Practicando con la API Stream

Java 8 introdujo una característica revolucionaria: **la API de Streams**. Esta permite procesar colecciones de datos de forma declarativa y eficiente, lo que hace que tu código sea más limpio, legible y, a menudo, más rápido.

¿Qué es la API Stream de Java 8?

Antes de resolver los problemas prácticos de la API Stream de Java 8, comprendamos qué es la API Stream.

La API Stream de Java admite muchos grupos de operaciones. A diferencia de las colecciones, los streams no almacenan datos; en cambio, procesan datos de una fuente (como una colección, un array o un canal de E/S) en una secuencia de operaciones.

Los conceptos clave de la API Stream giran en torno a:

- **Origen**: *Donde se origina el flujo de datos* (por ejemplo, una lista, un conjunto, un mapa (a través de sus vistas) o un array).
- **Operaciones intermedias**: *Estas operaciones transforman o filtran el flujo de datos*. Se ejecutan de forma diferida, es decir, **no se ejecutan hasta que se invoca una operación terminal**. Algunos ejemplos son `filter()`, `map()`, `sorted()` y `distinct()`.
- **Operaciones terminales**: *Son las operaciones finales de un flujo de datos* que proporcionan un resultado o realizan una acción, como imprimir o guardar datos. Consumen el flujo y marcan su final. Algunos ejemplos son `forEach()`, `collect()`, `reduce()`, `count()`, `anyMatch()` y `allMatch()`.

Se presenta aquí algunos ejemplos sencillos que implican el uso de varios métodos de la API Stream. Esta vez, no se te proporcionarán tareas específicas. En su lugar, te daré una lista con la que trabajar y el resultado esperado. En otras palabras, tu objetivo es realizar operaciones en la lista para obtener el resultado esperado.

Este tipo de tarea te ayudará a desarrollar la habilidad de obtener resultados sin una especificación clara. Solo verás los datos de entrada y el resultado esperado.

Nota:

Se recomienda trabajar este código en un IDE, ya que allí podrás ver los distintos métodos e IntelliJ te ayudará con la generación de código y las referencias a métodos. Sin embargo, nadie le impide completar esta tarea aquí en la forma que usted desee.

Ejercicio 01

```
import java.util.Arrays;
import java.util.List;

public class Ejercicio201 {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
        // Escriba su código aquí
    }
}
```

Utilice la API Stream para elevar al cuadrado cada número de la lista y recopile el resultado en una nueva lista.

Salida esperada:

```
[1, 4, 9, 16, 25]
```

Ayuda

Utilice aquí los métodos `map()` y `toList()`.

Solución

```
import java.util.Arrays;
import java.util.List;

public class Ejercicio201 {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
        List<Integer> result = numbers.stream().map(e -> e * e).toList();
        System.out.println(result);
        // Escriba su código aquí
    }
}
```

Ejercicio 02

```
import java.util.Arrays;
import java.util.List;

public class Ejercicio202 {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David", "Eva");
        // Escriba su código aquí
    }
}
```

Utilice la API de Stream para obtener la longitud del nombre más largo de la lista.

Resultado esperado:

```
7
```

Ayuda:

Utilice los métodos intermedios `map()` y `max()`, y el método final `get()`.

Solución

```
import java.util.Arrays;
import java.util.List;

public class Ejercicio202 {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David", "Eva");
        Integer result = names
                .stream()
                .map(e -> e.length())
                .max((integer, newInteger) -> Integer.compare(integer, newInteger))
                .get();
        System.out.println(result);
        // Escriba su código aquí
    }
}
```

Ejercicio 03

```
import java.util.Arrays;
import java.util.List;

public class Ejercicio203 {
    public static void main(String[] args) {
        List<String> sentences = Arrays.asList(
                "Java Stream API provides a fluent interface for processing sequences
                    of elements.",
                "It supports functional-style operations on streams of elements,
                    such as map-reduce transformations.",
                "In this exercise, you need to count the total number of words in
                    all sentences."
        );
        // Escriba su código aquí
    }
}
```

Utilice la API Stream para contar el número total de palabras distintas (sin distinción entre mayúsculas y minúsculas) en todas las oraciones.

Resultado esperado:

```
37
```

Ayuda

Utilice los métodos intermedios `flatMap()` y `distinct()`, y el método final `count()`.

Solución

```
import java.util.Arrays;
import java.util.List;

public class Ejercicio203 {
    public static void main(String[] args) {
        List<String> sentences = Arrays.asList(
                "Java Stream API provides a fluent interface for processing sequences
                    of elements.",
                "It supports functional-style operations on streams of elements, such
                    as map-reduce transformations.",
                "In this exercise, you need to count the total number of words in all
                    sentences."
        );
        long count = sentences.stream()
                .flatMap(e -> Arrays.stream(e.split(" ")))
                .distinct()
                .count();
        System.out.println(count);
        // Escriba su código aquí
    }
}
```

Ejercicio 04

```
import java.util.Arrays;
import java.util.List;

public class Ejercicio204 {
    public static void main(String[] args) {
        List<String> words = Arrays.asList("apple", "banana", "cherry", "date", "elderberry");
        // Escriba su código aquí
    }
}
```

Utilice la API de Stream para encontrar la concatenación de las dos primeras palabras que tengan longitudes pares.

Use collect(Collectors.joining()) para concatenar las palabras.

Resultado esperado:

```
bananacherry
```

Ayuda:

Utilice los métodos intermedios filter() y limit(), y el método final collect().

Solución

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class Ejercicio204 {
    public static void main(String[] args) {
        List<String> words = Arrays.asList("apple", "banana", "cherry", "date", "elderberry");
        String result = words.stream()
                .filter(e -> e.length() % 2 == 0)
                .limit(2)
                .collect(Collectors.joining());
        System.out.println(result);
        // Escriba su código aquí
    }
}
```

Ejercicio 05

```
import java.util.Arrays;
import java.util.List;

public class Ejercicio205 {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
        // Escriba su código aquí
    }
}
```

Utilice la API Stream para hallar la suma de los cuadrados de los números pares de la lista.

Utilice `mapToInt` en lugar de map para acceder al método terminal de suma.

Resultado esperado:

```
220
```

Ayuda:

Utilice los métodos intermedios `mapToInt()` y `filter()`, y el método final `sum()`.

Solución:

```
import java.util.Arrays;
import java.util.List;

public class Ejercicio205 {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
        int sum = numbers.stream()
                .mapToInt(e -> e * e)
```

```
                .filter(e -> e % 2 == 0)
                .sum();
        System.out.println(sum);
        // Escriba su código aquí
    }
}
```

Ejercicio 06: Filtrar una lista de enteros

Supongamos que tenemos una lista de enteros y necesitamos filtrar solo los números pares.

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class Ejercicio206 {

    public static void main(String[] args) {

        List<Integer> integerList = Arrays.asList(11, 12, 13, 14, 15, 16, 17, 18, 19, 20);

        List<Integer> evenNumbers = integerList.stream()
            .filter(num -> num % 2 == 0)
            .collect(Collectors.toList());

        System.out.println(evenNumbers); // Output: [12, 14, 16, 18, 20]
    }
}
```

Explicación:

Al principio tenemos una lista de números: Esta lista es la entrada para la operación de flujo.

- `nums.stream()`: Crea un flujo a partir de la lista, lo que permite realizar operaciones de estilo funcional.
- `.filter(nums ->nums% 2 == 0)`: Una operación intermedia que conserva solo los números pares aplicando la expresión lambda dada.
- `.collect(Collectors.toList())`: Una operación final que recopila los elementos del flujo filtrado en una nueva lista.

Ejercicio 07: Convertir una lista de cadenas a mayúsculas

Supongamos que tenemos una lista de cadenas y necesitamos convertir cada cadena de esa lista a mayúsculas.

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class StreamMapExample {

    public static void main(String[] args) {

        List<String> words = Arrays.asList("apple", "banana", "cherry");

        List<String> upperCaseWords = words.stream()
            .map(String::toUpperCase)
```

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

        System.out.println(upperCaseWords); // Output: [APPLE, BANANA, CHERRY]
    }
}
```

Explicación:

- `words.stream()`: Crea un flujo de cadenas a partir de la lista, lo que permite realizar operaciones con flujos.
- `.map(String::toUpperCase)`: Una operación intermedia que transforma cada cadena del flujo a mayúsculas mediante una referencia a un método.
- `.collect(Collectors.toList())`: Una operación final que recopila las cadenas transformadas en una nueva lista.

Ejercicio 08: Cómo encontrar el primer elemento de una lista

Tenemos una lista de cadenas y queremos encontrar el primer elemento de esa lista.

```
import java.util.Arrays;
import java.util.List;
import java.util.Optional;

public class Ejercicio208 {

    public static void main(String[] args) {

        List<String> colors = Arrays.asList("rojo", "verde", "azul");

        Optional<String> firstColor = colors.stream()
                                            .findFirst();

        firstColor.ifPresent(color ->
            System.out.println("El primer color es: " + color)
        ); // Output: The first color is: red
    }
}
```

Explicación:

- `.findFirst()`: Una operación terminal que devuelve el primer elemento del flujo, envuelto en un `Optional`. Maneja los casos en que el flujo podría estar vacío.
- `ifPresent()`: Un método de la clase Optional que ejecuta la expresión lambda proporcionada solo si hay un valor presente.

Ejercicio 09: Cálculo de la suma de los números en una lista

Queremos calcular la suma de todos los números en una lista.

```
import java.util.Arrays;
import java.util.List;

public class ReduceExample {

    public static void main(String[] args) {
```

```
        List<Integer> values = Arrays.asList(10, 20, 30, 40, 50);

        int numberSum = values.stream()
                              .reduce(0, Integer::sum);

        System.out.println("This is sum of all numbers in list : " + numberSum);
        // Output: This is sum of all numbers in list : 150
    }
}
```

Explicación:

- `.reduce(0, Integer::sum)`: Operación terminal que combina todos los elementos del flujo en un único resultado.
- `0`: Valor inicial (también conocido como valor *neutro*) para la operación de reducción.
- `Integer::sum`: Referencia al método `sum()` de la clase `Integer`, que realiza la suma de elementos.

Ejercicio 10: Conteo de elementos según una condición

Necesitamos contar el número de cadenas en la lista cuya longitud sea mayor que 2.

```
import java.util.Arrays;
import java.util.List;

public class Ejercicio210 {

    public static void main(String[] args) {

        List<String> fruits = Arrays.asList("apple", "fig", "banana", "kiwi", "orange", "grape",
                                            "avocado", "coconut", "raspberry", "apricot");

        long count = fruits.stream()
                           .filter(fruit -> fruit.length() > 4)
                           .count();

        System.out.println("Recuento de frutos cuya longitud de nombre es mayor a 4 : " + count);
        // Output: Count of fruits which length is greater than 4 : 3
    }
}
```

Explicación:

- Filtrado: Primero filtramos el flujo para conservar solo las frutas con una longitud mayor a 4.
- `.count()`: Esta es la operación final que devuelve cuántos elementos quedan en el flujo después del filtrado.

Ejercicio 11: Ordenar una lista de objetos

Necesitamos ordenar una lista de objetos de tipo Estudiante según su número de matrícula.

```
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
import java.util.stream.Collectors;

class Student {
    String name;
    int rollNumber;

    public Student(String name, int rollNumber) {
        this.name = name;
        this.rollNumber = rollNumber;
    }

    public String getName() {
        return name;
    }

    public int getRollNumber() {
        return rollNumber;
    }

    @Override
    public String toString() {
        return "Student{name='" + name + "', rollNumber=" + rollNumber + '}';
    }
}

public class Ejercicio211 {
    public static void main(String[] args) {
        List<Student> studentList = new ArrayList<>();
        studentList.add(new Student("Raul", 205));
        studentList.add(new Student("Sandra", 202));
        studentList.add(new Student("Amalia", 208));
        studentList.add(new Student("Patricia", 200));

        List<Student> sortedByRollNumber = studentList.stream()
                .sorted(Comparator.comparingInt(Student::getRollNumber))
                .collect(Collectors.toList());

        System.out.println(sortedByRollNumber);
    }
}
```

Explicación:

- `.sorted(Comparator.comparingInt(Student::getRollNumber))`: Esta es una operación intermedia que ordena el flujo de objetos Student.
- `Comparator.comparingInt(Student::getRollNumber)` crea un Comparator que compara los objetos `Student` según su número de matrícula (`rollNumber`), obtenido mediante el método `getRollNumber()`. Se utiliza el método `comparisonInt` porque `rollNumber` es un número entero.
- `.collect(Collectors.toList())`: Esta operación final recopila los objetos `Student` ordenados del flujo y devuelve una nueva lista que los contiene.

Ejercicio 12: Encontrar el valor máximo en una lista

Necesitamos encontrar el valor máximo presente en una lista de números enteros.

```
import java.util.Arrays;
import java.util.List;
import java.util.Optional;
```

```
public class Ejercicio212 {
    public static void main(String[] args) {
        List<Integer> scores = Arrays.asList(85, 92, 78, 95, 88, 97, 80);

        Optional<Integer> maxScore = scores.stream()
                                           .max(Integer::compare);

        maxScore.ifPresent(score -> System.out.println("La puntuación máxima es: " + score));

        // Example with an empty list
        List<Integer> emptyScores = Arrays.asList();
        Optional<Integer> maxEmptyScore = emptyScores.stream()
                                                     .max(Integer::compare);
        System.out.println("Puntuación máxima en una lista vacía: " + maxEmptyScore);
        // Output: Maximum score in an empty list: Optional.empty
    }
}
```

Explicación:

- `.max(Integer::compare)`: Esta es una operación terminal que encuentra el elemento máximo en el flujo utilizando el orden natural de los enteros (definido por Integer::compare).
- `Devuelve un Optional<Integer>` para manejar el caso en que el flujo esté vacío y, por lo tanto, no haya un valor máximo.
- `.ifPresent(score ->System.out.println(...))`: Esta parte del código comprueba si hay un valor presente en el Optional. Si lo hay, se ejecuta la expresión lambda proporcionada (que imprime la puntuación máxima).

Ejercicio 13: Agrupar elementos por una propiedad

Crear una agrupación de una lista de objetos Person por su edad.

```
import java.util.Arrays;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

class Person {
    String name;
    int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    @Override
    public String toString() {
        return "Person{name='" + name + "', age=" + age + '}';
    }
}

public class Ejercicio213 {
    public static void main(String[] args) {
```

```
        List<Person> people = Arrays.asList(
                new Person("Alicia", 30),
                new Person("Bertha", 25),
                new Person("Carlos", 35),
                new Person("Dario", 30)
        );

        Map<Integer, List<Person>> peopleByAge = people.stream()
                .collect(Collectors.groupingBy(Person::getAge));

        System.out.println(peopleByAge);
    }
}
```

Explicación:

- `.collect(Collectors.groupingBy(Person::getAge))`: Esta es la operación terminal que agrupa los objetos `Person` en el flujo según su edad.
- `Collectors.groupingBy(Person::getAge)` crea un `Collector` que organiza los elementos en un `Map`.

En el `Map` resultante, cada edad única se convierte en una clave, y el valor asociado a esa clave es una lista de todos los objetos Person que tienen esa edad específica.

Ejercicio 14: Comprobar si algún elemento cumple una condición

Ahora necesitamos comprobar si hay alguna cadena en la lista que comience con la letra “b”.

```
import java.util.Arrays;
import java.util.List;

public class Ejercicio214 {
    public static void main(String[] args) {
        List<String> animals = Arrays.asList("gato", "perro", "canario", "elefante");

        boolean startsWithB = animals.stream()
                                    .anyMatch(animal -> animal.startsWith("b"));

        System.out.println("¿Algún animal empieza con la letra 'b'? " + startsWithB);

        List<String> fruits = Arrays.asList("manzana", "uva", "sandía");
        boolean startsWithSInFruits = fruits.stream()
                                            .anyMatch(fruit -> fruit.startsWith("s"));
        System.out.println("Alguna fruta empieza con 's'? " + startsWithSInFruits);
    }
}
```

Explicación:

- `.anyMatch(animal ->animal.startsWith(“b”))`: Esta es una operación terminal que comprueba si al menos un elemento del flujo coincide con el predicado dado.

La expresión `lambda animal ->animal.startsWith(“b”)` es el **predicado**. Toma cada cadena de animal del flujo y comprueba si comienza con la letra “**b**”.

La operación `anyMatch()` devuelve **true** en cuanto encuentra un elemento que satisface el predicado. Si recorre todo el flujo sin encontrar una coincidencia, devuelve **false**.

Ejercicio 15: Aplanar una lista de listas

Dada una lista de listas de enteros, aplanarla en una sola lista.

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class Ejercicio215 {
    public static void main(String[] args) {
        List<List<Integer>> listOfLists = Arrays.asList(
                Arrays.asList(10, 20, 30),
                Arrays.asList(40, 50),
                Arrays.asList(60, 70, 80, 90)
        );

        List<Integer> flattenedNumbersList = listOfLists.stream()
                .flatMap(List::stream)
                .collect(Collectors.toList());

        System.out.println(flattenedNumbersList);
        // Output: [10, 20, 30, 40, 50, 60, 70, 80, 90]

        List<List<String>> listOfStringLists = Arrays.asList(
                Arrays.asList("a", "b"),
                Arrays.asList("c"),
                Arrays.asList("d", "e", "f")
        );

        List<String> flattenedStringList = listOfStringLists.stream()
                .flatMap(List::stream)
                .collect(Collectors.toList());

        System.out.println(flattenedStringList);
        // Output: [a, b, c, d, e, f]
    }
}
```

Explicación:

- `.flatMap(List::stream)`: Esta es una operación intermedia que transforma cada lista interna dentro de `listOfLists` en un flujo independiente con sus elementos.

  La operación `flatMap` luego aplana estos flujos individuales en un único flujo continuo de enteros. En efecto, toma un flujo de listas y lo convierte en un flujo con los elementos que contienen dichas listas.

- `.collect(Collectors.toList())`: Esta operación final recopila todos los elementos del flujo aplanado y los agrupa en una nueva lista `List<Integer>` cuyo nombre es `flattenedNumbersList`.

Referencias

- https://engineerscodinghub.com/top-java-8-stream-api-practice-problems
- https://codefinity.com/courses/v2/