1.- Streams

El concepto de **Stream** surge a partir de Java 8 y permitió comenzar a trabajar con colecciones de datos de una manera más funcional y declarativa. Proporciona una manera moderna de procesar colecciones sin la necesidad de escribir bucles explícitos ni crear variables intermedias.
Recordad que una colección (Lista, Set, Mapa ..) almacena datos, mientras que los stream procesan los datos de las colecciones de forma eficiente.

Aquí te explico en detalle para qué sirven, por qué son útiles y algunos de sus métodos más comunes.

Un **Stream** en Java representa una secuencia de elementos sobre la cual se pueden realizar diversas operaciones, como filtrado, mapeo, modificaciones, etc. Estas operaciones pueden ser intermedias o finales.

- Las intermedias permiten encadenar varias transformaciones una a continuación de otra, pero no generan ningún resultado final ni modifican el origen de datos.
- Las finales son las que desencadenas el procesamiento del stream de forma que, cuando se ejecutan, todas las operaciones intermedias encadenadas hasta ese punto se ejecutan y el Stream ya no puede reutilizarse.

Conceptos clave:

•Inmutabilidad: Los Streams no modifican las fuentes originales (por ejemplo, Listas, Sets, etc.).
•Lazy Evaluation (Evaluación perezosa): Las operaciones intermedias de los Streams no se ejecutan hasta que se encuentra una operación terminal.
•Unidireccional: Los Streams solo pueden ser recorridos una vez.

¿Por qué son útiles?

- **Concisión**: Reduce la cantidad de código necesario para realizar operaciones complejas.
- **Legibilidad**: El código se vuelve más fácil de leer y entender.
- **Flexibilidad**: Puedes encadenar múltiples operaciones sin necesidad de crear variables intermedias.

Ejemplo de un Stream:

```
List<String> nombres = Arrays.asList("Ana", "Pedro", "Juan", "Maria");

nombres.stream()
    .filter(n -> n.length() > 3)    // Operación intermedia: filtrar nombres con más de 3 letras
    .map(String::toUpperCase)       // Operación intermedia: convertir a mayúsculas
    .forEach(System.out::println); // Operación final: imprimir cada elemento
```

2.- Creación de Streams

Desde colecciones (listas, set,...): stream()

```
List<String> lista = Arrays.asList("Java", "Python", "C++");
Stream<String> stream = lista.stream();
```

Desde arrays:

Se puede crear un Stream a partir de un array usando el método **Arrays.stream()** o directamente con el método **Stream.of**:

```
String[] lenguajes = {"Java", "Python", "C++"};
Stream<String> streamDesdeArray = Arrays.stream(lenguajes);

// Otra forma:
Stream<String> otroStream = Stream.of(lenguajes);
```

Desde valores literales:

Puedes crear un Stream a partir de un número variable de elementos usando **Stream.of():**

```
Stream<String> stream = Stream.of("Java", "Python", "C++");
```

Stream generado dinámicamente:

Puedes crear un Stream infinito de valores generados dinámicamente usando **Stream.generate()**. Debes usar operaciones como **limit**() para evitar que sea infinito.

```
Stream<String> repetidos = Stream.generate(() -> "Java");
repetidos.limit(3).forEach(System.out::println); // Java Java Java

Random random = new Random();
Stream<Integer> streamEnteros = Stream.generate(() -> random.nextInt(10)).limit(5);
streamEnteros.forEach(System.out::println);
```

Stream Iterativo:

Permite generar una secuencia infinita a partir de un valor inicial y una función de actualización:

```
Stream<Integer> numeros = Stream.iterate(0, n -> n + 2).limit(5);
numeros.forEach(System.out::println);  // Imprime los primeros 5 números pares
```

3.- Operaciones en Streams

3.1 Operaciones intermedias

Son las operaciones que permiten transformar o filtrar el Stream, pero no producen resultados por sí solas. Se ejecutan de forma **perezosa**. Estas operaciones devuelven un nuevo Stream y se pueden encadenar.

- **filter()**: Filtra elementos según una condición.

  Ejemplo1:

```
List<String> palabras = Arrays.asList("Java", "python", "JavaScript");
palabras.stream().filter(s -> s.startsWith("J")).forEach(System.out::println);
// Imprime "Java", "JavaScript"
```

  Ejemplo2:

```
String[] nombres = { "Juan", "Pedro", "Maria", "Ana" };
Stream<String> nuevoStream = Stream.of(nombres).filter(name-> name.contains("o"));
```

- **map()**: Transforma los elementos del Stream a otro tipo o valor.
  Ejemplo1:

```
Integer[] numeros = {1, 2, 3, 4, 5};
Stream<Integer> nuevo = Arrays.stream(numeros).map(n -> n * 2);
```

  //nuevo obtiene: {2,4,6,8,10}

  Ejemplo2:

```
Stream<String> valores = Stream.of("java", "python", "c++");
valores.map(palabra -> palabra.toUpperCase())
                      .forEach(palabra -> System.out.println(palabra));  //JAVA PYTHON C++

//Equivalente:
valores.map(String::toUpperCase).forEach(System.out::println);
```

- **distinct():** Elimina elementos duplicados.

  Ejemplo:

```
List<Integer> listaNumeros = Arrays.asList(1, 2, 2, 3, 4, 4);
listaNumeros.stream().distinct().forEach(System.out::println); // Imprime 1, 2, 3, 4
```

- **sorted():** Ordena los elementos. Se utiliza para ordenar los elementos de un Stream. De forma predeterminada, ordena los elementos en su orden natural (por ejemplo, para números, de menor a mayor). También puede aceptar un comparador personalizado para ordenaciones específicas.

  Ejemplo1: Ordenar los números en sentido ascendente:

```
List<Integer> numeros = Arrays.asList(5, 2, 9, 1, 7, 3);
Stream<Integer> nuevo= numeros.stream().sorted();
```

  Ejemplo2: Ordenar los números en sentido descendente:

```
List<Integer> numeros = Arrays.asList(5, 2, 9, 1, 7, 3);
Stream<Integer> nuevo= numeros.stream().sorted((a, b) -> b - a);
```

  Ejemplo3:

```
List<String> names = Arrays.asList("Carlos", "Ana", "Beatriz");
names.stream().sorted().forEach(System.out::println); // Imprime Ana, Beatriz, Carlos
```

  Ejemplo4:

```
List<Cliente> clientes = clienteService.getClientes();

clientes.stream().sorted((c1,c2) -> c1.getNombre().compareTo(c2.getNombre())).toList();
```

- **limit()**: Limitar el número de elementos

```
List<Integer> num = Arrays.asList(1, 2, 3, 4, 5, 6);
List<Integer> primerosTres = num.stream().limit(3).toList();

System.out.println(primerosTres);
```

- **skip():** Salta los primeros n elementos de un stream y sólo procesa los que siguen

```
List<Integer> saltarTres = num.stream()
    .skip(2).limit(3).toList();
System.out.println(saltarTres);  // [3,4,5]
```

3.2 Operaciones finales

Son aquellas que generan un resultado y finalizan el flujo de datos.

- **ForEach:** Aplica una acción a cada elemento del stream.

```
List<String> frutas = Arrays.asList("Fresa", "Melocotón", "Uvas");
frutas.stream().forEach(System.out::println);
frutas.stream().forEach(f -> System.out.println(f));
```

- **toList():** Obtiene una lista (a partir de Java 16)

```
List<String> frutas = Arrays.asList("Fresa", "Melocotón", "Uvas");
frutas.stream().toList();
```

- **Collect**: Transforma el stream en una colección (lista, conjunto,mapa, …)

```
Set<String> conjuntoNombres = nombres.stream()
          .collect(Collectors.toSet());  // Convierte el Stream en un Set

System.out.println(conjuntoNombres);

List<String> words = Arrays.asList("Java", "Streams", "Collectors");

Map<String, Integer> longitudPalabras = words.stream()
    .collect(Collectors.toMap(
        palabra -> palabra,            // Clave: la propia palabra
        palabra -> palabra.length()                // Valor: la longitud de la palabra
));

System.out.println(longitudPalabras);  // Salida: {Java=4, Streams=7, Collectors=10}
```

- **Reduce**: Realiza una operación de reducción sobre los elementos del stream, combinando los elementos en un solo valor. Esta operación es útil para sumar valores, encontrar el máximo o mínimo, o realizar combinaciones más complejas.

  Ejemplo: Sumar los valores de una lista:

```
List<Integer> n = Arrays.asList(1, 2, 3, 4);
int suma = n.stream().reduce(0, (a, b) -> a + b);

//equivalente:
suma = n.stream().reduce(0, Integer::sum); // Suma los números: 10
```

- **Count:** Devuelve el número de elementos en el stream.
  Long conteo = Stream.of("Java", "Python", "C++").count(); // Retorna 3

```
Long conteo = Stream.of("Java", "Python", "C++").count(); // Retorna 3
```

- **findFirst() y findAny()**: Encuentra el primer o cualquier elemento sin importar el orden. Ambos devuelven *Optional* por si no encuentra ningún elemento que cumpla con las condiciones.

```
// Usamos findFirst() para encontrar el primer elemento que empiece con 'C'
Optional<String> primerNombre = nombres.stream()
    .filter(nombre -> nombre.startsWith("C"))
    .findFirst();

// Verificamos si el Optional contiene un valor
if (primerNombre.isPresent()) {
    System.out.println("El primer nombre que empieza con C es: " + primerNombre.get());
} else {
    System.out.println("No se encontró ningún nombre que empiece con C.");
}
```

```
Optional<String> cualquierNombre = nombres.stream()
        .filter(nombre -> nombre.startsWith("A"))
        .findAny();

    // Verificamos si el Optional contiene un valor
    if (cualquierNombre.isPresent()) {
        System.out.println("Se encontró un nombre que empieza con A: " + cualquierNombre.get());
    } else {
        System.out.println("No se encontró ningún nombre que empiece con A.");
    }
```

  - **isPresent**() verifica si hay un valor en el Optional
  - **get**() obtiene ese valor si está presente.

- **AllMatch(), anyMatch(), noneMatch()**: Verifican condiciones sobre todos, alguno o ninguno de los elementos en un stream, respectivamente. Devuelve un boolean

```
// Verifica si todos los números son pares
boolean todosSonPares = num.stream().allMatch(n1 -> n1 % 2 == 0);

if (todosSonPares) {
    System.out.println("Todos los números son pares.");
} else {
    System.out.println("No todos los números son pares.");
}
```

- **Max() y Min():** Devuelven el valor máximo o mínimo del stream según un comparador dado, respectivamente.

```
// Encontrar la persona de mayor edad usando max() con expresión lambda
Optional<Cliente> personaMayor = clientes.stream()
    .max((p1, p2) -> p1.getEdad().compareTo(p2.getEdad()));

// Imprimir el resultado
personaMayor.ifPresent(valor -> System.out.println("La persona mayor es: " + valor));
```

System.out.println

Hay varias formas de usar System.out.println con Streams en Java:

1. Usar **forEach con una expresión lambda** para imprimir cada elemento del Stream:

```
List<String> nombres2 = Arrays.asList("Ana", "Pedro", "Maria");
nombres2.stream().forEach(x->System.out.println(x));
```

2. Usar forEach con una referencia a método: **System.out::println**

```
List<String> nombres2 = Arrays.asList("Ana", "Pedro", "Maria");
nombres2.stream().forEach(System.out::println);
```

3. Usando funciones intermedias:

```
// Filtrar nombres que comienzan con "A" y luego imprimir
lista.stream()
        .filter(nombre -> nombre.startsWith("A"))
        .forEach(System.out::println);
```

```
// Convertir nombres a mayúsculas y luego imprimir
nombres.stream()
        .map(String::toUpperCase)
        .forEach(System.out::println);
```

```
nombres.stream()
        .filter(nombre -> nombre.startsWith("A"))
        .collect(Collectors.toSet()).forEach( nombre-> System.out.println("Nombre filtrado: " + nombre));
```