Stream is a special iterator class that allows us to process the collections of the objects in a fuctional manner.

We can get the stream object by

Stream stream = collectionObj.stream() ;

**Advantages :**

Conciseness : With Java Streams, you can perform operations like filter, map, and reduce without having to write for loops, which can make your code more readable and maintainable.

Efficiency / Lazy evaluation : Java Streams can improve the performance of your code by performing operations lazily and only on the elements of the stream that are actually used.

Parallelization: Java Streams can be easily parallelized, which means that operations can be performed on multiple elements of the stream simultaneously.This can help to improve the performance of your code on multi-core processors.

# map()

In Java 8, the map() method is an intermediate operation provided by the Stream interface. It is used to transform the elements of a stream by applying a function to each element.

The map() operation takes a Function as a parameter, which is called for each value in the input stream and produces one result value sent to the output stream.

**class** **Square**{

**public** **static** int getSquare(int num){

**return** num\*num ;

}

}

**class** **JavaMain** {

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

Io.initializeIO() ;

*// Io.output.println() ;*

List<Integer> list = Arrays.asList(3, 6, 9, 12, 15);

Function<Integer , Integer> sqFun = Square::getSquare ;

list.stream().map(sqFun).forEach(System.out::println) ;

Io.closeIO() ;

}

}

**class** **JavaMain** {

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

Io.initializeIO() ;

*// Io.output.println() ;*

List<Integer> list = Arrays.asList(3, 6, 9, 12, 15);

list.stream()

.map(number -> number \* 3)

.forEach(System.out::println);

Io.closeIO() ;

}

}

# forEach()

The forEach() method in Java 8 streams is a terminal operation that is used to iterate over all the elements of a stream and perform a specified operation on each element.

The operation is defined by a Consumer functional interface, which represents an operation that takes an input and returns no result.

# collect()

The collect() method in Java 8 streams is a terminal operation that is used to accumulate the elements of a stream into a mutable result container, such as a List, Set, or Map.

The Collector interface provides various built-in implementations, such as toList(), toSet(), toMap(), and toConcurrentMap(), which can be used to collect the stream elements into a List, Set, Map, or ConcurrentMap, respectively.

You can also create your own custom collectors by implementing the Collector interface.

List<Integer> list = Arrays.asList(3, 3, 6, 9, 12, 13 , 12);

list.stream()

.forEach(System.out::println);

System.out.println() ;

Set<Integer> set = list.stream().collect(Collectors.toSet()) ;

set.stream().forEach(System.out::println);

**class** **Student**{

**private** int id ;

**private** String name ;

**public** Student(int id , String name ){

**this**.id = id ;

**this**.name = name ;

}

**public** int getId() {

**return** id;

}

**public** String getName() {

**return** **this**.name;

}

}

Map<Integer, String> studentMap = studentList.stream()

.collect(Collectors.toMap(Student::getId, Student::getName));

**Question :** getId() , getName() are non- static methods of Student , is it legal to access using Student::getId in java 8 method references for above context ?

It is not legal to access non-static methods like getId() and getName() using Student::getId in Java 8 method references. Method references can only be used to refer to static methods, instance methods of a particular object, or instance methods of an arbitrary object of a particular class. Non-static methods like getId() and getName() are instance methods and require an object to be invoked.

In the given code, Student::getId and Student::getName are used as method references to the getId() and getName() methods of the Student class, respectively. However, they are used in the context of a stream pipeline, where they are invoked on each element of the stream. In this case, the method references are automatically bound to the objects being processed by the stream.

Therefore, it is legal to use Student::getId and Student::getName in the given code because they are being used in the context of a stream pipeline, where they are automatically bound to the objects being processed by the stream. However, it is not legal to use them as standalone method references outside of a stream pipeline.

Or simply we could have used the objects directly here like below

Map<Integer, String> studentMap = studentList.stream()

.collect(Collectors.toMap(student -> student.getId(), student -> student.getName()));

Output :

{1=Sesh, 2=Rao, 3=Suresh, 4=Mahesh}

# sorted()

The sorted() method in Java 8 streams is an intermediate used to sort the elements of a stream. It returns a new stream consisting of the elements of the original stream, but in a sorted order. The elements are sorted according to their natural order, or a provided Comparator can be used to define a custom order.

This method does not affect the ordering of the elements in the original stream.

List<Integer> list = Arrays.asList(3, 13, 6, 19, 12, 13 , 12);

list.stream().sorted().forEach(System.out::println);

custom sorting

List<Integer> list = Arrays.asList(3, 13, 6, 19, 12, 13 , 12);

list.stream().sorted((i1 , i2) -> i2.compareTo(i1) ).forEach(System.out::println);

# min()/max()

The min() method in Java 8 streams is used to find the minimum element in a stream according to the provided Comparator. It returns an Optional describing the minimum element of the stream, or an empty Optional if the stream is empty.

List<Integer> list = Arrays.asList(3, 13, 6, 19, 12, 13 , 12);

int num = list.stream().min((i1 , i2) -> i1.compareTo(i2) ).get() ;

System.out.println(num) ;

# toArray()

The toArray() method in Java 8 streams is used to convert a stream into an array. It returns an array containing the elements of the stream.

The toArray() method can be used with an argument that specifies the type of the resulting array. For example, stream.toArray(String[]::new) creates a String array, otherwise it always returns the Object[] array.

List<Integer> list = Arrays.asList(2, 4, 1, 3, 7, 5, 9, 6, 8);

Integer[] array = list.stream()

.toArray(Integer[]::new);

System.out.println(Arrays.toString(array));

# of()

It can take any group of values and convert them to stream so that , all stream operations can be applied.

The of() method can be used to create a stream of any type, including streams of objects, primitives, and even custom objects.

The of() method can be used with any number of arguments, including zero arguments.

Stream<String> stream = Stream.of("apple", "banana", "cherry");

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

# Parallel Stream

Parallel streams are a feature introduced in Java 8 that allow for the simultaneous execution of code on multiple cores of a processor. In a regular stream, the code is executed sequentially, while in a parallel stream, the code is divided into multiple streams that are executed in parallel on separate cores. The final result is the combination of the individual outcomes, although the order of execution is not under our control.

Use only when the tasks are independent.

# Lazy loading

Lazy loading is a feature of Java 8 Streams that allows intermediate operations to be evaluated only when a terminal operation is invoked.

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

Stream<String> stream = words.stream()

.filter(w -> {

System.out.println("Filtering " + w);

**return** w.startsWith("c");

})

.map(w -> {

System.out.println("Mapping " + w);

**return** w.toUpperCase();

});

*// Terminal operation*

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

O/p :

Filtering apple

Filtering banana

Filtering cherry

Mapping cherry

CHERRY

Filtering date

Filtering elderberry

# peek()

The peek() operation in Java 8 Streams is an intermediate operation that allows you to perform a side-effect on each element of the stream as they are encountered.

It does not modify the stream elements, but it can be useful for debugging or logging purposes.

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

numbers.stream().filter( n -> n%2==0 )

.peek(n -> System.out.println("Filtered : " + n ) )

.forEach(System.out::println) ;

O/p :

Filtered : 2

2

Filtered : 4

4

# reduce()

The reduce() operation in Java 8 Streams is a terminal operation that allows you to perform a reduction on the elements of the stream to produce a single result.

It takes a binary operator as a parameter that specifies how to combine the elements of the stream. The reduce() operation can be used to perform a variety of reduction operations, such as finding the sum, minimum, maximum, or average of the elements of the stream.

In Java 8, there are two versions of the reduce() method:

* Optional<T> reduce(BinaryOperator<T> accumulator): This version of reduce() does not require an identity element. It returns an Optional because the stream may be empty. If the stream is empty.

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

int sum1 = numbers.stream().reduce((a, b) -> a + b).get() ;

* T reduce(T identity, BinaryOperator<T> accumulator): This version of reduce() requires an identity element. It provides a default value to start the reduction operation. If the stream is empty, the identity element is returned.

int sum2 = numbers.stream().reduce(0 , (a, b) -> a + b) ;

# iterate()

iterate() operation is used to create an infinite stream by repeatedly applying a function to generate a sequence of values.

This is particularly useful for generating a sequence of values based on a starting element and a function that defines how to produce the next element in the sequence.

# limit()

This operation in Java 8 is used to limit the number of elements in a stream to a specified maximum size. It returns a new stream consisting of the elements of the original stream, truncated to be no longer than the specified maximum size.

# skip()

The skip() operation in Java 8 is used to skip the first n elements of a stream and return a new stream consisting of the remaining elements.

Stream<Integer> infEvenStream = Stream.iterate(2 , i-> i+2 ) ;

infEvenStream.skip(3).limit(3).forEach(System.out::println) ;

O/p :

8

10

12

# findFirst()

The findFirst() operation in Java 8 is used to find the first element in a stream that matches a given predicate. It returns an Optional object that may or may not contain a non-null value, depending on whether the stream is empty or not. If the stream is empty, then an empty Optional is returned.

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

Optional<Integer> firstEven = numbers.stream()

.filter(n -> n % 2 == 0)

.findFirst();

System.out.println(firstEven.get()) ;

# findAny()

The findAny() operation in Java 8 is used to find any element in a stream that matches a given predicate.

It is non-deterministic in nature, meaning that it is not guaranteed to return the same element if you call this method again.

# anyMatch()

operation in Java 8 is used to check if any element in a stream matches a given predicate. It returns a boolean value indicating whether any of the elements in the stream match the predicate or not. If the stream is empty, then false is returned.

List<Integer> numbers = Arrays.asList(1, 3, 5, 7, 8, 9, 11);

boolean anyEven = numbers.stream()

.anyMatch(n -> n % 2 == 0);

System.out.println(anyEven) ;

# allMatch()

The allMatch() operation in Java 8 is used to check if all elements in a stream match a given predicate. It returns a boolean value indicating whether all of the elements in the stream match the predicate or not. If the stream is empty, then true is returned.

It is important to note that the allMatch() method is a short-circuiting operation, meaning that it stops processing the stream as soon as it finds a non-matching element.

# noneMatch()

The noneMatch() operation in Java 8 is used to check if none of the elements in a stream match a given predicate. It returns a boolean value indicating whether none of the elements in the stream match the predicate or not. If the stream is empty, then true is returned.

# groupingBy()

The groupingBy() operation in Java 8 streams is used to group the elements of a stream based on a given criteria. It returns a Map object where the keys are the groups and the values are the elements that belong to each group.

The groupingBy() operation can be used in combination with other stream operations to perform various aggregate operations on the groups, such as counting, summing, averaging.

# flatMap()

The flatMap() operation in Java 8 streams is used to flatten a stream of collections to a stream of objects. It is an intermediate operation that returns a new stream.

During the flattening operation, the objects from all the collections in the original stream are combined into a single collection.

It performs flattening and mapping simultaneously.

List<List<Integer>> listOfLists = Arrays.asList(

Arrays.asList(1, 2, 3),

Arrays.asList(4, 5 , 1, 2, 4 ),

Arrays.asList(6, 7, 8 , 3 , 7)

);

Set<Integer> flattenedList = listOfLists.stream()

.flatMap(list -> list.stream() ).collect(Collectors.toSet()) ;

flattenedList.stream().forEach(System.out::println) ;

System.out.println() ;

O/p :

1

2

3

4

5

6

7

8

# Questions

## Find duplicate elements in a given list using streams ?

List<Integer> list = Arrays.asList(10 , 20 , 10 , 20 , 30 , 45 , 65 , 15 ) ;

List<Integer> newlist = **new** ArrayList<>() ;

list.stream().filter(num -> {

**if**(newlist.contains(num))

**return** **false** ;

**else**{

newlist.add(num) ;

**return** **true** ;

}

}).forEach(System.out::println) ;

System.out.println() ;

newlist.stream().forEach(System.out::println) ;

## Find the number of occurrence of each word in a given String ?

String input = "this moment this moment take this";

Map<String , Long> freqMap = Stream.of(input.split(" ")).

collect(Collectors.groupingBy(Function.identity() , Collectors.counting())) ;

freqMap.entrySet().stream().forEach(System.out::println) ;

Function.identity() will just return the same as it takes an i/p

O/p :

take=1

this=3

moment=2