```java
package com.ravi.to_map;

import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

//Print the length of the City Collectors.toMap()

public class ToMapDemo2
{
    public static void main(String[] args)
    {
        List<String> listOfCountry =
List.of("India","Australia","USA","China","Japan");

        Map<String, Integer> map = listOfCountry.stream()
             .collect(Collectors.toMap(String::toUpperCase, String::length));

        map.forEach((k,v) -> System.out.println("The length of :"+k+" is :"+v));

    }

}
```

```java
package com.ravi.to_map;

import java.util.ArrayList;
import java.util.Map;
import java.util.stream.Collectors;

//Don't allow duplicate key using Collectors.toMap()

record Product(Integer id, String name)
{

}

public class ToMapDemo3
{
    public static void main(String[] args)
    {
       Product p1 = new Product(111, "Camera");
       Product p2 = new Product(222, "Laptop");
       Product p3 = new Product(111, "Mobile");  //Duplicate key


       ArrayList<Product> listOfProduct = new ArrayList<>();
       listOfProduct.add(p1);
       listOfProduct.add(p2);
       listOfProduct.add(p3);

       Map<Integer, String> collect = listOfProduct.stream()
         .collect(Collectors.toMap(Product::id, Product::name));

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


    }

}
```

Note : If we will take duplicate key then we will get an exception i.e java.lang.IllegalStateException

In order to remove the duplicate key as per our requirement we can take 3rd parameter i.e BinaryOperator<T,T> merger

Collectors.toMap(Function<T,R> keyMapper, Function<T,R> valueMapper, BinaryOperator<T,T> merger)

```java
package com.ravi.to_map;

import java.util.ArrayList;
import java.util.Map;
import java.util.stream.Collectors;

//Don't allow duplicate key using Collectors.toMap()

record Product(Integer id, String name)
{

}

public class ToMapDemo3
{
    public static void main(String[] args)
    {
       Product p1 = new Product(111, "Camera");
       Product p2 = new Product(222, "Laptop");
       Product p3 = new Product(111, "Mobile");


       ArrayList<Product> listOfProduct = new ArrayList<>();
       listOfProduct.add(p1);
       listOfProduct.add(p2);
       listOfProduct.add(p3);

       Map<Integer, String> collect = listOfProduct.stream()
         .collect(Collectors.toMap(Product::id, Product::name,(oldKey, newKey)->newKey ));

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


    }

}
```

IQ:

By default toMap() method return type is HashMap (HashMap::new) so output is un-predictable, If we want Predictable output then we need to take 4th parameter as Supplier<T>

Collectors.toMap(Function<T,R> keyMapper, Function<T,R> valueMapper, BinaryOperator<T,T> merger, Supplier<T>)

```java
package com.ravi.to_map;

import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.stream.Collectors;

// IQ : Print the Data in the same order as they inserted in the Map

record Trainer(Integer id, String name, Double salary)
{

}

public class ToMapDemo4
{
    public static void main(String[] args)
    {
        ArrayList<Trainer> listOfTrainers = new ArrayList<>();
         listOfTrainers.add(new Trainer(111, "Scott", 350000D));
         listOfTrainers.add(new Trainer(222, "Smith", 322000D));
         listOfTrainers.add(new Trainer(333, "Alen", 367000D));
         listOfTrainers.add(new Trainer(444, "John", 349000D));
         listOfTrainers.add(new Trainer(111, "Ravi", 350000D));


         Map<Integer, String> collect = listOfTrainers.stream()
            .collect(Collectors.toMap(Trainer::id, Trainer::name,(oldKey, newKey)-> newKey,
LinkedHashMap::new));


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


    }

}
```

public static Collector<CharSequence, ?, String> joining(CharSequence delimiter)
----
* It will join the given CharSequenec to all the elements.

```java
package com.ravi.collect;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

//Collectors.joining(CharSequence delimiter)
public class CollectDemo1
{
     public static void main(String[] args)
     {
         List<String> list = Arrays.asList("a", "b", "c", "d");
         String result = list.stream().collect(Collectors.joining("@"));
         System.out.println(result);
     }
}
```

public static <T, K> Collector<T, ?, Map<K, List<T>>> groupingBy(Function<? super T, ? extends K> classifier)

Searching/Grouping criteria will become the key of the Map

It is a predefined static method of Collectors class.

It is used to convert the stream into Map. Here Map keys are the function which are passing as a parameter to groupingBy() method and the value of Map is list of items.

```java
package com.ravi.collect;

import java.util.Arrays;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

//Group the city name according to length of the city name

public class CollectDemo2
{
    public static void main(String[] args)
    {
        List<String> cities = Arrays.asList("Delhi", "Indore", "Kolkata", "Pune",
"Hyderabad","Mumbai","Chennai","Ampt");

        Map<Integer, List<String>> collect = cities.stream()
            .collect(Collectors.groupingBy(String::length));


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

    }

}
```

```java
package com.ravi.collect;

import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;
import java.util.stream.Stream;

//Group the student based on the course

record Student(String course, Integer id, String name)
{

}


public class CollectDemo3
{
    public static void main(String []args)
    {
        Student s1 = new Student("Java",101,"Scott");
        Student s2 = new Student("Java",102,"Smith");
        Student s3 = new Student("Java",103,"Samrat");
        Student s4 = new Student("HTML",104,"Raj");
        Student s5 = new Student("HTML",105,"Rahul");
        Student s6 = new Student("REACT",106,"Alen");
        Student s7 = new Student("REACT",107,"John");


        Map<String, List<Student>> collect = Stream.of(s1,s2,s3,s4,s5,s6,s7)
         .collect(Collectors.groupingBy(Student::course));

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

    }

}
```

```java
package com.ravi.collect;

import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;
import java.util.stream.Stream;

//Group the employee based on the department

record Department(Integer deptId, String deptName)
{
}



record Employee(Integer empId, String empName, double salary, Department dept)
{
    //111 , "A", 23890.89, new Department(1,"IT");
}



public class CollectDemo4
{
    public static void main(String[] args)
    {
        Employee e1 = new Employee(111, "Raj", 23789.89, new Department(1, "IT"));
        Employee e2 = new Employee(222, "Rahul", 23789.89, new Department(1, "IT"));

        Employee e3 = new Employee(333, "Scott", 23789.89, new Department(2, "Sales"));
        Employee e4 = new Employee(444, "Smith", 23789.89, new Department(2, "Sales"));

        Employee e5 = new Employee(333, "Virat", 23789.89, new Department(3, "HR"));
        Employee e6 = new Employee(444, "Rohit", 23789.89, new Department(3, "HR"));


        Stream<Employee> streamOfEmp = Stream.of(e1,e2,e3,e4,e5,e6);

        Map<Department, List<Employee>> deptWiseEmp =
streamOfEmp.collect(Collectors.groupingBy(Employee::dept));

        deptWiseEmp.forEach((dep, emps)-> System.out.println(dep+" : "+emps));

    }

}
```

public static <T> Collector<T, ?, Map<Boolean, List<T>>> partitioningBy(Predicate<? super T> predicate)

It is a predefined static method of Collectors class.

It is used to partition the elements of a stream into two groups based on a given predicate.

The result is a Map with Boolean keys, where the true key corresponds to elements that satisfy the predicate, and the false key corresponds to elements that do not satisfy the predicate.

```java
package com.ravi.collect;

import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

//Partition the element based on the condition given as Predicate

public class CollectDemo5
{
    public static void main(String[] args)
    {
        List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

       Map<Boolean, List<Integer>> collect = numbers.stream()
          .collect(Collectors.partitioningBy(n -> n%2==1));

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


     }
}
```

Optional<T> reduce(BinaryOperator<T> accumulator)
T reduce(T identity, BinaryOperator<T> accumulator)

It is a predefined method of Stream interface.

This method is useful for combining stream elements into a single result because it accept BinaryOperator<T> as a parameter, such as computing the sum, product, or concatenation of elements.

It performs a reduction on the elements of the stream, using an associative accumulation function, and returns an Optional.

```java
package com.ravi.reduce;

import java.util.Optional;
import java.util.stream.Stream;

//add the number using reduce

public class ReduceDemo1
{
    public static void main(String[] args)
    {

       Stream<Integer> integerStream = Stream.of(1, 2, 3, 4, 5,6);
       Optional<Integer> optional = integerStream.reduce(Integer::sum);
       optional.ifPresent(System.out::println);



        System.out.println("========================================");

        integerStream = Stream.of(1, 2, 3, 4, 5);
        Integer reduce = integerStream.reduce(100, Integer::sum);
        System.out.println(reduce);
    }
}
```

```java
package com.ravi.reduce;

import java.util.Arrays;
import java.util.List;
import java.util.Optional;

//find the maximum number using reduce

public class ReduceDemo2
{
     public static void main(String[] args) {
            List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

            Optional<Integer> max = numbers.stream()
                                      .reduce(Integer::max);

            max.ifPresent(System.out::println);
        }
}
```

```java
package com.ravi.reduce;

import java.util.Arrays;
import java.util.List;
import java.util.Optional;

//Concatenate all the String into Single String

public class ReduceDemo3
{
    public static void main(String[] args) {
        List<String> words = Arrays.asList("Java", "is", "Best", "language");

        Optional<String> concatenated = words.stream()
                                      .reduce((a, b) -> a + " " + b);

        concatenated.ifPresent(System.out::println);
      }
}
```

```java
package com.ravi.reduce;

import java.util.stream.Stream;

//Find the total sale of a shop in a particular day

record Sale(String item, Double amount)
{

}

public class ReduceDemo4
{
    public static void main(String[] args)
    {
        Stream<Sale> sales = Stream.of(
                new Sale("Camera", 10000.0),
                new Sale("Mobile", 50000.0),
                new Sale("Laptop", 80000.0),
                new Sale("LED", 20000.0),
                new Sale("Watch", 5000.0)
            );

        Double totalSale = sales.reduce(0.0, (sum , sale)-> sum + sale.amount(),Double::sum);

        System.out.println("Total Sale for today is :"+totalSale);



    }
}
```