Java8 Streams

If we want to represent a group of objects as a single entity then we should go for collection.

But if we want to process objects from the collection then we should go for streams.

In Java, **Stream** is an **interface** and streams are present in java's utility package named **java.util.stream.**

Stream API is used to process collections of objects. Streams are designed to be efficient and can support improving your program's performance by allowing you to avoid unnecessary loops and iterations. **Streams can be used for filtering, collecting, printing, and converting from one data structure to another,** etc.

The features of Java stream are mentioned below:

- A stream is not a data structure, instead it takes input from the Collections, Arrays or I/O channels.
- Streams don't change the original data structure, they only provide the result as per the pipelined methods..

Functional Style:

Streams support functional programming principles, allowing developers to pass functions as arguments, which enhances code readability and reduces boilerplate.

Lazy Evaluation:

Intermediate operations (like filter and map) are lazy. They do not process elements until a terminal operation (like collect, forEach, or reduce) is called. This can lead to performance optimizations, as only the necessary elements are processed.

Pipeline of Operations:

Streams can be chained together to form a pipeline, making it easy to compose multiple operations in a readable way.

Parallel Processing:

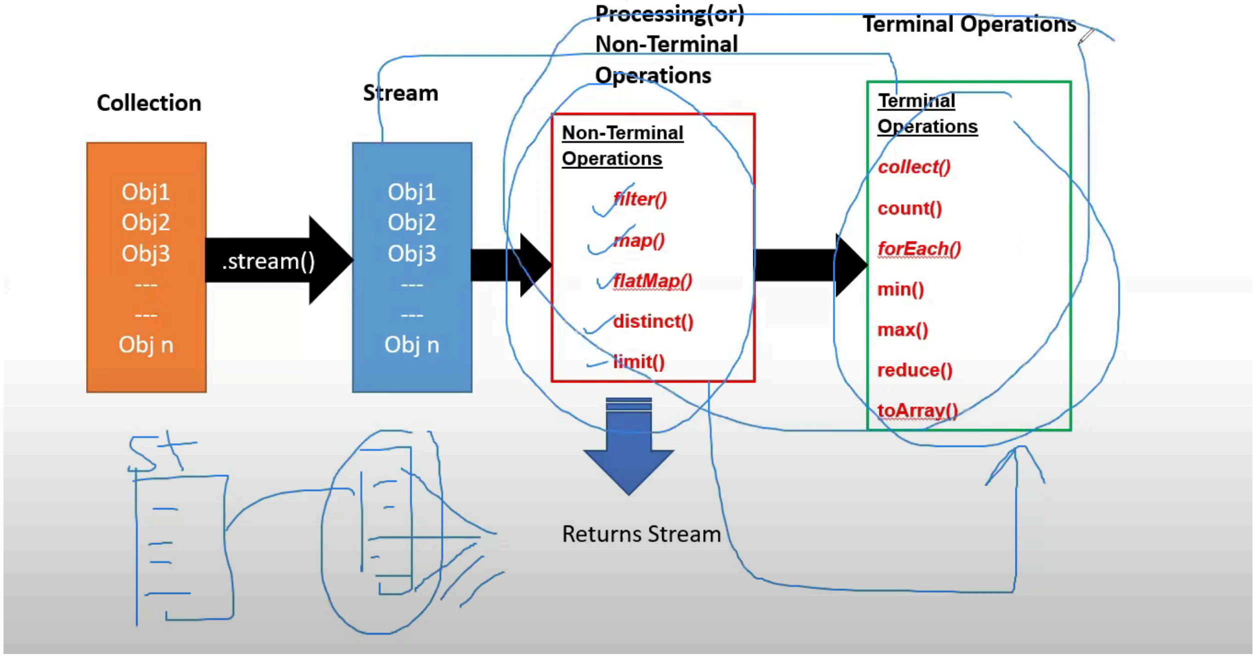
Streams can be executed in parallel using the parallelStream() method. This allows developers to leverage multi-core processors easily, improving performance for large datasets.

Immutable and Stateless:

Streams do not modify the source data structure. They produce new streams as the result of operations, maintaining immutability.

Types of Stream Operations

1. **Intermediate Operations**: These are operations that return another stream and can be chained together. Examples include **filter, map ,flatMap ,distinct ,limit ,sorted and skip**
2. **Terminal Operations**: These operations produce a result or side effect and end the stream. Examples include **collect, forEach, reduce ,count ,min ,max , toArray anyMatch**, **allMatch**, and **noneMatch**

3. 

We can also apply terminal operation directly on stream. But it depends on the operation we will use . mostly first will apply intermediate and then terminal.

Comparison of java.io.Stream and java.util.stream:

Feature	**java.io.Stream**	**java.util.stream**
Purpose	This refers to the traditional I/O classes that handle input and output operations. It includes classes for reading from and writing to files, network sockets, and other data sources. Common classes include InputStream, OutputStream, FileInputStream, and FileOutputStream.	This package is focused on the Stream API introduced in Java 8. It provides a functional approach to processing sequences of data, such as collections, arrays, or I/O streams. The key class here is Stream<T>, which allows for operations like filtering, mapping, and reducing data.

Functionality	Provides methods for reading and writing data in a byte-oriented or character-oriented way. Designed for handling raw data, such as bytes, characters, or objects directly.	Enables processing of data in a functional style, allowing for operations such as transformations, filtering, and aggregations. It operates on collections of data and provides a fluent API for composing complex data processing pipelines.
Usage Context	Used when you need to perform file I/O or interact with data sources directly. Typically involves reading from and writing to various data formats	Used when you want to manipulate and process collections of data in a more abstract and high-level way. Ideal for operations like searching, filtering, and mapping collections.

Examples	try (InputStream inputStream = new FileInputStream("file.txt")) {
	    int data;
	    while ((data = inputStream.read()) != -1) {
	        System.out.print((char) data);
	    }
	} catch (IOException e) {
		List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
		List<String> filteredNames = names.stream()
		    .filter(name -> name.startsWith("A"))
		    .collect(Collectors.toList());
		// Result: ["Alice"]

	e.printStackTrace();
	}	

Filters : Which filter the data based on condition.

This method takes a predicate as an argument and returns a stream consisting of resulting elements. **Stream filter(Predicate predicate)** returns a stream consisting of the elements of this stream that match the given predicate. This is an ***intermediate operation.*** These operations are always lazy i.e, In Java Streams, operations like filter() are indeed lazy. This means that they don't perform their filtering operation immediately when invoked. Instead, they set up a pipeline of operations that will only be executed when a terminal operation is invoked.

Stream<T> filter(Predicate<? super T> predicate);

Syntax: stream.filter(predicate)

Ex1

```
package com.filters;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
//This method takes a predicate as an argument and returns a stream consisting of resulting
elements.
public class FiltersDemo1 {
        public static void main(String[] args) {
```

```
        /*
         * ArrayList<Integer> number = new ArrayList<Integer>(); number.add(10);
         * number.add(15); number.add(20); number.add(25); number.add(30);
         * number.add(35); number.add(40);
         */
        // or we can create collection using Array.asList
        List<Integer> num = Arrays.asList(2, 3, 4, 5, 6, 7, 8, 9, 10);
        // without streams filtering condition : even numbers from collection using for
        // each loop.
        // empty arryaList for storing results
        System.out.println("without streams filtering condition : even numbers from
collection using for each loop");
        ArrayList<Integer> result = new ArrayList<Integer>();
        for (Integer n : num) {
            if (n % 2 == 0) {
                result.add(n);
            }
        }
        System.out.println(result);
        // using streams
        List<Integer> list = Arrays.asList(12, 13, 14, 15, 16, 17, 18, 19, 20);
        // creating empty list for storing result
        List<Integer> res = new ArrayList<Integer>();
        System.out.println(" using streams and storing results in new collection.");
        res = list.stream().filter(n -> n % 2 == 0).collect(Collectors.toList());
        System.out.println(res);
        // Explanation :
        // Stream Creation: numbers.stream() creates a stream from the list of numbers.
        // Filter Operation: The filter method takes a predicate (a functional interface
        // that returns a boolean) and returns a new stream that contains only the
        // elements that match the predicate.
        // Collecting Results: Finally, collect(Collectors.toList()) gathers the
        // filtered elements into a new list.
        // we can also use same expression, without storing results in new collection.
        System.out.println("using streams ,same expression, without storing results in
new collection.");
        list.stream().filter(n -> n % 2 == 0).forEach(System.out::println); // directly
printing results in console

/*****************************************************************************
*/
        // Ex 2: filter() method with the operation of filtering out the elements
        // divisible by 5.
        System.out.println("filter() method with the operation of filtering out the elements
divisible by 5");
        // Creating a list of Integers
        List<Integer> l = Arrays.asList(3, 4, 6, 12, 20);
        // Getting a stream consisting of the
        // elements that are divisible by 5
        // Using Stream filter(Predicate predicate)
        l.stream().filter(a -> a % 5 == 0).forEach(System.out::println);

/*****************************************************************************
**/
        // EX 3 : filter() method with the operation picking the elements ending with
        // custom alphabetically letter say it be ‘s’ for implementation purposes.
        System.out.println("filtering string elements ending with 's'");
        List<String> li = Arrays.asList("dhoni", "virat", "ram", "sam", "madam");
        List<String> r = li.stream().filter(b ->
b.endsWith("m")).collect(Collectors.toList());
```

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

Output

```
without streams filtering condition : even numbers from collection using for each loop
[2, 4, 6, 8, 10]
using streams and storing results in new collection.
[12, 14, 16, 18, 20]
using streams ,same expression, without storing results in new collection.
12
14
16
18
20
filter() method with the operation of filtering out the elements divisible by 5
20
filtering string elements ending with 's'
[ram, sam, madam]
```

Ex 2:

```
package com.filters;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
public class FilterDemo2 {
	public static void main(String[] args) {
		// Updated list with names of various lengths
		List<String> names = Arrays.asList("pavankumar", "raman", "rajashekar", "jay", "lucas");
		// Want to filter the names having length more than 4 and less than 6
```

```
                List<String> result = names.stream().filter(n -> n.length() > 4 && n.length() < 6) // Filter names
with length >

                                                                                    // 4 and < 6
                        .collect(Collectors.toList()); // Collect the result into a new list
                System.out.println(result);
                // if want to just print instead of storing collection
                System.out.println("want to just print instead of storing collection");
                names.stream().filter(n -> n.length() > 4 && n.length() < 6).forEach(n->System.out.println(n));
                //      names.stream().filter(n -> n.length() > 4 && n.length() < 6).forEach(System.out::println);



        //want remove null values and printing non-null values.
                System.out.println("want remove null values and printing non-null values.");
                List<String> na = Arrays.asList("dhoni", "raina", null, "virat",null);
                na.stream().filter(a-> a!=null).forEach(a ->System.out.println(a));


        }
}
```

Output

```
[raman, lucas]
want to just print instead of storing collection
raman
lucas
want remove null values and printing non-null values.
dhoni
raina
virat
```

Ex3

```
package com.filters;
import java.util.ArrayList;
import java.util.List;
class Product {
        int id;
```

```
    String name;
    double price;
    public Product(int id, String name, double price) {
        super();
        this.id = id;
        this.name = name;
        this.price = price;
    }
}
public class FilterDemo3 {
    public static void main(String[] args) {
        List<Product> p = new ArrayList<Product>();
        p.add(new Product(1, "hp", 20000));
        p.add(new Product(2, "lenova", 340000));
        p.add(new Product(3, "asus", 29000));
        p.add(new Product(4, "dell", 30000));
        p.stream().filter(a -> a.price > 25000).forEach(b -> System.out.println(b.price));
    }
}
```

Output

```
340000.0
29000.0
30000.0
```

MAP:

The map() function in Java Streams is a powerful tool for transforming the elements of a stream. It allows you to apply a function to **each element**, producing a new stream with the transformed values. It’s widely used for:

- Converting data types.
- Extracting fields from objects.
- Applying mathematical transformations.

You can chain map() with other stream operations like filter(), reduce(), and collect()

Stream map(Function mapper) is an **intermediate operation**. These operations are always lazy. Intermediate operations are invoked on a Stream instance and after they finish their processing, they give a Stream instance as output.

Stream<R> map(Function<? super T, ? extends R> mapper);

```
package com.map;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
public class Mapdemo1 {
    public static void main(String[] args) {
        System.out.println("Cubing Each Element in a List: ");
  // Creating a list of Integers
        List<Integer> list = Arrays.asList(3, 6, 9, 12, 15);
        list.stream().map(n -> n * n * n).forEach(System.out::println);
        // EX2
        System.out.println("convert the Strings to UpperCase: ");
        List<String> list1 = Arrays.asList("ramesh", "king", "pavan", "vamsi", "vinay", "k", "s");
        List<String> answer = list1.stream().map(s -> s.toUpperCase()).collect(Collectors.toList());
        System.out.println(answer);

  //EX3
        System.out.println("Squaring the Numbers and Filtering Negative Values:");
        // : Squaring the Numbers and Filtering Negative Values
        List<Integer> numbers = Arrays.asList(1, -2, 3, -4, 5);
        // SysoutSquare the numbers and remove negative results using filter
        List<Integer> squaredPositiveNumbers = numbers.stream().map(n -> n * n) // Square each number
                .filter(n -> n >= 0) // Keep only non-negative numbers (this is redundant for
squaring)
                .collect(Collectors.toList());
        System.out.println(squaredPositiveNumbers);
    }
}
```

Output

```
Cubing Each Element in a List:
27
216
729
1728
3375
convert the Strings to UpperCase:
[RAMESH, KING, PAVAN, VAMSI, VINAY, K, S]
Squaring the Numbers and Filtering Negative Values:
[1, 4, 9, 16, 25]
```

Ex2

```
package com.map;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
public class MapDemo2 {
        public static void main(String[] args) {
                List<String> vehicles = Arrays.asList("Toyota", "Honda", "Ford", "BMW", "Audi", "Tesla");
//before java 8
                /*
                 * for ( String str:vehicles) { System.out.println(str.length()); }
                 */
                System.out.println("finding length of vehicle:");
                vehicles.stream().map(s -> s.length()).forEach(s -> System.out.println(s));
                System.out.println("finding length of vehicle start with prefix  T :");
                vehicles.stream().filter(a -> a.startsWith("T")).map(s -> s.length()).forEach(s ->
System.out.println(s));
                System.out.println("finding length of vehicle end with suffix  a :");
                vehicles.stream().filter(a -> a.endsWith("a")).map(s -> s.length()).forEach(s ->
System.out.println(s));
                System.out.println("mathematical operation :");
                List<Integer> list = Arrays.asList(3, 6, 9, 12, 15);
                list.stream().filter(s -> s > 1 && s < 10).map(s -> s % 2).forEach(s -> System.out.println(s));
                System.out.println("Mapping First, Filtering Second");
                // List of numbers
                List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
                // First map each number to its square, then filter out squares greater than 50
                List<Integer> result = numbers.stream().map(n -> n * n) // Map: square each number
                        .filter(n -> n <= 50) // Filter: keep only squares <= 50
                        .collect(Collectors.toList());
                System.out.println(result);
        }
}
```

Output

```
finding length of vehicle:
6
5
4
3
4
5
finding length of vehicle start with prefix  T :
6
5
finding length of vehicle end with suffix  a :
6
5
5
mathematical operation :
1
0
1
Mapping First, Filtering Second
[1, 4, 9, 16, 25, 36, 49]
```

EX3

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
class Employee {
    String name;
    double salary;
    Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
    }
    public String getName() {
        return name;
    }
    public double getSalary() {
        return salary;
    }
}
public class EmployeeFilterMapExample {
    public static void main(String[] args) {
        List<Employee> employees = Arrays.asList(new Employee("ram", 75000), new Employee("kl",
50000),
                new Employee("dhoni", 120000), new Employee("virat", 90000));
        List<String> highEarners = employees.stream().filter(emp -> emp.getSalary() > 80000) // Filter
employees with

                                            // salary > 80000
                .map(emp -> emp.getName()) // Map to employee names
                .collect(Collectors.toList());
        System.out.println(highEarners); // Output: [Charlie, David]
    }
}
```

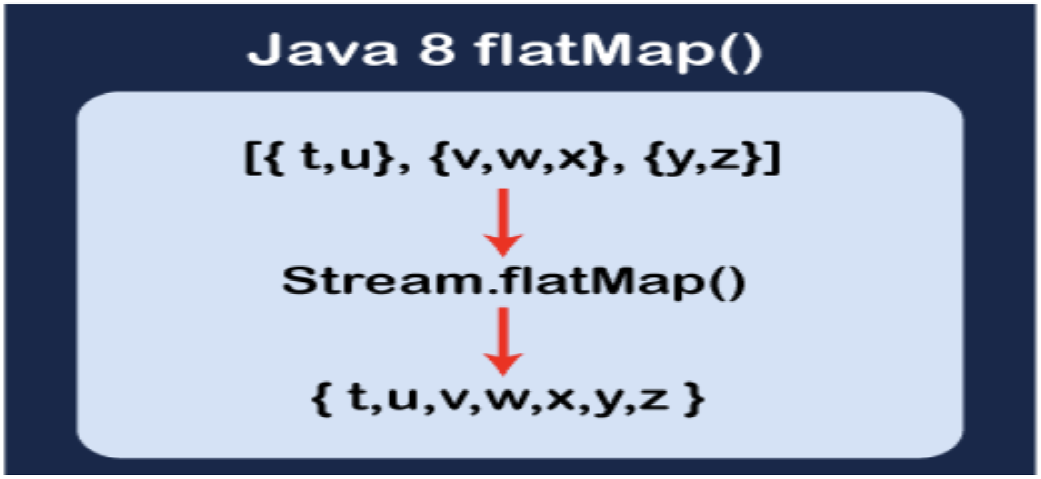
Output

```
[dhoni, virat]
```

Flapt Map

The flatMap() method applies operation as a mapper function and provides a stream of element values. It means that in each iteration of each element the map() method creates a separate new stream. By using the flattening mechanism, it merges all streams into a single resultant stream. In short, it is used to convert a Stream of Stream into a list of values.

Flattening is the process of converting several lists of lists and merging all those lists to create a single list containing all the elements from all the lists.
Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper);

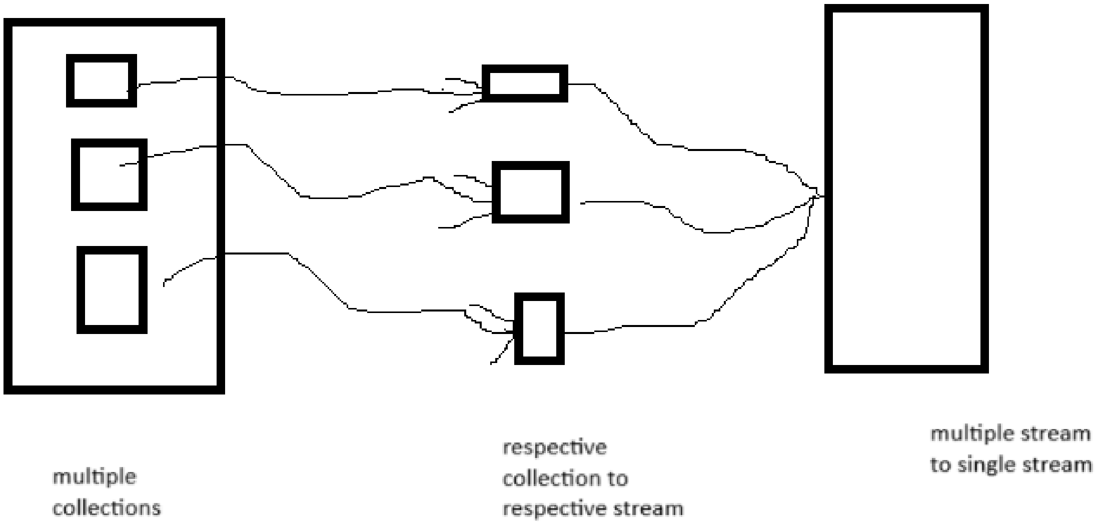


Java 8 flatMap()

[{ t,u}, {v,w,x}, {y,z}]

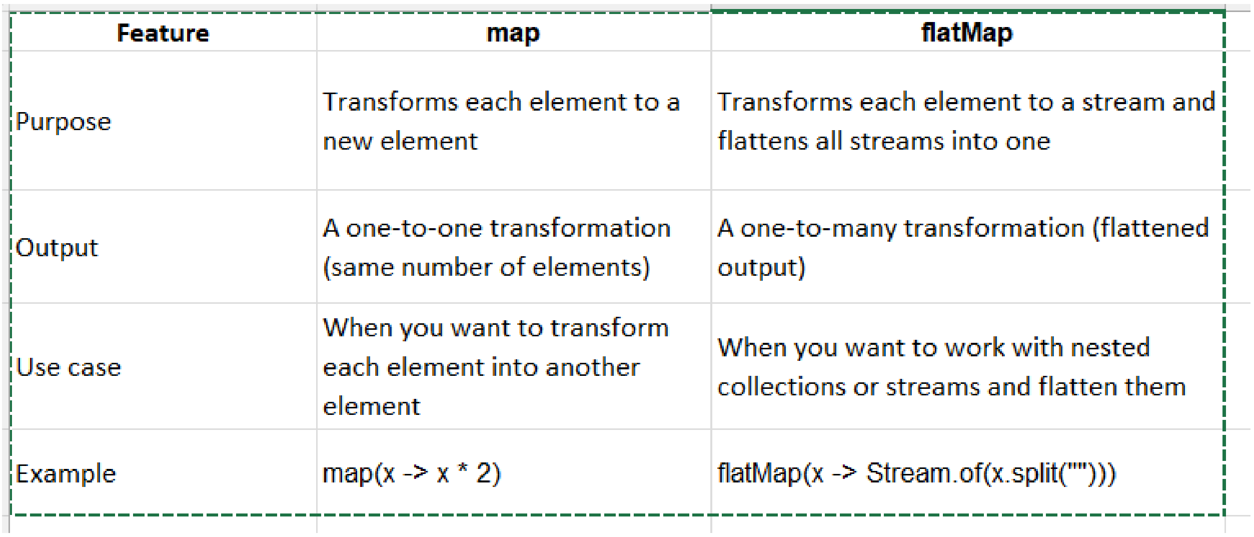
Stream.flatMap()

{ t,u,v,w,x,y,z }



flatMap() V/s map():

- map() transforms each element of a stream into another object, resulting in a stream of the same size as the input. It’s used for one-to-one transformations.
- flatMap() transforms each element of a stream into zero or more elements, potentially changing the size of the stream. It’s used for one-to-many transformations and flattening nested structures.



Feature	map	flatMap
Purpose	Transforms each element to a new element	Transforms each element to a stream and flattens all streams into one
Output	A one-to-one transformation (same number of elements)	A one-to-many transformation (flattened output)
Use case	When you want to transform each element into another element	When you want to work with nested collections or streams and flatten them
Example	map(x -> x * 2)	flatMap(x -> Stream.of(x.split("")))

When to use map vs flatMap:

- Use **map** when you want to convert each element in the stream into another element, maintaining the same number of elements in the stream.
- Use **flatMap** when each element in the stream results in a sequence (e.g., a list or a stream), and you want to flatten those sequences into a single stream.

Ex1

```
package com.flatmap;
import java.util.*;
import java.util.stream.*;
public class FlatMapExample {
        public static void main(String[] args) {
                /*
```

```
         * List<String> list1 = new ArrayList<String>(); list1.add("apple");
         * list1.add("banana");
         */
        List<String> list1 = Arrays.asList("apple", "banana");
        List<String> list2 = Arrays.asList("orange", "pear");
        List<String> list3 = Arrays.asList("grape", "kiwi");
        // Combine the lists into a List<List<String>>
        List<List<String>> listOfLists = Arrays.asList(list1, list2, list3);
        System.out.println(listOfLists);
        /*
         * System.out.println("before java 8"); for (List<String> a : listOfLists) {
         *
         * for (String s : a) { System.out.println(s); } }
         */
        // or
        /*
         * System.out.println("before java 8");
         *
         * List<String> finalList = new ArrayList<String>(); finalList.addAll(list1);
         * finalList.addAll(list2); finalList.addAll(list3);
         * System.out.println(finalList);
         */
        System.out.println("after java 8");
        // Using flatMap to flatten the List of Lists
        List<String> flattenedList = listOfLists.stream().flatMap(l ->
l.stream()).collect(Collectors.toList());
        System.out.println(flattenedList);
    }
}
```

Output

```
[[apple, banana], [orange, pear], [grape, kiwi]]
after java 8
[apple, banana, orange, pear, grape, kiwi]
```

Ex2

```
package com.flatmap;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
public class CricketerFlatMapExample {
        public static void main(String[] args) {
                // Three separate lists of cricketer names
                List<String> list1 = Arrays.asList("Sachin Tendulkar", "Rahul Dravid", "Sourav Ganguly");
                List<String> list2 = Arrays.asList("Virat Kohli", "Rohit Sharma", "Shikhar Dhawan");
                List<String> list3 = Arrays.asList("MS Dhoni", "Yuvraj Singh", "Zaheer Khan");
                // Create an empty list of lists and add list1, list2, and list3 to it
                List<List<String>> cricketerLists = new ArrayList<List<String>>();
                cricketerLists.add(list1);
                cricketerLists.add(list2);
                cricketerLists.add(list3);
                // Combine all three lists into a single list of lists
                // List<List<String>> cricketerLists = Arrays.asList(list1, list2, list3);
                // want print Criceter name , before java 8
                /*
                 * for (List<String> a : cricketerLists) {
                 *
                 * for (String s : a) { System.out.println(s); } }
                 */
                // Use flatMap to flatten the list of lists
                List<String> allCricketers = cricketerLists.stream().flatMap(list -> list.stream()) // Flatten each
inner list
                                .collect(Collectors.toList());
                System.out.println(allCricketers);
        }
}
```

Output

```
[Sachin Tendulkar, Rahul Dravid, Sourav Ganguly, Virat Kohli, Rohit Sharma, Shikhar Dhawan, MS Dhoni, Yuvraj Singh, Zaheer Khan]
```

Ex3

```
package com.flatmap;
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;
// Define the Student class
class Student {
    int id;
    String name;
    String grade;
    // Constructor
    public Student(int id, String name, String grade) {
        this.id = id;
        this.name = name;
        this.grade = grade;
    }
}
public class FlatMapEx {
    public static void main(String[] args) {
        // Create a list to store Student objects
        List<Student> studentList1 = new ArrayList<>();
        // Add Student objects to the list
        studentList1.add(new Student(1, "raju", "A"));
        studentList1.add(new Student(2, "ramesh", "B"));
        studentList1.add(new Student(3, "rakesh", "A"));
        studentList1.add(new Student(4, "rithu", "C"));
        List<Student> studentList2 = new ArrayList<>();
        // Add Student objects to the list
        studentList2.add(new Student(5, "gani", "A"));
        studentList2.add(new Student(6, "gouri", "B"));
```

```
                    studentList2.add(new Student(7, "ganga", "A"));
                    studentList2.add(new Student(8, "ganesh", "C"));
                    // adding list of student(list<student>) collections to new list collection(list
                    // )
                    List<List<Student>> finalList = new ArrayList<List<Student>>();
                    finalList.add(studentList1);
                    finalList.add(studentList2);
                    // or List<List<Student>> finalList = Arrays.asList(studentList1, studentList2);
                    // want print student name , before java 8
                    /*
                     * for(List<Student> a:finalList ) {
                     *
                     * for(Student s : a) { System.out.println(s.name); } }
                     */
                    // USING JAVA8
                    List<String> result = finalList.stream().flatMap(s -> s.stream()).map(s ->
s.name).collect(Collectors.toList());
                    System.out.println(result);
          }
}
```

Output

```
[raju, ramesh, rakesh, rithu, gani, gouri, ganga, ganesh]
```

Distinct,Count

The distinct() method is used to eliminate duplicate elements from a stream.

The count() method is a terminal operation that returns the number of elements in a stream. It is often used to determine the size of a stream or to count the elements

```
package com.distinct;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
public class DistinctExample {
        public static void main(String[] args) {
                // EX1 : Removing Duplicates from a List of Integers
                List<Integer> numbers = Arrays.asList(1, 2, 2, 3, 4, 4, 5);
                System.out.println("List of Integers:" + numbers);
                List<Integer> distinctNumbers = numbers.stream().distinct().collect(Collectors.toList());
                // numbers.stream().distinct().forEach(s -> System.out.println(s));
                System.out.println("Removing Duplicates from a List of Integers: " + distinctNumbers); //
Output: [1, 2, 3, 4,

                                                                                                  // 5]
                // EX2
                List<String> words = Arrays.asList("apple", "banana", "Apple", "banana", "cherry");
                List<String> distinctWords = words.stream().distinct().collect(Collectors.toList());
                System.out.println(distinctWords);
                // count
                List<Integer> coun = Arrays.asList(1, 2, 2, 3, 4, 4, 5);
                long uniqueCount = coun.stream().distinct().count();
                System.out.println("Number of unique values: " + uniqueCount);



        }
}
```

Output

```
List of Integers:[1, 2, 2, 3, 4, 4, 5]
Removing Duplicates from a List of Integers: [1, 2, 3, 4, 5]
[apple, banana, Apple, cherry]
Number of unique values: 5
```

Limit,min,max

Limit: It is used to limit the number of elements in the stream to the first n elements.

Min:return the minimum values of the elements in a stream, respectively, based on a provided comparator

Max: return the maximum values of the elements in a stream, respectively, based on a provided comparator

The min and max methods in Java Streams return an Optional<T> to handle cases where a result might not be available. Here’s why:

1. Handling Empty Streams

- If the stream is empty, there is no minimum or maximum element to return. Instead of returning null, which can lead to NullPointerException, Optional<T> allows Java to handle empty cases gracefully.
- With Optional, you can check if a result exists using methods like .isPresent() or .ifPresent(), reducing the risk of runtime errors.

2. Avoiding NullPointerException

- Using Optional reduces the need for null checks, as the result is wrapped in an object with methods to safely access the value or handle the absence of it.
- Optional provides methods like .orElse() or .orElseThrow() to define default values or throw exceptions if the value is absent.

```
package com.limit;
import java.util.Arrays;
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;
public class LimitExample {
        public static void main(String[] args) {
```

```
		List<Integer> numbers = Arrays.asList(10, 20, 30, 40, 50, 60);
		List<Integer> limitedNumbers = numbers.stream().limit(3).collect(Collectors.toList());
		System.out.println(limitedNumbers); // Output: [10, 20, 30]
		// ex2: Given a list of integers, filter out the even numbers and return only
		// the first 2 even numbers.
		List<Integer> b = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);
		List<Integer> limitedEvens = b.stream().filter(n -> n % 2 ==
0).limit(2).collect(Collectors.toList());
		System.out.println(limitedEvens); // Output: [2, 4]
		// ex3: min() and max

		/*
		 * If the stream is empty, there is no minimum or maximum element to return.
		 * Instead of returning null, which can lead to NullPointerException,
		 * Optional<T> allows Java to handle empty cases gracefully. With Optional, you
		 * can check if a result exists using methods like .isPresent() or .ifPresent(),
		 * reducing the risk of runtime errors.
		 */
		List<Integer> ab = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);
		Optional<Integer> mi = ab.stream().min((x, y) -> {
			return x.compareTo(y);
		});
		System.out.println(mi.get());
		// max
		Optional<Integer> ma = ab.stream().max((j, k) -> {
			return j.compareTo(k);
		});
		System.out.println(ma.get());
	}
}
```

Output

```
[10, 20, 30]
[2, 4]
1
```

Reduce

The reduce method in Java Streams is a **terminal operation** that performs a reduction on the elements of the stream using an associative accumulation function and returns a single result. It can be used for various operations like summing, multiplying, concatenating strings, finding the maximum or minimum, and many other types of aggregation.

```
package com.limit;
import java.util.Arrays;
import java.util.List;
import java.util.Optional;
public class ReduceEx {
        public static void main(String[] args) {
                List<String> names = Arrays.asList("gopi", "ramesh", "dachin");
                // Using reduce to concatenate strings with a hyphen
                Optional<String> combinedString = names.stream().reduce((str1, str2) -> {
                        return str1 + str2;
                });
                // Displaying the combined String
                System.out.println(combinedString.get());
                /*
                 * List<String> names1 = Arrays.asList(); // Using reduce to concatenate strings
                 * with a hyphen Optional<String> combinedString1 =
                 * names1.stream().reduce((str1, str2) -> { return str1 + str2; });
                 *
                 * // Displaying the combined String System.out.println(combinedString1.get());
                 */
                /*
                 * Uses an identity element (initial value) and a binary function to reduce the
                 * elements. Returns T, without Optional, because the identity value is returned
                 * if the stream is empty.
                 */
                List<Integer> su = Arrays.asList(1, 2, 3, 4, 5);
                Integer sum = su.stream().reduce(0, (a, b) -> a + b);
                System.out.println(sum); // Output: 15
    //without optional if the stream is empty , it will print initial value
                List<Integer> add = Arrays.asList();
                Integer sum1 = add.stream().reduce(0, (a, b) -> a + b);
                System.out.println(sum1); // Output: 15
                List<Integer> multi = Arrays.asList(1, 2, 3, 4, 5);
                Integer product = multi.stream().reduce(1, (a, b) -> a * b);
                System.out.println(product); // Output: 120
        }
}
```

Output

```
gopirameshdachin
15
0
120
```

toArray

```
package com.limit;
import java.util.List;
import java.util.stream.Stream;
public class ToArrayExample {
  public static void main(String[] args) {
    List<String> names = List.of("Alice", "Bob", "Charlie");
    Object[] namesArray = names.stream().toArray();


    for (Object name : namesArray) {
      System.out.println(name); // Output: Alice, Bob, Charlie
    }
  }
}
```

Output

```
Alice
Bob
Charlie
```

Sorted

```
package com.streammethods;
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;
import java.util.stream.Collectors;
public class SortedExamples {
        public static void main(String[] args) {
                List<Integer> list = Arrays.asList(56, 34, 2, 3, 6, 9, 8, 0);
                List<Integer> Sortedlist = list.stream().sorted().collect(Collectors.toList());
                System.out.println(Sortedlist);// BY default it will sort in ascending order.
                System.out.println("reverse order or descending order");
                list.stream().sorted(Comparator.reverseOrder()).forEach(s -> System.out.println(s));
                // String
                List<String> names = Arrays.asList("rahul", "gopi", "shekar", "seetha", "zeeva");
                System.out.println(names.stream().sorted().collect(Collectors.toList()));

System.out.println(names.stream().sorted(Comparator.reverseOrder()).collect(Collectors.toList()));
        }
}
```

Output

```
2
0
[gopi, rahul, seetha, shekar, zeeva]
[zeeva, shekar, seetha, rahul, gopi]
```

anyMatch ,allMatch ,noneMatch

anyMatch()

- **Purpose**: Returns true if any element of the stream matches the given predicate (condition). If at least one element satisfies the condition, it returns true; otherwise, it returns false.
- **Signature**: boolean anyMatch(Predicate<? super T> predicate)

allMatch()

- **Purpose**: Returns true if **all** elements in the stream match the given predicate. If all elements satisfy the condition, it returns true; otherwise, it returns false.
- **Signature**: boolean allMatch(Predicate<? super T> predicate)

noneMatch()

- **Purpose**: Returns true if **none** of the elements in the stream match the given predicate. If no elements satisfy the condition, it returns true; otherwise, it returns false.
- **Signature**: boolean noneMatch(Predicate<? super T> predicate)

Key Differences:

- anyMatch(): Returns true if **at least one** element matches the predicate.
- allMatch(): Returns true if **all** elements match the predicate.
- noneMatch(): Returns true if **no elements** match the predicate.

```
package com.streammethods;
import java.util.Arrays;
import java.util.List;
public class Demo {
	public static void main(String[] args) {
		// anymatch
		List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
		boolean result = numbers.stream().anyMatch(n -> n > 3);
		System.out.println(result);
		List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");
		// Check if any name starts with 'A'
```

```
        boolean anyMatch = names.stream().anyMatch(name -> name.startsWith("A"));
        System.out.println(anyMatch); // Output: true
        // allmatch
        // List<Integer> number = Arrays.asList(1, 2, 3, 4, 5); //true
        List<Integer> number = Arrays.asList(1, 2, 3, 4, 5, 6); // false
        boolean resul = number.stream().allMatch(n -> n < 6);
        System.out.println(resul); // Output: (all numbers are less than 6)
        List<String> na = Arrays.asList("Alice", "Alex", "Adam");
        // Check if all names start with 'A'
        boolean allMatch = na.stream().allMatch(name -> name.startsWith("A"));
        System.out.println(allMatch); // Output: true
        // nonematch
        List<Integer> numb = Arrays.asList(1, 2, 3, 4, 5);
        boolean res = numb.stream().noneMatch(n -> n > 5);
        System.out.println(res); // Output: true (no numbers are greater than 5)
        List<String> namess = Arrays.asList("Alice", "Bob", "Charlie", "David");
        // Check if no name starts with 'Z'
        boolean noneMatch = namess.stream().noneMatch(name -> name.startsWith("Z"));
        System.out.println(noneMatch); // Output: true
    }
}
```

Output

```
true
true
false
true
true
true
```

findAny , findFirst

findAny()

- **Purpose**: Returns an Optional<T> describing **any** element of the stream. This is especially useful in parallel streams, where it might return **any** element encountered first during execution.
- **Characteristics**:
    - May return different elements on multiple executions for unordered streams

findFirst()

- **Purpose**: Returns an Optional<T> describing the **first** element of the stream. For ordered streams, it is guaranteed to return the first element.
- **Characteristics**:
    - Always retrieves the first element for ordered streams.

```
package com.streammethods;
import java.util.Arrays;
import java.util.List;
import java.util.Optional;
public class Demo2 {
        public static void main(String[] args) {
                // findAny
                List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");
                Optional<String> anyName = names.stream().findAny();
                System.out.println(anyName.get()); // Output: Could be any element, e.g., "Alice"

                //findFirst

    Optional<String> firstName = names.stream().findFirst();
    System.out.println(firstName.get()); // Output: "Alice"
        }
}
```

Output

```
Alice
Alice
```

Stream concatenation

We can concatenate two or more streams using the `Stream.concat()` method or by other means.

1. Using `Stream.concat()`

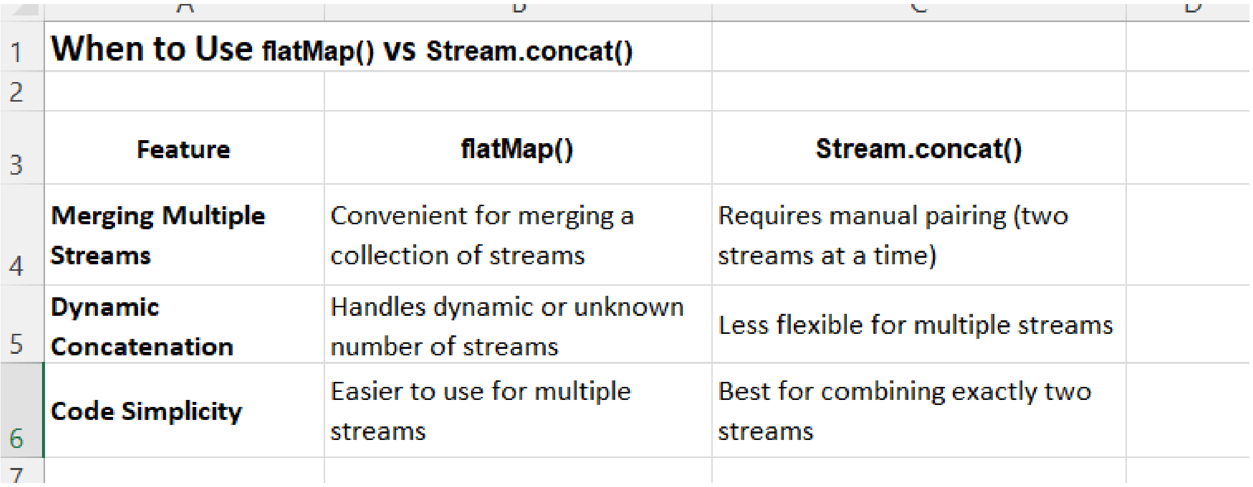
The `Stream.concat()` method combines two streams into a single stream.

```
package com.streammethods;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;
// Streams in Java are immutable. Once a stream is created, you cannot directly add or modify its elements
//Stream<String> stream1 = Stream.of("Apple", "Banana");
//Stream<String> stream2 = Stream.of("Cherry", "Date");
public class StreamConcatExample {
        public static void main(String[] args) {
                List<String> names = Arrays.asList("ramesh", "raju", "ravi");
                List<String> names1 = Arrays.asList("mahi", "mani", "ma");
                Stream<String> stream1 = names.stream();
                Stream<String> stream2 = names1.stream();
                List<String> res = Stream.concat(stream1, stream2).collect(Collectors.toList());
                for (String s : res) {
                        System.out.println(s);
                }
                /*
                 * Stream<String> stream1 = Stream.of("Alice", "Bob"); Stream<String> stream2 =
                 * Stream.of("Charlie", "David");
                 *
                 * Stream<String> concatenatedStream = Stream.concat(stream1, stream2);
```

```
                         *
                         * concatenatedStream.forEach(System.out::println);
                         */
                        // Output:
                        // Alice
                        // Bob
                        // Charlie
                        // David
            }
}
```

Output

```
ramesh
raju
ravi
mahi
mani
ma
```



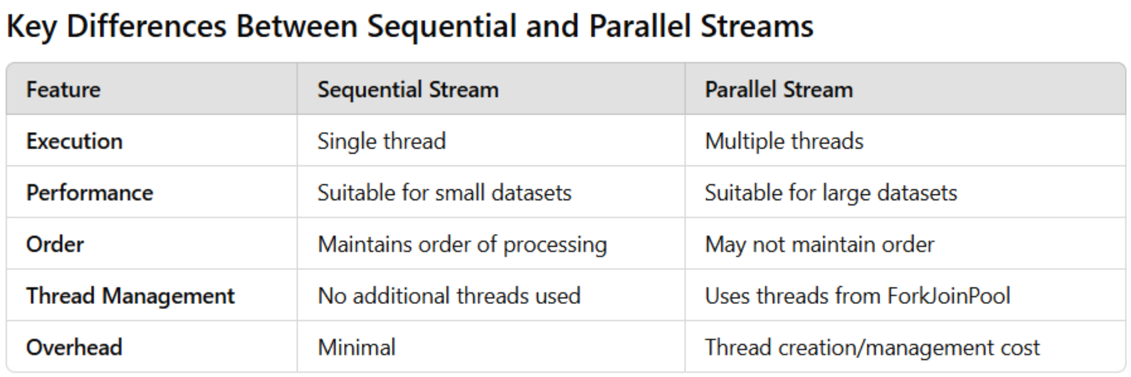
When to Use flatMap() vs Stream.concat()		
Feature	**flatMap()**	**Stream.concat()**
Merging Multiple Streams	Convenient for merging a collection of streams	Requires manual pairing (two streams at a time)
Dynamic Concatenation	Handles dynamic or unknown number of streams	Less flexible for multiple streams
Code Simplicity	Easier to use for multiple streams	Best for combining exactly two streams

parallel streams

Java provides a `parallelStream` method in the Streams API to enable parallel processing of data. It splits the data source into multiple chunks, processes them in separate threads, and then combines the results. This is particularly useful for improving performance in scenarios where the data set is large and the computation is intensive.

Why Parallel Streams?

Parallel Streams were introduced to increase the performance of a program, but opting for parallel streams isn't always the best choice. There are certain instances in which we need the code to be executed in a certain order and in these cases, we better use sequential streams to perform our task at the cost of performance. The performance difference between the two kinds of streams is only of concern for large-scale programs or complex projects. For small-scale programs, it may not even be noticeable. Basically, you should consider using Parallel Streams when the sequential stream behaves poorly.



Key Differences Between Sequential and Parallel Streams

Feature	Sequential Stream	Parallel Stream
Execution	Single thread	Multiple threads
Performance	Suitable for small datasets	Suitable for large datasets
Order	Maintains order of processing	May not maintain order
Thread Management	No additional threads used	Uses threads from ForkJoinPool
Overhead	Minimal	Thread creation/management cost

Ex

```
        package parallelstream;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
class Employee {
        int id;
        String name;
        double salary;
```

```
    public Employee(int id, String name, double salary) {
        super();
        this.id = id;
        this.name = name;
        this.salary = salary;
    }
    public int getId() {
        return id;
    }
    public String getName() {
        return name;
    }
    public double getSalary() {
        return salary;
    }
}
public class ParallelStreamEx {
    public static void main(String[] args) {
        List<Employee> emp = new ArrayList<Employee>();
        emp.add(new Employee(101, "ram", 20000));
        emp.add(new Employee(102, "rakesh", 25000));
        emp.add(new Employee(103, "raju", 10000));
        emp.add(new Employee(104, "raja", 12000));
        emp.add(new Employee(105, "ravi", 30000));
        // using normal stream(sequential flow)
        System.out.println("using normal stream(sequential flow)");
        emp.stream().filter(e -> e.getSalary() > 15000).limit(2)
                .forEach(em -> System.out.println(em.name + " " + em.salary));
        System.out.println(" ");
        // using parallel stream(parallel flow(May not maintain order))
        System.out.println("using parallel stream(parallel flow):");
        emp.parallelStream().filter(e -> e.getSalary() > 15000).limit(2)
                .forEach(em -> System.out.println(em.name + " " + em.salary));
        System.out.println(" ");
        // converting stream to parallel stream(May not maintain order).
        System.out.println("converting stream to parallel stream:");
        emp.stream().parallel().filter(e -> e.getSalary() > 15000).limit(2)
                .forEach(em -> System.out.println(em.name + " " + em.salary));
    }
}
```

Output

```
using normal stream(sequential flow)
ram 20000.0
rakesh 25000.0
using parallel stream(parallel flow):
rakesh 25000.0
ram 20000.0
converting stream to parallel stream:
ram 20000.0
rakesh 25000.0
```