Java 8 Stream Examples - 2022

**Sum of all Salaries of Employees**

List<Person> list = Arrays.*asList*(**new** Person(**"Ram"**, 23, 1000),  
 **new** Person(**"Hari"**, 63, 3000),  
 **new** Person(**"Shyam"**, 23, 7000));  
*// Sum of all salary of employees***int** totalSal = **list.stream().map(e -> e.getSal()).reduce((x, y) -> x + y).get();**  
System.***out***.println(**"Total Salary: "** + totalSal);  
totalSal = **list.stream().collect(Collectors.*summingInt*(e -> e.getSal()));**  
System.***out***.println(**"Total Salary: "** + totalSal);

**Who is drawing highest salary?**

**Comparator<Person> bySal = Comparator.*comparing*(p -> p.getSal());**  
Person maxSalPerson = **list.stream().max(bySal).get();**  
System.***out***.println(maxSalPerson);

**Comparator<Person> bySal = Comparator.comparing(p -> p.getSal());** OR

**Comparator<Person> bySal = (Person p1, Person p2) -> p1.getSal().compareTo(p2.getSal());**

**Sort a List of fruits in Ascending and Descending order**

List<String> list = Arrays.*asList*(**"Apple"**, **"Orange"**,**"Banana"**,**"Pineapple"**,**"Guava"**,**"Mango"**);  
List<String> flist = **list.stream().sorted(Comparator.*naturalOrder*()).collect(Collectors.*toList*());**  
System.***out***.println(flist);  
flist = **list.stream().sorted().collect(Collectors.*toList*());**  
System.***out***.println(flist);

*//Sorting in* ***Descending order***flist = **list.stream().sorted(Comparator.*reverseOrder*()).collect(Collectors.*toList*());**System.***out***.println(flist);

Also it can be written like this

**Comparator<String> cmp = Comparator.*comparing*(e->e,Comparator.*reverseOrder*());**  
fruits.stream().sorted(cmp).collect(Collectors.*toList*()).forEach(a -> System.*out*.println(a));

**Sort Custom Object in Ascending and Descending order**

*//Sort object by name*Comparator<Person> byName = **Comparator.*comparing*(e -> e.getName());**  
List<Person> pList = **list.stream().sorted(byName).collect(Collectors.*toList*());**  
System.***out***.println(pList);

You can also write like this for objects

Comparator<Person> byName = **(x, y) -> x.getName().compareTo(y.getName())**;

list = list.stream().sorted(byName).collect(Collectors.*toList*());

*//****Sort object by name in Descending order***Comparator<Person> byName = **Comparator.*comparing*(e -> e.getName(), Comparator.*reverseOrder*());**  
List<Person> pList = list.stream().sorted(byName).collect(Collectors.*toList*());  
System.***out***.println(pList);

Comparator<Person> byName = **Comparator.*comparing*(e -> e.getName());**  
List<Person> pList = **list.stream().sorted(byName.reversed()).collect(Collectors.*toList*());**  
System.***out***.println(pList);

**Multiple all the numbers into two in a list.**

List<Integer> list = Arrays.*asList*(1,2,3,4,5);  
List<Integer> someList = list.stream().map(e -> e\*2).collect(Collectors.*toList*());  
System.***out***.println(someList);

**Get total salary paid to each department**

List<Emp> elist = Arrays.*asList*(  
 **new** Emp(**"Ram"**, **"Engineering"**, 3000),  
 **new** Emp(**"Hari"**, **"Medical"**, 5000),  
 **new** Emp(**"Shyam"**, **"PWD"**, 3000),  
 **new** Emp(**"Radheshyam"**, **"Engineering"**, 6000),  
 **new** Emp(**"Sophi"**, **"Education"**, 3000),  
 **new** Emp(**"Rita"**, **"Education"**, 9000)  
);  
  
*// Get the total salary paid to each department*Map<String, Integer> map = elist.stream()  
 **.collect(Collectors.*groupingBy*(e -> e.getDept(), Collectors.*summingInt*(e-> e.getSal())));**  
System.***out***.println(map);

**Find the total count of each fruits in a list(Unique Fruit or Repeated no)**

List<String> list = Arrays.*asList*(**"Apple"**, **"Orange"**, **"Banana"**,  
 **"Pineapple"**, **"Guava"**, **"Apple"**, **"Mango"**, **"Banana"**);  
  
Map<String, Long> m = list.stream()  
 **.collect(Collectors.*groupingBy*(Function.*identity*(), Collectors.*counting*()));**  
System.***out***.println(m);

List<Person> list = List.*of*( **new** Person("John",23,1000, "Bangalore"),  
 **new** Person("Shyam",24,2000,"Chennai"),  
 **new** Person("Hari",25,3000,"Chennai"),  
 **new** Person("Gopi",26,2000,"Bangalore")  
 );  
  
**// Group person based upon their City name**  
Map<String, List<Person>> map = list.stream().collect(Collectors.*groupingBy*(e->e.getCity()));  
System.***out***.println(map);  
  
**// How many are from Bangalore or Chennai**  
Map<String, Long> map1 = list.stream().collect(Collectors.*groupingBy*(e->e.getCity(), Collectors.*counting*()));  
System.***out***.println(map1);

**Output**

**{Guava=1, Apple=2, Mango=1, Pineapple=1, Orange=1, Banana=2}**

**FindAny()/FindFirst()**

List<Integer> list = Arrays.*asList*(1,2,3,4,5);  
Integer val = **list.stream().filter( e -> (e%2 ==0) ).findAny().get();**  
System.***out***.println(val);

It is mostly used with filter.

List<Integer> list = Arrays.*asList*(1,2,3,4,5);  
Integer val = **list.stream().filter( e -> (e%2 ==0) ).findFirst().get();**  
System.***out***.println(val);

**In case of Object**

List<Emp> elist = Arrays.*asList*(  
 **new** Emp(**"Ram"**, **"Engineering"**, 3000),  
 **new** Emp(**"Hari"**, **"Medical"**, 5000),  
 **new** Emp(**"Shyam"**, **"PWD"**, 3000),  
 **new** Emp(**"Radheshyam"**, **"Engineering"**, 6000),  
 **new** Emp(**"Sophi"**, **"Education"**, 3000),  
 **new** Emp(**"Rita"**, **"Education"**, 9000)  
);  
  
Emp ee = elist.stream().filter( e -> e.getName().startsWith(**"R"**)).findAny().get();  
System.***out***.println(ee);

**AnyMatch()/AllMatch()/NoneMatch - Returns Boolean**

List<Emp> elist = Arrays.*asList*(  
 **new** Emp(**"Ram"**, **"Engineering"**, 3000),  
 **new** Emp(**"Hari"**, **"Medical"**, 5000),  
 **new** Emp(**"Shyam"**, **"PWD"**, 3000),  
 **new** Emp(**"Radheshyam"**, **"Engineering"**, 6000),  
 **new** Emp(**"Sophi"**, **"Education"**, 3000),  
 **new** Emp(**"Rita"**, **"Education"**, 9000)  
);  
  
*// AnyMatch, AllMatch, NoneMatch***boolean** flag = **elist.stream().anyMatch(e -> e.getSal() > 3000);**  
System.***out***.println(flag);  
flag = **elist.stream().allMatch(e -> e.getSal() > 1000);**  
System.***out***.println(flag);  
flag = **elist.stream().noneMatch(e -> e.getSal() > 10000);**  
System.***out***.println(flag);

**toArray()**

Employee[] employees = empList.stream().toArray(Employee[]::new);

**flatMap()**

A stream can hold complex data structures like Stream<List<String>>. In cases like this, flatMap() helps us to flatten the data structure to simplify further operations.

List<List<String>> namesNested = Arrays.*asList*(  
 Arrays.*asList*("Jeff", "Bezos"),  
 Arrays.*asList*("Bill", "Gates"),  
 Arrays.*asList*("Mark", "Zuckerberg"));  
namesNested.forEach(a-> System.*out*.println(a)); [Jeff, Bezos] [Bill, Gates] [Mark, Zuckerberg]  
System.*out*.println("========================");  
List<String> namesFlatStream = **namesNested.stream()  
 .flatMap(List::stream)  
 .collect(Collectors.*toList*());**  
namesFlatStream.forEach(a-> System.*out*.println(a)); Jeff Bezos Bill Gates Mark Zuckerberg

The above can be written like this.

List<List<String>> listOfList = List.*of*(  
 List.*of*("a","b"),List.*of*("c","d"),List.*of*("e","f")  
);  
**List<String> flatList = listOfList.stream().flatMap(e-> e.stream()).collect(Collectors.*toList*());**System.***out***.println(flatList);

**Peek()**

peek() is an intermediate operation.

We saw forEach() earlier in this section, which is a terminal operation. However, sometimes we need to perform multiple operations on each element of the stream before any terminal operation is applied.

Some operations are deemed short-circuiting operations. Short-circuiting operations allow computations on infinite streams to complete in finite time:

Stream<Integer> infiniteStream = Stream.iterate(2, i -> i \* 2);

List<Integer> collect = infiniteStream

.skip(3)

.limit(5)

.collect(Collectors.toList());

**Comparison Based Stream Operations**

**Sorted()**

**List<Employee> employees = empList.stream()**

**.sorted((e1, e2) -> e1.getName().compareTo(e2.getName()))**

**.collect(Collectors.toList());**

**min and max**

**Employee firstEmp = empList.stream()**

**.min((e1, e2) -> e1.getId() - e2.getId())**

**.orElseThrow(NoSuchElementException::new);**

**Lazy Evaluation**

One of the most important characteristics of Java streams is that they allow for significant optimizations through lazy evaluations.

All intermediate operations are lazy, so they’re not executed until a result of a processing is actually needed.

**Distinct()**

distinct() does not take any argument and returns the distinct elements in the stream, eliminating duplicates. It uses the equals() method of the elements to decide whether two elements are equal or not.

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

List<Integer> distinctIntList = **intList.stream().distinct().collect(Collectors.toList());**

**Java Stream Specializations**

**Creation**

The most common way of creating an IntStream is to call mapToInt() on an existing stream

Integer latestEmpId = empList.stream().mapToInt(Employee::getId).max()

.orElseThrow(NoSuchElementException::new);

**partitioningBy()**

We can partition a stream into two – based on whether the elements satisfy certain criteria or not.

List<Integer> intList = Arrays.asList(2, 4, 5, 6, 8);

**Map<Boolean, List<Integer>> isEven = intList.stream().collect(Collectors.partitioningBy(i -> i % 2 == 0));**

**Parallel Streams**

Using the support for parallel streams, we can perform stream operations in parallel without having to write any boilerplate code; we just have to designate the stream as parallel:

Employee[] arrayOfEmps = {

new Employee(1, "Jeff Bezos", 100000.0),

new Employee(2, "Bill Gates", 200000.0),

new Employee(3, "Mark Zuckerberg", 300000.0)

};

List<Employee> empList = Arrays.asList(arrayOfEmps);

empList.stream().parallel().forEach(e -> e.salaryIncrement(10.0));

**generate()**

We provide a Supplier to generate() which gets called whenever new stream elements need to be generated:

**Stream.generate(Math::random).limit(5).forEach(System.out::println);**

**Iterate()**

**iterate() takes two parameters: an initial value, called seed element and a function which generates next element** using the previous value. iterate(), by design, is stateful and hence may not be useful in parallel streams:

Stream<Integer> evenNumStream = **Stream.iterate(2, i -> i \* 2)**;

List<Integer> collect = evenNumStream.limit(5).collect(Collectors.toList());

In Java 9 we have the new version of iterate(), which adds a new parameter, which is a predicate used to decide when the loop should terminate. As long as the condition remains true, we keep going.

Stream.*iterate*(1, i -> i < 10, i -> i + 1)  
 .forEach(System.*out*::println);

**Java Streams Improvements In Java 9**

**takeWhile()**

The takeWhile method takes (elements from a stream) while a given condition is true.

**Stream.*iterate*(1, i -> i + 1).takeWhile(n -> n <= 10).map(x -> x \* x)  
 .forEach(System.*out*::println);**

**dropWhile()**

[filter](https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/stream/Stream.html#filter(java.util.function.Predicate)) - is a ***stateless*** intermediate operation which always returns a stream consisting only of elements the matches the given predicate.

[dropWhile](https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/stream/Stream.html#dropWhile(java.util.function.Predicate)) - is a ***statefull*** intermediate operation, which also expects a predicate and basically acts like a stateful filter. After the first element which doesn't match the predicate has been encountered, dropWhile() stops discarding elements from the stream.

**dropWhile only skips a continuous prefix of matches. filter drops any mismatched element**.

If this stream is **ordered** then the longest prefix is a contiguous sequence of elements of this stream that match the given predicate.

If this stream is **unordered**, and some (but not all) elements of this stream match the given predicate, then the behavior of this operation is **nondeterministic**; it is free to drop **any subset** of matching elements (which includes the empty set).

It works when there is a continuous similar elements.

Stream.*of*("aman", "amar", "suraj",  
 "suvam", "Zahafuj").dropWhile(name -> (name.charAt(0) == 'a'))  
 .forEach(x -> System.*out*.println(x)); *//suraj, suvam,Zahafuj* Stream.*of*("Ram","aman", "amar", "suraj",  
 "suvam", "Zahafuj").dropWhile(name -> (name.charAt(0) == 'a'))  
 .forEach(x -> System.*out*.println(x));*//Ram,aman, amar, suraj,suvam, Zahafuj*

**A stream does not store data and, in that sense, is not a data structure. It also never modifies the underlying data source.**

**For a List of Strings, the comparator can be written in the following manner for sorting.**

List<String> list = List.*of*("a","e", "b","k", "c", "d");  
  
// **Ascending order**  
**Comparator<String> cmp = (String a, String b) -> a.compareTo(b);  
Comparator<String> cmp1 = Comparator.*comparing*(e->e);  
Comparator<String> cmp2 = Comparator.*naturalOrder*();**  
// **Descending Order**  
**Comparator<String> cmp3 = (String a, String b) -> b.compareTo(a);  
Comparator<String> cmp4 = Comparator.*reverseOrder*();  
Comparator<String> cmp5 = Comparator.*comparing*(e->e,Comparator.*reverseOrder*());**

**You can use any one of the above for sorting**

List<String> sorted = list.stream().sorted(**cmp5**).collect(Collectors.*toList*());  
System.***out***.println(sorted);

**In case of Custom Object like this.**

**public class** Person **implements** Comparable {  
 **private** String name;  
 **private int** age;  
 **private** String city;  
 **private** Integer sal;

@Override  
 **public int** compareTo(Object o) {  
 **return** Integer.*compare*(**this**.sal, ((Person)o).getSal());  
 }

….

}

// Ascending order  
**Comparator<Person> bySal1 = (Person p1, Person p2) -> p1.getSal().compareTo(p2.getSal());  
Comparator<Person> bySal2 = Comparator.*naturalOrder*(); 🡸 if impelements Comparable  
Comparator<Person> bySal3 = Comparator.*comparing*(e->e.getSal());**  
  
// Descending Order  
**Comparator<Person> bySal4 = (Person p1, Person p2) -> p2.getSal().compareTo(p1.getSal());  
Comparator<Person> bySal5 = Comparator.*reverseOrder*(); 🡸 if impelements Comparable  
Comparator<Person> bySal6 = Comparator.*comparing*(e->e.getSal(), Comparator.*reverseOrder*());**  
  
List<Person> sortedList = plist.stream().sorted(bySal6).collect(Collectors.*toList*());  
System.***out***.println(sortedList);