Stream API

The Stream API introduced in Java 8 is a powerful feature that allows developers "**to process collections of objects**" in a functional style.

It is part of the **java.util.stream** package and is used to perform operations such as filtering, mapping, and reducing.

- Stream : A sequence of elements supporting sequential and parallel operations.

- Intermediate Operations:
    - Operations that return a stream (e.g., filter(), map(), sorted()).

- Terminal Operations :
    - Operations that produce a result or side-effect
    (e.g., collect(), forEach(), reduce()).

- Lazy Evaluation Intermediate operations are not executed until a terminal operation is invoked.

StreamDemo1.java

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

public class StreamDemo1 {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Venkat Srikanth", "Vishwa", "Vcube", "Java");

        // Creating a Stream
```

```
      Stream<String> nameStream = names.stream();

      // Example: filter names that start with 'V'
      List<String> filtered = nameStream
                              .filter(name -> name.startsWith("V"))
                              .collect(Collectors.toList());

      System.out.println(filtered);  // Output: [Venkat Srikanth, Vishwa, Vcube]
   }
}
```

Common Stream Operations

Operation	Description	Example
filter()	Filters elements	stream.filter(x -> x > 10)
map()	Transforms elements	stream.map(String::toUpperCase)
sorted()	Sorts elements	stream.sorted()
distinct()	Removes duplicates	stream.distinct()
limit(n)	Limits the stream to n elements	stream.limit(5)
collect()	Converts stream to list, set, etc.	collect(Collectors.toList())
forEach()	Performs action on each element	forEach(System.out::println)

reduce()	Reduces to a single value	stream.reduce(0, Integer::sum)

1) filter map reduce

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);
int sum = numbers.stream()
              .filter(n -> n % 2 == 0)
              .map(n -> n * n)
              .reduce(0, Integer::sum);
System.out.println(sum);  // Output: 56 (4 + 16 + 36)
```

2)filter even numbers

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6);

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

System.out.println(evenNumbers); // Output: [2, 4, 6]
```

3)Convert Strings to Uppercase

```
List<String> names = Arrays.asList("srikanth", "vcube", "java");
List<String> upperNames = names.stream()
                           .map(String::toUpperCase)
```

```
                          .collect(Collectors.toList());
System.out.println(upperNames); // Output: [SRIKANTH, VCUBE, JAVA]
```

4) Sort a List

```
List<String> fruits = Arrays.asList("Mango", "Banana", "Apple", "Orange");
List<String> sorted = fruits.stream()
                        .sorted()
                        .collect(Collectors.toList());
System.out.println(sorted); // Output: [Apple, Banana, Mango, Orange]
```

5) Count Elements Matching a Condition

```
long count = Stream.of("apple", "banana", "cherry")
                .filter(s -> s.contains("a"))
                .count();
System.out.println(count); // Output: 3
```

flatMap

In Java, flatMap is a method commonly used in **functional programming** with **Streams** to flatten nested structures, such as Stream<Stream<T>> into a single Stream<T>.

It's often used when each element of a stream needs to be transformed into **multiple**

elements, and you want to **flatten** the result.

Concept

- **map()**: Transforms each element into another **single** element.
- **flatMap()**: Transforms each element into a **stream** and then flattens all streams into a single one.

Syntax :

```
Stream<T> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)
```

6) FlatMap: Flatten Nested Lists

```
List<List<String>> nestedList = Arrays.asList(
    Arrays.asList("a", "b"),
    Arrays.asList("c", "d"),
    Arrays.asList("e")
);
List<String> flatList = nestedList.stream()
                              .flatMap(List::stream)
                              .collect(Collectors.toList());
System.out.println(flatList); // Output: [a, b, c, d, e]
```

7) Extracting words from sentences

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

```
public class FlatMapWords {
    public static void main(String[] args) {
        List<String> sentences = Arrays.asList("hello world", "java stream flatmap");
        List<String> words = sentences.stream()
            .flatMap(sentence -> Arrays.stream(sentence.split(" ")))
            .collect(Collectors.toList());

        System.out.println(words);  // Output: [hello, world, java, stream, flatmap]
    }
}
```

When to use flatMap?

Use flatMap when:

- Each element should map to **multiple** elements

  (e.g. from String to Stream<String>)

Find First Matching Element

```
Optional<String> first = Stream.of("one", "two", "three")
                               .filter(s -> s.length() == 3)
                               .findFirst();

first.ifPresent(System.out::println);
```

Group by with Collectors.groupingBy()

```
class Person {
    String name;
    String city;

    Person(String name, String city) {
        this.name = name;
        this.city = city;
    }

    String getCity() { return city; }
    String getName() { return name; }
}

List<Person> people = Arrays.asList(
    new Person("Alice", "New York"),
    new Person("Bob", "London"),
    new Person("Charlie", "New York")
);

Map<String, List<Person>> groupedByCity = people.stream()
    .collect(Collectors.groupingBy(Person::getCity));

groupedByCity.forEach((city, list) -> {
    System.out.println(city + ": " +

list.stream().map(Person::getName).collect(Collectors.joining(", ")));
});
```

Reduce to Calculate Product

```
int product = Stream.of(1, 2, 3, 4)
                    .reduce(1, (a, b) -> a * b);

System.out.println(product); // Output: 24
```

Remove Duplicates

```
List<Integer> nums = Arrays.asList(1, 2, 2, 3, 4, 4, 5);
List<Integer> unique = nums.stream()
                           .distinct()
                           .collect(Collectors.toList());
System.out.println(unique); // Output: [1, 2, 3, 4, 5]
```

Peek for Debugging

```
List<String> debug = Stream.of("apple", "banana", "cherry")
    .peek(s -> System.out.println("Processing: " + s))
    .map(String::toUpperCase)
    .collect(Collectors.toList());
```

What is a Parallel Stream?

- A **Parallel Stream** divides the content into multiple chunks and processes them **in parallel** using multiple threads.

  (from the **ForkJoinPool.commonPool()** by default).
- It can significantly improve performance for large data sets or CPU-intensive tasks.

Simple Parallel Stream Example

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

public class ParallelStreamExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("John", "Jane", "Jack",
"Jill", "Jerry", "Jim");

        names.parallelStream()
             .forEach(name ->
System.out.println(Thread.currentThread().getName() + " - " + name));
    }
}
```

Performance Comparison (Sequential vs Parallel)

```
import java.util.stream.IntStream;

public class ParallelVsSequential {
    public static void main(String[] args) {
        long start, end;

        // Sequential
        start = System.currentTimeMillis();
        IntStream.range(1, 1_000_000)
                 .sum();
        end = System.currentTimeMillis();
        System.out.println("Sequential took: " + (end - start) +
"ms");

        // Parallel
        start = System.currentTimeMillis();
        IntStream.range(1, 1_000_000)
                 .parallel()
                 .sum();
        end = System.currentTimeMillis();
        System.out.println("Parallel took: " + (end - start) + "ms");
    }
}
```

Preserving Order in Parallel Stream

Use `.forEachOrdered()` instead of `.forEach()` if you need to maintain order:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

numbers.parallelStream()
       .forEachOrdered(System.out::println);
```

Parallel Reduce Operation

```
int sum = IntStream.range(1, 1000)
                   .parallel()
                   .reduce(0, Integer::sum);
System.out.println("Sum: " + sum);
```

Using Parallel with Map and Collect

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

List<String> upperWords = words.parallelStream()
                               .map(String::toUpperCase)
```

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

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