Usage of Collectors in Java 8 – 2022

Collectors is used as an accumulator and provides utility methods

**Collectors.toList()**

List<Person> list = Arrays.*asList*( new Person("John", 23),  
 new Person("Vidya", 24), new Person("Roma", 34));  
List<String> pList = list.stream().map(**s -> s.getName()**).collect(Collectors.*toList*());  
System.*out*.println(pList);

You can also write like this  
pList = list.stream().map(**Person::getName**).collect(Collectors.*toList*());  
System.*out*.println(pList);

**Collectors.toSet()**

**Accumulate names into a TreeSet**

Set<String> set = list.stream().map(Person::getName).collect(**Collectors.*toCollection***(**TreeSet::new**));

list.stream().map(Person::getName).collect(**Collectors.*toSet()***);

**Collectors.joining()**

// Convert elements to strings and concatenate them, separated by semi colon

String joined = list.stream().map(Person::getName).collect(**Collectors.*joining***(";"));  
System.*out*.println(joined);

**Collectors.summingInt()**

int total = list.stream().collect(**Collectors.*summingInt***(Person::getSal));  
System.*out*.println(total);

// **Compute sum of salaries by department**

Map<Department, Integer> totalByDept

= **employees.stream().collect(Collectors.groupingBy(Employee::getDepartment,**

**Collectors.summingInt(Employee::getSalary)));**

**Collectors.groupingBy()**

// **Group employees by department**

**Map<String, List<Person>>** byDept = list.stream().collect(Collectors.*groupingBy*(Person::getDept));  
System.*out*.println(byDept);

Map<Integer, List<Student>> stdByAge = list.stream().collect(Collectors.groupingBy(Student::getAge));

List<String> items =  
 Arrays.*asList*(**"apple"**, **"apple"**, **"banana"**, **"apple"**, **"orange"**, **"banana"**, **"papaya"**);

**Map<String, Long>** result =  
 items.stream().collect(Collectors.*groupingBy*(**Function.*identity*()**, Collectors.*counting*()));

**groupingBy is used only iniside collect() method.**

In case of Map, the first argument will be the value coming from Collectors.groupingBy().

**Collectors.partitioningBy()**

partioningBy() always provides a **Map<Boolean, T>** type. It always divides the list into two parts based upon the condition.

// **Partition students into passing and failing**

Map<Boolean, List<Student>> passingFailing =

**students.stream().collect(Collectors.partitioningBy(s -> s.getGrade() >= 30));**

**All Even Odd Numbers**

List<Integer> numList = Arrays.*asList*(0, 1, 2, 3, 4, 5, 6, 7, 8, 9);  
Map<Boolean, List<Integer>> numMap = numList.stream().collect(Collectors.*partitioningBy*(num -> (num % 2) == 0));  
 List<Integer> evenList = numMap.get(**true**);  
 List<Integer> oddList = numMap.get(**false**);  
 System.***out***.println(**"All Evens : "** + evenList);  
 System.***out***.println(**"All Odds : "** + oddList);

**List of senior and junior developers**

List<Developer> team = Arrays.*asList*(**new** Developer(28, **"Sam"**), **new** Developer(23, **"John"**),  
 **new** Developer(35, **"Vidya"**), **new** Developer(50, **"Peter"**),**new** Developer(50, **"Sun"**));  
 Map<Boolean, List<Developer>> dataMap = team.stream().collect(**Collectors.*partitioningBy***(d -> d.getAge() > 30));  
 System.***out***.println(**"List of Senior developers : "** + dataMap.get(**true**));  
 System.***out***.println(**"List of Junior developers : "** + dataMap.get(**false**));

**Collectors.collectingAndThen()**

Stream<String> s = Stream.*of*(**"apple"**, **"banana"**, **"orange"**, **"kiwi"**);  
 List<String> readOnlyList =  
 **s.collect(Collectors.*collectingAndThen*(Collectors.*toList*(), Collections::*unmodifiableList*));**

**Collectors.maxBy()/minBy()**

**employee with the maximum salary**

String maxSalaryEmp = employeeList.stream().collect(

        Collectors.collectingAndThen(

            Collectors.maxBy(Comparator.comparing(Employee::getSalary)),

            (Optional<Employee> emp)-> emp.isPresent() ? emp.get().getName() : "none") );

Comparator<Person> byAge = Comparator.*comparing*(e -> e.getAge());  
 list.stream().collect(Collectors.*maxBy*(byAge))

.ifPresent( i -> System.*out*.println("Value: "+i));  
Person p = list.stream().collect(Collectors.*maxBy*(byAge)).get();

**Collectors.toMap()**

Map<String,String> map = Stream.of("AA","BB","CC").collect(Collectors.toMap(k->k, v->v+v));

map.forEach((k,v)->System.out.println("key:"+k +" value:"+v));

**List to Map with Key Mapper and Value Mapper**

List<String> list = Arrays.asList(“Mohan”,”Shyam”,”John”);

Map<String, Object> map = list.stream().collect(Collectors.toMap(Function.identity(), s->s));

With Object

List<Person> list = new ArrayList<>();

list.add(new Person(100, "Mohan"));

list.add(new Person(200, "Sohan"));

list.add(new Person(300, "Mahesh"));

Map<Integer, String> map = list.stream().collect(Collectors.toMap(Person::getId, Person::getName));

map.forEach((x, y) -> System.out.println("Key: " + x +", value: "+ y));

If keys will be duplicate then, it will throw IllegalStateException. To solve it, we pass merge function as BinaryOperator.

**List to Map with Key Mapper, Value Mapper and Merge Function**

The above can be written as

Map<Integer, String> map = list.stream().collect(Collectors.toMap(Person::getId, Person::getName, (x, y) -> x+", "+ y));

map.forEach((x, y) -> System.out.println("Key: " + x +", value: "+ y));

**List to Map with Key Mapper, Value Mapper, Merge Function and Map Supplier**

List<Person> list = new ArrayList<>();

list.add(new Person(100, "Mohan"));

list.add(new Person(100, "Sohan"));

list.add(new Person(300, "Mahesh"));

LinkedHashMap<Integer, String> map = list.stream().collect(Collectors.toMap(Person::getId, Person::getName,

(x, y) -> x+", "+ y, LinkedHashMap::new));

**Collectors.joining()**

List<String> list = Arrays.asList("Ram","Shyam","Shiv","Mahesh");

String result = list.stream().collect(Collectors.joining());

System.out.println(result);

result= list.stream().collect(Collectors.joining(","));

System.out.println(result);

result= list.stream().collect(Collectors.joining("-","[","]"));

System.out.println(result);

**Output**

RamShyamShivMahesh

Ram,Shyam,Shiv,Mahesh

[Ram-Shyam-Shiv-Mahesh]

**Collectors.joining() with List of Objects**

List<Person> list = Person.getList();

System.out.println("--Join person name--");

String result= list.stream().map(p -> p.getName()).collect(Collectors.joining());

System.out.println(result);

result= list.stream().map(p -> p.getName()).collect(Collectors.joining("|"));

System.out.println(result);

result= list.stream().map(p -> p.getName()).collect(Collectors.joining("-","[","]"));

System.out.println(result);

**Stream reduce()**

Returns a Collector which performs a reduction of its input elements under a specified **BinaryOperator** using the provided identity.

**Stream.reduce() with Accumulator**

int[] array = {23,43,56,97,32};

Arrays.stream(array).reduce((x,y) -> x+y).ifPresent(s -> System.out.println(s));

Arrays.stream(array).reduce(Integer::sum).ifPresent(s -> System.out.println(s));

Arrays.stream(array).reduce(StatisticsUtility::addIntData).ifPresent(s -> System.out.println(s));

public class StatisticsUtility {

public static int addIntData(int num1, int num2) {

return num1 + num2;

}

}

**Stream.reduce() with Identity and Accumulator**

int[] array = {1,2,3,4,5};  
*//Set start value. Result will be start value + sum of array.*int startValue = 100;  
int sum = Arrays.*stream*(array).reduce(startValue, (x,y) -> x+y);  
System.*out*.println(sum);  
sum = Arrays.*stream*(array).reduce(startValue, Integer::*sum*);

**Stream.reduce() with Identity, Accumulator and Combiner**

List<Integer> list2 = Arrays.asList(2, 3, 4);

//Here result will be 2\*2 + 2\*3 + 2\*4 that is 18.

int res = list2.parallelStream().reduce(2, (s1, s2) -> s1 \* s2, (p, q) -> p + q);

System.out.println(res);

List<String> list1 = Arrays.asList("Mohan", "Sohan", "Ramesh");

String result = list1.parallelStream().reduce("-", (s1, s2) -> s1 + s2, (p, q) -> p + q);

System.out.println(result);

}

**Output**

18

-Mohan-Sohan-Ramesh