujo con los datos almacenados en la coleccion. Es el mecanismo más utilizado para generar nuevos flujos.
- Desde arreglos: el método Arrays.stream() recibe como parámetro un arreglo y genera un flujo con los datos del mismo.
- Desde valores fijos: el método Stream.of() recibe una serie de objetos separados por comas y genera un nuevo flujo con esos valores.
- Flujos infinitos con iterate(): el método Stream.iterate() recibe dos parámetros: un valor inicial y un UnaryOperator del mismo tipo que el valor inicial. El método genera un flujo infinito en el que el primer valor es el valor inicial indicado en el parámetro y los siguiente se obtienen con el resultado de aplicar el unary operator al valor anterior. Dado que el flujo es infinito, debe ser interrupido con la operación intermedia limit()
- Flujos infinitos con generate(): el método Stream.generate() recibe un Supplier y genera un flujo cuyos valores son los resultados de llamar repetidamente al Supplier. También debe ser interrumpido con limit().
- Métodos generadores: algunos objetos permite generar flujos particulares según sus responsabilidades. Por ejemplo: el método Files.lines() genera un flujo de String con cada línea de un archivo de texto, o el método getResultStream() de JPA que retorna un flujo de entidades sin que ocupen nunca memoria del proceso, como sí lo hace getResultList().

Operaciones intermedias

Las operaciones intermedias procesan cada uno de los elementos del flujo con algún criterio u objetivo y retornan otro flujo con el resultado de la operación. De esta manera se pueden encadenar más de una operación intermedia, las cuales se ejecutan en orden para cada uno de los elementos del flujo.

Habitualmente las operaciones intermedia reciben como parámetros instancias de interfaces funcionales, las cuales pueden ser redactadas como funciones lambda.

A continuación se presentan las operaciones intermedias más habituales:

filter

La operación intermedia filter recibe como parámetro un predicado, es decir una función lambda que recibe un parámetro y retorna un boolean. Filter invoca a la función lambda una vez por cada elemento del flujo pasándoselo como parámetro y retorna un nuevo flujo que contiene cada uno de los elementos para los que la función retornó verdadero.

```
List<Integer> numeros = new ArrayList<>();
// ...
// llenar la lista con números
///...

Stream<Integer> pares = numeros.stream().filter(x -> x % 2 == 0);

// pares es un flujo que contendrá
// los mismos números de la lista original
// pero sólo si son pares.
```

distinct

La operación intermedia distinct() retorna un nuevo flujo con los mismos datos del flujo original pero eliminando los repetidos.

map

La operación intermedia map() recibe como parámetro una instancia de la interfaz funcional Function de forma tal que genere un nuevo valor (incluso posiblemente de otro tipo) a partir de cada uno de los elementos del flujo.

```
List<Integer> numeros = new ArrayList<>();
// ...
// llenar la lista con números
///...

Stream<Integer> cuadrados = numeros.stream().map(x -> x * x);
Stream<Double> raices = numeros.stream().map(x -> Math.sqrt(x))
// cuadrados es un flujo que contendrá
// los cuadrados de todos los números de la lista original
// mientras que raices contiene la raíz cuadrada de cada uno
```

limit

Recibe un número entero y genera un nuevo flujo que contiene como máximo la cantidad indicada. Es especialmente útil para los flujos infinitos:

```
Random r = new Random();
Stream<Integer> aleatorios = Stream.generate(() ->
r.nextInt()).limit(100);
/// aleatorios contendrá 100 números enteros generados al azar.
```

`sorted`

Retorna un nuevo flujo con los mismos elementos que el original pero ordenados. Si el flujo es de objetos que implementan Comparable los ordena según el resultado del método compareTo. Tambien puede recibir un objeto Comparator y los ordena con el resultado del método compare.

Operaciones terminales

Las operaciones terminales recolectan todos los elementos de un flujo y los procesan pero sin retornar un nuevo flujo. Algunas operaciones realizan una operaciones y no retornan nada mientras que otras obtienen un resultado. Es importante destacar que toda manipulación de un flujo debe finalizar siempre con una operación terminal. Las operaciones intermedias no recorren el conjunto de datos ni ejecutan nada hasta que no se ejecuta una operación terminal.

Las operaciones terminales más usadas son las siguientes:

`count`

Retorna la cantidad de elementos del flujo.

`forEach`

Ejecuta un Consumer por cada elemento. El consumer recibe como parámetro el dato y no retorna nada

```
pares.forEach(x -> System.out.println(x));
// Imprime cada uno de los números del flujo pares.
```

min / max

Con el mismo criterio que la operación sorted, obtienen respectivamente el menor y mayor valor del flujo

anyMatch / allMatch / noneMatch

Estas operaciones intermedias reciben un predicado y retornan un boolean. La operación anyMatch retorna verdadero si algún elemento del flujo cumple con el predicado, allMatch si todos cumplen y noneMatch si para ningún elemento el predicado se hace verdadero.

```
if (aleatorios.anyMatch(x -> x > 100)) {
    System.out.println("Hay algún número mayor a 100");
    if (aleatorios.allMatch(x -> x > 100))
        System.out.println("Y son todos mayores a 100");
}
```

Collectors (mínimo imprescindible y útiles de verdad)

En la API de Streams de Java, un **Collector** es un objeto que define cómo transformar los elementos de un flujo en un resultado final.
Su función principal es **acumular**, **combinar** y **finalizar** los datos procesados en el stream, permitiendo producir colecciones, resúmenes, agrupamientos o incluso valores únicos a partir de los elementos.

Se utiliza mediante el método terminal `collect(...)`, que recibe un Collector como parámetro.
La clase de utilidad `Collectors` provee implementaciones listas para los casos más comunes (listas, conjuntos, mapas, estadísticas, uniones de cadenas, agrupamientos, etc.).

En términos sencillos:

```
[ Elementos del Stream ] → [ Collector ] → [ Resultado final ]
```

- **Entrada:** elementos del stream.
- **Collector:** define cómo juntarlos.
- **Salida:** resultado final (lista, mapa, string, estadística, etc.).

`toList`, `toSet`, `toCollection`

```
List<String> nombres = List.of("Ana", "Bruno", "Ana");
List<String> lista = nombres.stream().collect(Collectors.toList()); //
mantiene orden y duplicados
Set<String> conjunto = nombres.stream().collect(Collectors.toSet()); //
elimina duplicados, sin orden garantizado
Deque<String> deque =
nombres.stream().collect(Collectors.toCollection(ArrayDeque::new));
```

Cuándo usar:

- `toList()` cuando querés una lista inmutable *de facto* (desde Java 16 devuelve una lista no modificable).
- `toCollection(...)` cuando necesitás una estructura específica.

`joining`

```

```

```
String csv = nombres.stream().collect(Collectors.joining(", "));
// "Ana, Bruno, Ana"
```

Tips: admite separador, prefijo y sufijo: `joining(", ", "[", "]")` ⇒ `[Ana, Bruno, Ana]`.

`groupingBy` (agrupaciones frecuentes)

```
record Persona(String nombre, String ciudad, int edad) {}

List<Persona> personas = List.of(
    new Persona("Ana", "Córdoba", 22),
    new Persona("Luis", "Córdoba", 31),
    new Persona("Mia",  "Rosario", 19),
    new Persona("Leo",  "Rosario", 27),
    new Persona("Zoe",  "Mendoza", 24)
);

Map<String, List<Persona>> porCiudad = personas.stream()
    .collect(Collectors.groupingBy(Persona::ciudad));
```

Downstream collectors (contar, mapear, sumar):

```
Map<String, Long> cantidadPorCiudad = personas.stream()
    .collect(Collectors.groupingBy(Persona::ciudad,
Collectors.counting()));

Map<String, List<String>> nombresPorCiudad = personas.stream()
    .collect(Collectors.groupingBy(Persona::ciudad,
             Collectors.mapping(Persona::nombre, Collectors.toList())));

Map<String, Double> promedioEdadPorCiudad = personas.stream()
    .collect(Collectors.groupingBy(Persona::ciudad,
             Collectors.averagingInt(Persona::edad)));
```

`partitioningBy` (predicado booleano)

```
Map<Boolean, List<Persona>> menoresYMayores = personas.stream()
    .collect(Collectors.partitioningBy(p -> p.edad() < 21));
```

`toMap` (con merge y mapa ordenado)

```
// Clave: nombre, Valor: edad (si hay duplicado, me quedo con la edad
mayor)
Map<String, Integer> edadPorNombre = personas.stream()
    .collect(Collectors.toMap(
        Persona::nombre,
        Persona::edad,
        Integer::max // merge
    ));

// Mantener orden de inserción
Map<String, Integer> edadPorNombreOrdenado = personas.stream()
    .collect(Collectors.toMap(
        Persona::nombre,
        Persona::edad,
        (a,b) -> a,
        LinkedHashMap::new
    ));
```

`summarizingInt` (estadísticas rápidas)

```
IntSummaryStatistics stats = personas.stream()
    .collect(Collectors.summarizingInt(Persona::edad));
System.out.println(stats.getCount());
System.out.println(stats.getMin());
System.out.println(stats.getAverage());
System.out.println(stats.getMax());
```

Resumen de un stream `reduce`

El método `reduce` recibe un BinaryOperator y efectúa una operación de reducción o pliegue con todos los elementos del flujo. La reducción repite una operación entre todos los elementos obteniendo un único resultado. Para ello toma los dos primeros elementos del flujo y le aplica la operación (al ser un BinaryOperator la operacion recibe dos datos y retorna un resultado del mismo tipo), luego toma el tercero y repite la operación entre este y el resultado anterior. Luego procede de la misma manera con todos los elementos existentes hasta que el flujo finalice

```
int sumaCuadrados = cuadrados.reduce((a, b) -> a + b);
// Reduce suma los dos primeros números, que se reciben en los parámetros
a y b
// Luego obtiene el tercero y vuelve a ejecutarse con a igual a la suma
anterior
// y b con el valor del tercero. Desde ahí repite.
```

Variantes de `reduce`

Si entramos en detalle ahí algunas particularidades que si bien son menores está bueno conocer.

1. **Sin identidad (la versión mostrada arriba):**
    - Retorna un `Optional<T>` porque el stream podría estar vacío.
    - Útil cuando no se tiene un valor inicial claro.

```
Optional<Integer> suma = Stream.of(1, 2, 3, 4)
                               .reduce((a, b) -> a + b);
```

2. **Con identidad:**
    - Se especifica un valor inicial seguro (identidad matemática).
    - Evita trabajar con `Optional`.
    - Ejemplo: suma con identidad `0`, producto con identidad `1`.

```
int suma = Stream.of(1, 2, 3, 4)
                 .reduce(0, (a, b) -> a + b);
```

3. **Con identidad, acumulador y combinador:**
    - Forma completa que admite ejecución **paralela**.
    - El **acumulador** procesa elementos parciales.
    - El **combinador** une resultados intermedios.

```
int sumaParalela = IntStream.rangeClosed(1, 1000)
                            .parallel()
                            .reduce(0, Integer::sum, Integer::sum);
```

Referencias a métodos

Las referencias a métodos, también conocidas como "method references" en inglés, son una característica de programación funcional que está presente en lenguajes como Java, que soportan este paradigma.

En Java, las referencias a métodos permiten pasar una referencia a un método como argumento a una función de orden superior o a un método en lugar de tener que definir una expresión lambda para ese método. Es una forma más concisa y legible de expresar funciones o comportamientos que ya están definidos en algún lugar del código.

Existen cuatro tipos de referencias a métodos en Java:

- Referencias a métodos estáticos: Permiten referenciar a métodos estáticos de una clase.

```
// Sintaxis: Clase::métodoEstático
MiClase::metodoEstatico
```

- Referencias a métodos de instancia de un objeto particular: Permiten referenciar a métodos de instancia de un objeto específico.

```
// Sintaxis: objeto::métodoDeInstancia
miObjeto::metodoDeInstancia
```

- Referencias a métodos de instancia de un tipo arbitrario: Permiten referenciar a métodos de instancia de cualquier objeto de un tipo determinado.

```
// Sintaxis: Tipo::métodoDeInstancia
MiClase::metodoDeInstancia
```

- Referencias a constructores: Permiten referenciar a constructores de clases.

```
// Sintaxis: Tipo::new
MiClase::new
```

Estas referencias se pueden utilizar en contextos donde se espera una expresión lambda, siempre que el método al que se hace referencia tenga la misma firma que la función esperada. Es importante destacar que las referencias a métodos no ejecutan el método en el momento de definición, sino que proporcionan una referencia al mismo para que se pueda invocar más adelante.

Streams Primitivos (evitar boxing cuando suma)

¿Qué son los Streams primitivos?

Además de los `Stream<T>` genéricos, Java provee implementaciones especializadas para trabajar con **valores primitivos**:

- `IntStream`
- `LongStream`
- `DoubleStream`

Estos streams están optimizados para manejar números sin necesidad de *boxing* y *unboxing* (evitan convertir entre `int` ↔ `Integer`, `double` ↔ `Double`, etc.), lo que mejora el rendimiento en operaciones numéricas intensivas.

Se pueden crear de varias formas:

- `IntStream.range(1, 10)` → secuencia de enteros del 1 al 9.
- `IntStream.rangeClosed(1, 10)` → secuencia de enteros del 1 al 10.

- `new Random().ints(5)` → 5 enteros aleatorios.

Incluyen métodos terminales propios, como:

- `sum()`, `average()`, `min()`, `max()` → cálculos directos sin Collectors.
- `summaryStatistics()` → retorna un objeto con cantidad, mínimo, máximo, suma y promedio.

También ofrecen conversiones:

- `.boxed()` → convierte un `IntStream` en `Stream<Integer>`.
- `.mapToInt(...)`, `.mapToDouble(...)`, `.mapToLong(...)` → permiten cambiar de un stream de objetos a uno primitivo.

En términos sencillos:

- Usar **Streams primitivos** cuando el flujo es numérico y largo.
- Usar **Streams genéricos** cuando se trabaja con objetos o la expresividad es más importante que la micro-optimización.

> **Idea clave:** `IntStream`, `LongStream`, `DoubleStream` reducen overhead de boxing/unboxing en pipelines numéricos.

```
[ Datos primitivos ] → [ IntStream / LongStream / DoubleStream ] → [
Operaciones numéricas rápidas ]
```

Crear y convertir

```
IntStream unoACien = IntStream.rangeClosed(1, 100); // 1..100
DoubleStream aleatorios = new Random().doubles(5);  // 5 doubles en [0,1)

// De objetos a primitivo y viceversa
List<Integer> lista = List.of(1,2,3,4,5);
IntStream prim = lista.stream().mapToInt(Integer::intValue);
List<Integer> otra = prim.boxed().toList();
```

Agregaciones típicas

```
int suma = IntStream.range(0, 1_000_000).sum();
OptionalDouble promedio = IntStream.of(10, 20, 30).average();
IntSummaryStatistics statEdad =
personas.stream().mapToInt(Persona::edad).summaryStatistics();
```

Ejemplo integrador (edades)

```
record Alumno(String nombre, int edad) {}

List<Alumno> alumnos = List.of(
    new Alumno("Ana", 20), new Alumno("Bruno", 22), new Alumno("Carla",
19)
);

double prom = alumnos.stream().mapToInt(Alumno::edad).average().orElse(0);
int max  = alumnos.stream().mapToInt(Alumno::edad).max().orElse(-1);
int min  = alumnos.stream().mapToInt(Alumno::edad).min().orElse(-1);
```

Regla práctica: si nuestro pipeline es numérico y largo, *consideraremos* streams primitivos; si el valor viaja como objeto (p.ej. `Persona`), priorizamos así claridad y convertimos a primitivo **solo** donde aporta.

Buenas Prácticas, Anti-patrones y IO

Lazy evaluation y *single-use*

- Las operaciones intermedias **no** ejecutan nada hasta una terminal.
- Un `Stream` **no se puede reutilizar**: cada pipeline termina en una terminal.

```
Stream<String> s = Stream.of("a","b","c");
s.forEach(System.out::println);
// s.count(); // ❌ IllegalStateException: stream has already been
operated upon or closed
```

Evitar *side-effects*

- Preferí `map/filter` puras.
- Si necesitás efectos, sé explícito y local:

```
List<String> salida = new ArrayList<>();
// ❌ Anti-patrón: mutar una colección externa dentro del pipeline
// datos.stream().forEach(salida::add);

// ✅ Mejor: recolectá
List<String> salida2 = datos.stream().collect(Collectors.toList());
```

`peek` solo para depurar

```
List<Integer> r = IntStream.rangeClosed(1,5)
    .peek(i -> System.out.println("in:"+i))
    .map(i -> i*i)
    .peek(i -> System.out.println("out:"+i))
    .boxed().toList();
```

> **Regla:** no uses `peek` para lógica de negocio.

`reduce` con identidad y combinador

```
int total = IntStream.rangeClosed(1, 5)
    .reduce(0, (acum, x) -> acum + x);          // identidad = 0

int producto = IntStream.rangeClosed(1, 5)
    .reduce(1, (a, b) -> a * b);

// Para pipelines potencialmente paralelos: identidad + acumulador +
combinador
int sumaPar = IntStream.rangeClosed(1, 1_000)
    .parallel()
    .reduce(0, Integer::sum, Integer::sum);
```

Manejo correcto de archivos (`Files.lines`)

- `Files.lines(path)` **mantiene abierto** el recurso: usá *try-with-resources*.
- Indicá charset si aplica.

```
Path path = Path.of("personas.csv");
try (Stream<String> lineas = Files.lines(path, StandardCharsets.UTF_8)) {
    List<String> nombresOrdenados = lineas
        .map(linea -> linea.split(";")[1]) // suponiendo columna 1 =
nombre
        .filter(n -> !n.isBlank())
        .sorted()
        .toList();
}
```

Cuándo **no** usar Streams

- Cuando un simple `for` mejora **claridad** y **rendimiento**.
- Cuando necesitás control de flujo complejo (break/continue anidados, múltiples índices).
- Cuando estás mutando intensamente estructuras compartidas.

Mini-checklist

- [ ] ¿La versión con Streams es **más legible** que con `for`?
- [ ] ¿Evitaste *side-effects* dentro del pipeline?
- [ ] ¿Cerrás correctamente recursos (`try-with-resources`)?
- [ ] ¿Necesitás realmente recolectar a lista, o alcanza con una terminal (`forEach`, `count`)?

Apéndice rápido: `flatMap` en 2 ejemplos (opcional pero útil)

```
List<String> frases = List.of("hola mundo", "java streams");
List<String> tokens = frases.stream()
    .flatMap(f -> Arrays.stream(f.split(" ")))
    .toList();

record Curso(String nombre, List<String> alumnos) {}
List<Curso> cursos = List.of(
    new Curso("A", List.of("Ana","Bruno")),
    new Curso("B", List.of("Mia","Leo"))
);
List<String> todos = cursos.stream()
    .flatMap(c -> c.alumnos().stream())
    .toList();
```

Ejemplo: Listado por pantalla

Para mostrar por pantalla todo el contenido de un flujo se puede utilizar la operación terminal forEach, que recibe un Consumer. Por lo tanto puede pasarse cualquier expresión lambda que reciba un parámetro y no retorne nada. El método println del objeto System.out cumple con esa firma, entonces el uso sería:

```
// Generación de una lista de 100 números aleatorios enteros entre 1 y 100
List<Integer> numeros = new Random().ints(100,1,1000).boxed().toList();
// Listado por pantalla
numeros.stream().forEach(x -> System.out.println(x));
```

En este caso, dado que el único parámetro formal de la expresión lambda es el único parametro actual del método println, se puede enviar una referencia a dicho método, sin necesidad de hacer llamadas explícitas al mismo:

```
numeros.stream().forEach(System.out::println);
```

Ejemplo: Lectura de un archivo

Si se dispone de un archivo de texto que contiene un dato individual por cada línea se puede aprovechar la api de flujos para leerlo y obtener objetos a partir de su contenido de una forma ágil, en muy pocas líneas de código y sin usar ninguna estructura de control.

La clase Files provee el método estático readAllLines recibe el nombre de un archivo y devuelve una lista de String con el contenido de todas las líneas del archivo. Si se dispone de algún método que reciba un String y retorne una instancia de alguna clase cuyo estado pueda ser llenado interpretando el String, se puede utilizar la operación intermedia map para convertir el flujo de cadenas en un flujo de objetos:

```
List<Integer> numeros;
Path archivo = Paths.get("numeros.txt");
```

```
// El método Files.lines(...) devuelve un Stream<String> que mantiene
abierto el archivo
// hasta que se termina de recorrer el flujo. Para evitar dejar el recurso
abierto,
// lo envolvemos en un try-with-resources: así el Stream (y el archivo
asociado)
// se cierran automáticamente al salir del bloque try.
try (Stream<String> lineas = Files.lines(archivo)) {
    // Convertimos cada línea (String) a Integer
    numeros = lineas.map(Integer::valueOf)
                    .toList();
}
// Aquí el Stream 'lineas' ya está cerrado automáticamente
```

De la misma manera, si una clase posee un constructor que recibe un objeto (de la misma clase u otra) y con sus atributos pueda crear una nueva instancia, se puede pasar una referencia a dicho constructor. Un caso habitual se da al leer un archivo de valores separados por comas. En el siguiente ejemplo se dispone de un archivo que en cada línea posee los datos de una persona, separados por caracteres de punto y coma.

Si se le agrega a la clase Persona un constructor que reciba un String y lo interprete para asignar los valores de sus atributos, todo el proceso de la lectura del archivo se reduce a una línea:

```
// En la clase Persona:
public Persona(String linea) {
    String[]valores = linea.split(";");
    this.documento = Integer.valueOf(valores[0]);
    this.nombre = valores[1];
    this.apellido = valores[2];
    this.edad = Integer.valueOf(valores[3]);
}

// En la clase que lee el archivo
List<Persona> plantel = Files.lines(Paths.get("personas.csv"))
                             .map(Persona::new)
                             .toList();
```

Ejemplos de aplicación

- Procesamiento del archivo de numeros
- Procesamiento del archivo de personas

Apunte 11 – Procesamiento de Archivos CSV

Introducción

En este apunte vamos a explorar cómo trabajar con archivos CSV en Java. Nuestro objetivo no es seguir un paso a paso de implementación, sino comprender los conceptos, herramientas y APIs involucradas. Queremos que, cuando después sigamos un tutorial práctico, tengamos claras las bases conceptuales que justifican cada decisión.

Los archivos CSV son un formato de texto simple, donde los datos se organizan en filas y columnas, separados por un delimitador (generalmente `,` o `;`). Esta simplicidad los hace muy usados para intercambio de datos entre sistemas, aunque también implica ciertos desafíos como el manejo de comillas, espacios en blanco y caracteres especiales.

Si bien este tema es más bien una aplicación de lo que hemos visto hasta aquí que un apunte conceptual, lo agregamos en este punto dada la importancia que tiene en el contenido de la asignatura. En el parcial individual tendremos que llevar a cabo el proceso de un archivo CSV para cargar en una base de datos las instancias asociadas a este archivo y por lo tanto será necesario revisar este mecanismo y tenerlo claro.

Lectura básica con `Scanner` y `split`

Empezamos con la forma más elemental de procesar un archivo CSV: leerlo línea por línea y dividir cada línea en columnas usando un delimitador.

Herramientas:

- `java.util.Scanner` para iterar sobre las líneas del archivo.
- `String.split(";")` para dividir las cadenas.

Ambas clases (`Scanner` y `String`) forman parte del JDK y nos permiten construir una primera solución sin dependencias externas.

```
Path csv = Path.of("personas.csv");
try (Scanner sc = new Scanner(Files.newBufferedReader(csv))) {
    while (sc.hasNextLine()) {
        String[] columnas = sc.nextLine().split(";");
        // columnas[0] → documento
        // columnas[1] → nombre
        // columnas[2] → apellido
        // columnas[3] → edad
    }
}
```

> Aquí recorremos línea por línea, separamos por `;` o `,` y obtenemos un array de `String` que luego podemos mapear a objetos.

Como vimos en el apunte anterior una forma elemental de realizar este mapeo es agregar a la clase del objeto a construir un constructor que reciba un array de strings y realice la asignación de los datos a los atributos, aunque también podemos optar por mecanismos más sofisticados como usar alguna librería de mapeo.

Lectura robusta con `BufferedReader`

Cuando necesitamos mayor control (charset, excepciones, cierres automáticos), `BufferedReader` es una alternativa más explícita.

Herramientas:

- `java.nio.file.Files.newBufferedReader`
- `java.io.BufferedReader`

```
Path archivo = Path.of("personas.csv");
try (BufferedReader br = Files.newBufferedReader(archivo,
StandardCharsets.UTF_8)) {
    String linea;
    while ((linea = br.readLine()) != null) {
        String[] columnas = linea.split(";");
        // procesar columnas...
    }
}
```

> El `try-with-resources` asegura que el archivo se cierre. Controlamos directamente el charset (`UTF-8`) y evitamos problemas con acentos o caracteres especiales.

Uso de OpenCSV con `CSVReader`

Llegado un punto, manejar manualmente los delimitadores, comillas y escapes se vuelve tedioso. Aquí entra en juego una librería externa: OpenCSV.

Herramientas:

- Dependencia externa: `com.opencsv.CSVReader`.
- Capacidad de manejar delimitadores, comillas y escapes automáticamente.

Maven:

En el archivo `pom.xml` deberemos agregar la dependencia a la librería OpenCSV

```
<dependency>
  <groupId>com.opencsv</groupId>
  <artifactId>opencsv</artifactId>
  <version>5.9</version>
</dependency>
```

Y luego utilizando la librería podemos trabajar de la siguiente manera:

```
try (CSVReader reader = new
CSVReader(Files.newBufferedReader(Path.of("personas.csv")))) {
     String\[] fila;
     while ((fila = reader.readNext()) != null) {
         // fila\[0] → documento
         // fila\[1] → nombre
         // fila\[2] → apellido
         // fila\[3] → edad
     }

}
```

> Usamos `CSVReader` para leer filas completas ya separadas y limpias, evitando lidiar con casos problemáticos manualmente.

OpenCSV con `CSVReaderHeaderAware`

A veces preferimos trabajar por nombres de columnas en lugar de índices, para hacer el código más legible y menos dependiente del orden. Este proceso depende que el archivo CSV tenga una primera línea que en general llamamos encabezado con los nombres de cada una de las columnas.

Herramientas:

- `CSVReaderHeaderAware` de OpenCSV.

```
try (CSVReaderHeaderAware reader = new
CSVReaderHeaderAware(Files.newBufferedReader(Path.of("personas.csv")))) {
    Map<String, String> fila;
    while ((fila = reader.readMap()) != null) {
        String nombre = fila.get("nombre");
        String apellido = fila.get("apellido");
    }
}
```

Cada fila se transforma en un `Map<String,String>` donde la clave es el nombre de la columna y el valor el dato correspondiente.

`CsvToBean` + POJOs con Lombok

Contexto: Para un diseño más orientado a objetos, podemos mapear cada fila a una clase Java. Esto nos permite trabajar con objetos fuertemente tipados.

Herramientas:

- `CsvToBean` de OpenCSV.

- Anotaciones de OpenCSV (`@CsvBindByName`).
- Lombok para reducir boilerplate (`@Data`, `@NoArgsConstructor`).

```
@Data
@NoArgsConstructor
public class Persona {
    @CsvBindByName(column = "documento") private Integer documento;
    @CsvBindByName(column = "nombre")    private String nombre;
    @CsvBindByName(column = "apellido")  private String apellido;
    @CsvBindByName(column = "edad")      private Integer edad;
}

try (Reader r = Files.newBufferedReader(Path.of("personas.csv"))) {
    List<Persona> personas = new CsvToBeanBuilder<Persona>(r)
        .withType(Persona.class)
        .withSeparator(';')
        .withIgnoreLeadingWhiteSpace(true)
        .build()
        .parse();
}
```

OpenCSV crea objetos `Persona` automáticamente a partir de los datos del archivo, vinculando columnas con atributos gracias a las anotaciones.

Anexo – Procesamiento con Streams y Collectors

Una vez que tenemos los objetos `Persona` cargados desde un CSV, podemos aprovechar la API de Streams de Java para realizar consultas y transformaciones de forma declarativa.

Ejemplos:

1. **Promedio de edad de las personas**

```
double promedioEdad = personas.stream()
    .mapToInt(Persona::getEdad)
    .average()
    .orElse(0);
```

2. **Conteo de personas por apellido**

```
Map<String, Long> conteoPorApellido = personas.stream()
    .collect(Collectors.groupingBy(Persona::getApellido,
Collectors.counting()));
```

3. **Top 3 personas más grandes por edad**

```
List<Persona> top3 = personas.stream()
    .sorted(Comparator.comparing(Persona::getEdad).reversed())
    .limit(3)
    .toList();
```

4. **Generación de un string con todos los nombres**

```
String nombres = personas.stream()
    .map(Persona::getNombre)
    .collect(Collectors.joining(", "));
```

> Con esto vemos cómo el procesamiento de CSV se integra naturalmente con la programación funcional: leemos los datos, los convertimos en objetos y luego usamos Streams y Collectors para obtener información útil de manera simple y expresiva.

Anexo – Procesamiento declarativo con Streams & Collectors sobre CSV

Contexto. Una vez que mapeamos el CSV a `List<Persona>`, queremos explotar la API de Streams para consultas y transformaciones **declarativas** (sin escribir bucles manuales). Nuestro objetivo aquí es entender **qué** resuelve cada operación y **por qué** la elegimos, más que hacer un paso a paso.

Herramientas. Usamos `java.util.stream.*` (Streams), `java.util.Comparator` (ordenamientos), y `java.util.stream.Collectors` (recolecciones a listas, mapas, estadísticas, etc.).

Todo lo que mostramos a continuación forma parte del JDK (no dependemos de librerías externas), y se apoya en el diseño visto en el apunte anterior de Programación Funcional y Streams.

Escenario base

Partimos de una lista ya construida con OpenCSV:

```
// Suponemos la clase Persona del apunte
@Data
@NoArgsConstructor
public class Persona {
    @CsvBindByName(column = "documento") private Integer documento;
    @CsvBindByName(column = "nombre")    private String  nombre;
    @CsvBindByName(column = "apellido")  private String  apellido;
    @CsvBindByName(column = "edad")      private Integer edad;
}

List<Persona> personas;
try (Reader r = Files.newBufferedReader(Path.of("personas.csv"))) {
    personas = new CsvToBeanBuilder<Persona>(r)
        .withType(Persona.class)
        .withSeparator(';')
        .withIgnoreLeadingWhiteSpace(true)
        .build()
```

```
            .parse();
    }
```

Estadísticas y métricas rápidas

Promedio de edades (usamos stream primitivo para evitar boxing):

```
double promedio = personas.stream()
    .mapToInt(Persona::getEdad)
    .average()
    .orElse(0);
```

Resumen completo (`count`, `min`, `max`, `sum`, `avg`) con `summaryStatistics`:

```
IntSummaryStatistics stats = personas.stream()
    .mapToInt(Persona::getEdad)
    .summaryStatistics();
// stats.getCount(), stats.getMin(), stats.getAverage(), stats.getMax(),
stats.getSum()
```

Selecciones y ordenamientos

Top 3 por edad (descendente):

```
List<Persona> top3 = personas.stream()
    .sorted(Comparator.comparing(Persona::getEdad).reversed())
    .limit(3)
    .toList();
```

Filtrar y proyectar solo nombres de mayores de 21:

```
List<String> nombresMayores = personas.stream()
    .filter(p -> p.getEdad() != null && p.getEdad() > 21)
    .map(Persona::getNombre)
    .toList();
```

Transformaciones a Map y manejo de duplicados

Documento → Persona (con estrategia de merge):

```
Map<Integer, Persona> porDocumento = personas.stream()
    .filter(p -> p.getDocumento() != null)
```

```
    .collect(Collectors.toMap(
        Persona::getDocumento,
        p -> p,
        // Si hay documentos duplicados, nos quedamos con la persona de
mayor edad
        (p1, p2) -> (p1.getEdad() >= p2.getEdad() ? p1 : p2),
        LinkedHashMap::new // preserva orden de aparición
    ));
```

Apellido → lista de nombres (downstream `mapping`):

```
Map<String, List<String>> nombresPorApellido = personas.stream()
    .collect(Collectors.groupingBy(
        Persona::getApellido,
        Collectors.mapping(Persona::getNombre, Collectors.toList())
    ));
```

Conteos y agregaciones por clave

Cantidad por apellido:

```
Map<String, Long> conteoPorApellido = personas.stream()
    .collect(Collectors.groupingBy(Persona::getApellido,
Collectors.counting()));
```

Promedio de edad por apellido:

```
Map<String, Double> promedioEdadPorApellido = personas.stream()
    .collect(Collectors.groupingBy(
        Persona::getApellido,
        Collectors.averagingInt(Persona::getEdad)
    ));
```

Join de nombres (string final):

```
String listado = personas.stream()
    .map(Persona::getNombre)
    .collect(Collectors.joining(", ", "[", "]"));
```

Buenas prácticas aplicadas al CSV

- **`try-with-resources`** para `Reader`/`BufferedReader`/`Lines`: evitamos dejar archivos abiertos.

- **Validaciones y trim**: antes de mapear/convertir, conviene `trim()` y controlar nulos para evitar `NumberFormatException`. -* **Evitar side-effects** dentro del pipeline: preferimos recolectar (`collect`) a estructuras nuevas en vez de mutar listas externas.
- **Elegir la estructura correcta**: `List`, `Set`, `LinkedHashMap` según orden/duplicados que necesitemos.
- **No sobreusar Streams**: si un `for` simple es más claro, lo elegimos. Streams suman cuando mejoran **legibilidad** y **expresividad**.

Reflexión final

En este recorrido vimos cómo podemos ir desde la solución más simple (leer líneas y dividir con `split`) hasta un enfoque más robusto y orientado a objetos usando OpenCSV y Lombok.

No pretendemos que cada alumno memorice todas las variantes, sino que comprendamos qué herramientas tenemos disponibles y cuándo conviene aplicar cada una. Más adelante, en un paso a paso práctico, podremos ejercitar estos conceptos en ejemplos concretos y ver cómo aprovechar Streams y Collectors para procesar la información leída.