Class Collectors in Java 8

**public final class Collectors extends** [**Object**](file:///E:\devsoftwares\docs\api\java\lang\Object.html)

Implementations of [Collector](file:///E:\devsoftwares\docs\api\java\util\stream\Collector.html) that implement various useful reduction operations, such as accumulating elements into collections, summarizing elements according to various criteria, etc.

The following are examples of using the predefined collectors to perform common mutable reduction tasks:

// Accumulate names into a List

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);

// 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()***);

// 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);

// **Compute sum of salaries of employee**

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

// **Group employees by department**

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

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

Map<Department, Integer> totalByDept

= **employees.stream()**

**.collect(Collectors.groupingBy(Employee::getDepartment,**

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

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

Map<Boolean, List<Student>> passingFailing =

**students.stream().collect(Collectors.partitioningBy(s -> s.getGrade() >= PASS\_THRESHOLD));**

# **collectingAndThen**

**public static <T,A,R,RR>**[**Collector**](file:///E:\devsoftwares\java8docs\api\java\util\stream\Collector.html)**<T,A,RR>**

**collectingAndThen(**[**Collector**](file:///E:\devsoftwares\java8docs\api\java\util\stream\Collector.html)**<T,A,R> downstream,**

[**Function**](file:///E:\devsoftwares\java8docs\api\java\util\function\Function.html)**<R,RR> finisher)**

*CollectingAndThen* is a special collector that allows performing another action on a result straight after collecting ends.

List<String> people

= people.stream().collect(collectingAndThen(toList(), Collections::unmodifiableList));

An example is given below.

**public class** Test1 {  
 **public static void** main(String[] args) {  
 Stream<String> s = Stream.*of*(**"apple"**, **"banana"**, **"orange"**, **"kiwi"**);  
 List<String> readOnlyList =  
 **s.collect(Collectors.*collectingAndThen*(Collectors.*toList*(), Collections::*unmodifiableList*));**  
 System.***out***.println(readOnlyList);  
 }  
}

**Problem Description:**Given a stream of employees, we want to –

1. Find the employee with the maximum salary for which we want to use the ***maxBy collector***.
2. The output of the *maxBy collector* being an **Optional** value, we want to check whether a value is present and then print the max salaried employee’s name.

public class CollectingAndThenExample {

  static List<Employee> employeeList

      = Arrays.asList(new Employee("Tom Jones", 45, 15000.00),

      new Employee("Tom Jones", 45, 7000.00),

      new Employee("Ethan Hardy", 65, 8000.00),

      new Employee("Nancy Smith", 22, 10000.00),

      new Employee("Deborah Sprightly", 29, 9000.00));

  public static void main(String[] args) {

    String maxSalaryEmp = employeeList.stream().collect(

        Collectors.collectingAndThen(

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

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

    System.out.println("Max salaried employee's name: "+ maxSalaryEmp);

  }

}

# **groupingBy**

public static <T,K> [Collector](file:///E:\devsoftwares\java8docs\api\java\util\stream\Collector.html)<T,?,[Map](file:///E:\devsoftwares\java8docs\api\java\util\Map.html)<K,[List](file:///E:\devsoftwares\java8docs\api\java\util\List.html)<T>>> groupingBy([Function](file:///E:\devsoftwares\java8docs\api\java\util\function\Function.html)<? super T,? extends K> classifier)

Returns a Collector implementing a "group by" operation on input elements of type T, grouping elements according to a classification function, and returning the results in a Map.

There are no guarantees on the type, mutability, serializability, or thread-safety of the Map or List objects returned.

An example is given below.

**public class** Test1 {  
 **public static void** main(String[] args) {  
 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*()));  
 System.***out***.println(result);  
 }  
}

**OUTPUT**

{papaya=1, orange=1, banana=2, apple=3}

The above example with sorting is given below.

**public class** Test1 {  
 **public static void** main(String[] args) {  
 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*()));  
 System.***out***.println(result);  
  
 *// Sort a map and add to finalMap* Map<String, Long> finalMap = **new** LinkedHashMap<>();  
 result  
 .entrySet()  
 .stream()  
 .sorted(Map.Entry.<String, Long>*comparingByValue*().reversed())  
 .forEachOrdered(e -> finalMap.put(e.getKey(), e.getValue()));  
  
 System.***out***.println(finalMap);  
 }  
}

OUTPUT

{papaya=1, orange=1, banana=2, apple=3}

{apple=3, banana=2, papaya=1, orange=1}

Another example is given below.

**Group by Student’s class name or age.**

public class **Student** {

private String name;

private int age;

private String className;

public Student(String name,String className,int age){

this.name=name;

this.age=age;

this.className = className;

}

}

public class **CollectorsGroupingBy** {

public static void main(String[] args) {

Student s1 = new Student("Ram", "A", 20);

Student s2 = new Student("Shyam", "B", 22);

Student s3 = new Student("Mohan", "A", 22);

Student s4 = new Student("Mahesh", "C", 20);

Student s5 = new Student("Krishna", "B", 21);

List<Student> list = Arrays.asList(s1,s2,s3,s4,s5);

//Group Student on the basis of ClassName

System.out.println("----Group Student on the basis of ClassName----");

Map<String, List<Student>> stdByClass = list.stream()

.collect(Collectors.groupingBy(Student::getClassName));

stdByClass.forEach((k,v)->System.out.println("Key:"+k+" "+

((List<Student>)v).stream().map(m->m.getName()).collect(Collectors.joining(","))));

//Group Student on the basis of age

System.out.println("----Group Student on the basis of age----");

Map<Integer, List<Student>> stdByAge = list.stream()

.collect(Collectors.groupingBy(Student::getAge));

stdByAge.forEach((k,v)->System.out.println("Key:"+k+" "+

((List<Student>)v).stream().map(m->m.getName()).collect(Collectors.joining(","))));

}

}

Find the output.

----Group Student on the basis of ClassName----

Key:A Ram,Mohan

Key:B Shyam,Krishna

Key:C Mahesh

----Group Student on the basis of age----

Key:20 Ram,Mahesh

Key:21 Krishna

Key:22 Shyam,Mohan

**Collectors.maxBy and Collectors.minBy**

Collectors.maxBy collects maximum element in stream for a given comparator. Collectors.minBy collects minimum element of a stream for a given comparator.

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();  
System.*out*.println(p);

### **Collectors.toMap**

Collectors.toMap collects the stream element as Map. We need to define function to generate key and value for stream input value.

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));

**Output**

key:CC value:CCCC

key:BB value:BBBB

key:AA value:AAAA

# **partitioningBy**

public static <T> [Collector](file:///E:\devsoftwares\java8docs\api\java\util\stream\Collector.html)<T,?,[Map](file:///E:\devsoftwares\java8docs\api\java\util\Map.html)<[Boolean](file:///E:\devsoftwares\java8docs\api\java\lang\Boolean.html),[List](file:///E:\devsoftwares\java8docs\api\java\util\List.html)<T>>> partitioningBy([Predicate](file:///E:\devsoftwares\java8docs\api\java\util\function\Predicate.html)<? super T> predicate)

Returns a Collector which partitions the input elements according to a Predicate, and organizes them into a Map<Boolean, List<T>>. There are no guarantees on the type, mutability, serializability, or thread-safety of the Map returned.

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

Example : A list contains a set of numbers from 1 to 9 , provide a list of odd number and even number using partioningBy.

**public class** Test1 {  
 **public static void** main(String[] args) {  
 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);  
 }  
}

**OUTPUT**

All Evens : [0, 2, 4, 6, 8]

All Odds : [1, 3, 5, 7, 9]

**Example:** A company holds a list of developers, find out the list of senior and junior developers.   
**class** Developer {  
 **private int age**;  
 **private** String **name**;  
  
 **public** Developer(**int** age, String name) {  
 **this**.**age** = age;  
 **this**.**name** = name;  
 }  
 *// get / set methods* @Override  
 **public** String toString() {  
 **return "Developer{"** + **"name='"** + **name** + **'\''** + **'}'**;  
 }  
}  
  
**public class** Test1 {  
 **public static void** main(String[] args) {  
 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**));  
 }  
}

OUTPUT

List of Senior developers : [Developer{name='Vidya'}, Developer{name='Peter'}, Developer{name='Sub'}]

List of Junior developers : [Developer{name='Sam'}, Developer{name='John'}]

# **Collectors: joining() Example**

Suppose we the elements A,B,C,D that will be collected in following ways.   
1. **joining()**

It returns the Collector that is used to concatenate the elements with no delimiter. For our given input elements, the output string will be ABCD.   
2. **joining(CharSequence delimiter)**

It returns the Collector that is used to concatenate the elements with given delimiter. For our given input elements and separator (-), the output string will be A-B-C-D.   
3. **joining(CharSequence delimiter, CharSequence prefix, CharSequence suffix)**

It returns the Collector that is used to concatenate the elements with given delimiter, prefix and suffix . For our given input elements, separator(-), prefix([) and suffix(])the output string will be [A-B-C-D].

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

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);

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

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

System.out.println(result);

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

System.out.println(result);

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

System.out.println(result);

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

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

System.out.println(result);

result= list.stream().map(p -> p.getName()+"-" + p.getAge()).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.

On this page we will provide java 8 Stream reduce() example. It is used to get the sum of numbers stored in List, Array etc. It can also concatenate the string stored in List and Array etc. Java 8 lambda Stream.reduce() method can perform many more reducing task as required. We are providing here some usability of Stream.reduce(). Find the examples.

##### Contents

* [Stream.reduce() with Accumulator](https://www.concretepage.com/java/jdk-8/java-8-stream-reduce-example#accumulator)
* [Stream.reduce() with Identity and Accumulator](https://www.concretepage.com/java/jdk-8/java-8-stream-reduce-example#identity-accumulator)
* [Stream.reduce() with Identity, Accumulator and Combiner](https://www.concretepage.com/java/jdk-8/java-8-stream-reduce-example#identity-accumulator-combiner)
* [Reduce List and Array into a String](https://www.concretepage.com/java/jdk-8/java-8-stream-reduce-example#list-array-string)
* [Reduce List and Array into Sum](https://www.concretepage.com/java/jdk-8/java-8-stream-reduce-example#list-array-sum)

**Stream.reduce() with Accumulator**

Here we will pass BinaryOperator as accumulator. In case of numeric BinaryOperator, the start value will be 0. In case of string, the start value will be a blank string.   
reduce(BinaryOperator accumulator)  
The method will return Optional instance. Find the example.

public static void main(String[] args) {

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**

Here will use an identity and accumulator. We will pass the identity as start value.   
reduce(T identity, BinaryOperator<T> 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*);  
System.*out*.println(sum);

**Output**

115

115

115  
  
**Stream.reduce() with Identity, Accumulator and Combiner**

Here we will pass three arguments identity, accumulator and combiner in reduce() method. This method with these three arguments is used in parallel processing. Combiner works with parallel stream only, otherwise there is nothing to combine.   
reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)  
public static void main(String[] args) {

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

**Reduce List and Array into a String**

public static void main(String[] args) {

//Reduce Array to String.

String[] array = {"Mohan", "Sohan", "Mahesh"};

Arrays.stream(array).reduce((x, y) -> x +"," + y) .ifPresent(s -> System.out.println("Array to String: "+ s));

//Reduce List to String.

List<String> list = Arrays.asList("Mohan", "Sohan", "Mahesh");

list.stream().reduce((x, y) -> x +"," + y).ifPresent(s -> System.out.println("List to String: "+ s));

}

**Output**

Array to String: Mohan,Sohan,Mahesh

List to String: Mohan,Sohan,Mahesh

**Reduce List and Array into Sum**

public static void main(String[] args) {

//Reduce Array to sum.

int[] array = {30, 10, 20, 40};

int sum = Arrays.stream(array).reduce(0, (x, y) -> x + y);

System.out.println("Sum of Array: "+ sum);

//Reduce List to sum.

List<Integer> list = Arrays.asList(30, 10, 20, 40);

sum = list.stream().reduce(0, (x, y) -> x + y);

System.out.println("Sum of List: "+ sum);

}

Output

Sum of Array: 100

Sum of List: 100

# **Collectors.toMap()**

On this page we will provide java 8 convert List to Map using Collectors.toMap() example. Using lambda expression, we can convert List to Map in a single line. Java 8 provides Collectors.toMap() that is useful to convert List to Map. We need to pass mapping function for key and value. To avoid conflict of duplicate keys, we pass merge function otherwise it will throw IllegalStateException. By default Collectors.toMap() returns HashMap and if we want to change it we need to pass required supplier instance. Now find the toMap() method syntax.   
  
toMap(Function keyMapper, Function valueMapper, BinaryOperator mergeFunction, Supplier mapSupplier)  
  
The arguments are as follows.   
  
Function keyMapper: It generates key for Map.   
Function valueMapper: It generates value for Map.   
BinaryOperator mergeFunction: This is optional. The usability of merge function is to handle the situation of duplicate Map keys. Using BinaryOperator we can merge the values of duplicate keys. If we do not pass this argument, then by default it throws IllegalStateException in case of duplicate keys.   
  
Supplier mapSupplier: This is optional. It returns a Map in which data is filled as key/value. If we do not pass map supplier then the default supplier will return HashMap. If we want to get any other instance such as LinkedHashMap, we need to pass supplier as LinkedHashMap::new.

##### Contents

* [List to Map with Key Mapper and Value Mapper](https://www.concretepage.com/java/jdk-8/java-8-convert-list-to-map-using-collectors-tomap-example#key-value)
* [List to Map with Key Mapper, Value Mapper and Merge Function](https://www.concretepage.com/java/jdk-8/java-8-convert-list-to-map-using-collectors-tomap-example#merge)
* [List to Map with Key Mapper, Value Mapper, Merge Function and Map Supplier](https://www.concretepage.com/java/jdk-8/java-8-convert-list-to-map-using-collectors-tomap-example#supplier)

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

Here we will pass mapping function of key mapper and value mapper. The syntax of method is as follows.toMap(Function keyMapper, Function valueMapper) Now find a simple example.

public static void main(String[] args) {

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

list.add("Mohan");

list.add("Sohan");

list.add("Mahesh");

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

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

}

**Output**

Key: Mohan, value: Mohan

Key: Mahesh, value: Mahesh

Key: Sohan, value: Sohan

Now we have a List of user class Person. Find the code to convert the list into map.

public static void main(String[] args) {

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));

}

Here 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**

In this example we will pass BinaryOperator as merge function. When the toMap() method finds duplicate keys then the values are merged and it does not throw exception. Find the method syntax.   
toMap(Function keyMapper, Function valueMapper, BinaryOperator mergeFunction)

public static void main(String[] args) {

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

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

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

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

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));

}

**Output**

Key: 100, value: Mohan, Sohan

Key: 300, value: Mahesh

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

Here we will pass map supplier in the toMap() method. If we want to return LinkedHashMap, we need to pass supplier as LinkedHashMap::new.   
toMap(Function keyMapper, Function valueMapper, BinaryOperator mergeFunction, Supplier mapSupplier)  
Find the example.   
public static void main(String[] args) {

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));

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

}

Output

Key: 100, value: Mohan, Sohan

Key: 300, value: Mahesh