29-01-2025

Stream API in Java :

It is introduced from Java 8 onwards, the Stream API is used to process the collection/Array objects.

It contains classes for processing sequence of elements over Collection object and array.

Stream is a predefined interface available in java.util.stream sub package.

Package Information :
----
java.util -> Base package
java.util.function -> Functional interfaces
java.util.concurrent -> Multithreaded support
java.util.stream -> Processing of Collection/array Object

Interfaces which contains forEach() method in java :

The Java forEach() method is a technique to iterate over a collection such as (list, set or map) and stream. It is used to perform a given action on each of the element of the collection.

The forEach() method has been added in following places:

Iterable interface – This makes Iterable.forEach() method available to all collection classes. Iterable interface is the super interface of Collection interface

Map interface – This makes forEach() operation available to all map classes.

Stream interface – This makes forEach() operations available to all types of stream.

Creation of Streams to process the data :

We can create Stream from collection or array with the help of stream() and Stream.of(T ...values) methods:

A stream() method is added to the Collection interface and allows creating a Stream<T> using any collection object as a source

public java.util.stream.Stream<E> stream();

The return type of this method is Stream interafce available in java.util.stream sub package.

Eg:-

```
List<String> items = new ArrayList<String>();
                    items.add("Apple");
                    items.add("Orange");
                    items.add("Mango");
                    //Collection to stream
                    Stream<String> stream = items.stream();
-----------------------------------------------------------------
package com.ravi.basic;
import java.util.*;  //Base package
import java.util.stream.*; //Sub package
public class StreamDemo1
{
          public static void main(String[] args)
          {
                    List<String> items = new ArrayList<>();
                    items.add("Apple");
                    items.add("Orange");
                    items.add("Mango");

      //Collections Object to Stream
                    Stream<String> strm = items.stream();
                    strm.forEach(p -> System.out.println(p));
          }
}
------------------------------------------------------------------
public static java.util.stream.Stream  of(T ...values)
------------------------------------------------------
```

It is a static method of Stream interface through which we can create Stream of arrays and Stream of Collection. The return type of this method is Stream interface.

```
//Stream.of()
package com.ravi.basic;
import java.util.stream.*;
public class StreamDemo2
{
          public static void main(String[] args)
          {
                    //Stream of numbers
                    Stream<Integer> stream = Stream.of(1,2,3,4,5,6,7,8,9);
                    stream.forEach(p -> System.out.println(p));
                    System.out.println("..............................");

        //Anonymous Array Object (Stream of Arrays)

                    Stream<Integer> strm = Stream.of( new Integer[]{15,29,45,8,16} );
                    strm.forEach(p -> System.out.println(p));
          }
}
```

30-01-2025

Operation in Stream API :

In Stream API we have two types of operation

1) Intermediate Operation
2) Terminal Operation

Intermediate Operation :

Intermediate Operation will always produce another Stream, Here Streams are not in a closed position that means further we can apply any intermediate operation method.

In intermediate operation the method return type will always Stream because it is producing another Stream.

The following methods are available to perform intermediate operation.

filter(Predicate<T> predicate): Returns a new stream which contains filtered elements based on the boolean expression using Predicate.

map(Function<T, R> mapper): Transforms elements in the stream using the provided mapping function.

flatMap(Function<T, Stream<R>> mapper): Flattens a stream of streams into a single stream.

distinct(): Returns a stream with distinct elements (based on their equals method).

sorted(): Returns a stream with elements sorted in their natural order.

sorted(Comparator<T> comparator): Returns a stream with elements sorted using the specified comparator.

peek(Consumer<T> action): Allows us to perform an action on each element in the stream without modifying the stream.

limit(long maxSize): Limits the number of elements in the stream to a specified maximum size.

skip(long n): Skips the first n elements in the stream.

takeWhile(Predicate<T> predicate): Returns a stream of elements from the beginning until the first element that does not satisfy the predicate.

dropWhile(Predicate<T> predicate): Returns a stream of elements after skipping elements at the beginning that satisfy the predicate.

Note : All these methods return type is Stream.

The following progran explains that once a Stream is closed OR consumer by using terminal method then we can't re-use that
Stream, If we try to re-use then at runtime java.lang.IllegalStateException will be generated as shown in the program.

```
package com.ravi.testing;

import java.util.stream.Stream;

public class StreamOperation {

	public static void main(String[] args)
	{
		System.out.println("Main");
	  Stream<Integer> of = Stream.of(1,2,3,4,5,6,7,8,9,10);
	  of.filter(num -> num % 2 ==0).forEach(System.out::println); //Stream is closed OR
Consumed (final Operation)

	  of.forEach(System.out::println); //java.lang.IllegalStateException



	}

}
```

```
-------------------------------------------------------------
public abstract Stream<T> filter(Predicate<T> p) :
--------------------------------------------------
```

It is a predfined method of Stream interface. It is used to select/filter elements as per the Predicate passed as an argument. It is basically used to filter the elements based on boolean condition.

```
public abstract <T>  collect(java.util.stream.Collector c)
--------------------------------------------------------------
```

It is a predfined method of Stream interface. It is used to return the result of the intermediate operations performed on the stream.

It is a terminal operation. It is used to collect the data after filteration and convert the data to the Collection(List/Set/Map).

Collectors is a predfined final utility class available in java.util.stream sub package which conatins static method toList(),toSet(), toMap() to convert the data as a List/Set/Map i.e Collection object. The return type of these method is List/Set/Map interface.

```
------------------------------------------------------------
//Filter all the even numbers from Collection
package com.ravi.basic;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class StreamDemo3
{
        public static void main(String[] args)
        {
          List<Integer> listOfNumber = Arrays.asList(1,2,3,4,5,6,7,8,9,10,11,12);

          //Without Stream API
          List<Integer> even = new ArrayList<Integer>();

          for(Integer num : listOfNumber)
          {
                if(num % 2==0)
                {
                        even.add(num);
                }
          }
```

```
            even.forEach(System.out::println);

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

            //With Stream API
            listOfNumber.stream().filter(num -> num%2==0).forEach(System.out::println);

        }
}
------------------------------------------------------------
package com.ravi.basic;

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

public class FilterDemo {

        public static void main(String[] args)
        {
                List<String> listOfName =
List.of("Aryan","Ankit","Raj","Rohit","Aniket","Raj","Aryan","Ajinkya","Ankit");

                //Retrieve all the names which starts from character A and it should not
                //contain duplicate

                Set<String> filteredName = listOfName.stream().filter(str ->
str.startsWith("A")).collect(Collectors.toSet());

                System.out.println(filteredName);
        }

}
-----------------------------------------------------------

package com.ravi.basic;

import java.util.List;
import java.util.Set;
import java.util.stream.Collectors;
import java.util.stream.Stream;
```

```
public class FilterDemo1 {

        public static void main(String[] args)
        {
                //Retrieve all the names which starts from R and duplicates are allowed
      List<String> filteredName =
Stream.of("Aryan","Ankit","Raj","Rohit","Aniket","Raj","Aryan").filter(str->
str.startsWith("R")).collect(Collectors.toList());

     System.out.println(filteredName);



        }

}
```

//Filtering the name which starts with 'R' character with Stream API where duplicate are not allowed and in Alphabetical order

```
package com.ravi.basic;

import java.util.Arrays;
import java.util.Comparator;
import java.util.List;
import java.util.stream.Collector;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class StreamDemo4
{
        public static void main(String[] args)
        {
                List<String> listOfName =
Arrays.asList("Raj","Rahul","Ankit","Roshan","Raj","Scott","Rohit","Ratan","Ravi");

                listOfName.stream().filter(str ->
str.startsWith("R")).distinct().sorted().forEach(System.out::println);


        }
}
```

//Sorting the data

```
package com.ravi.basic;
import java.util.*;
import java.util.stream.*;
public class StreamDemo5
{
	public static void main(String[] args)
	{
	List<String> names = Arrays.asList("Zaheer","Rahul","Aryan","Sailesh","Zaheer");

		List<String> collect =
names.stream().distinct().sorted().collect(Collectors.toList());

		System.out.println(collect);
	}
}
------------------------------------------------------------
package com.ravi.basic;

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

//Fetch all the Employees name whose salary is greater than 50k

record Employee(Integer empId, String empName, Double empSalary)
{

}

public class StreamDemo6
{
	public static void main(String[] args)
	{
	  Employee e1 = new Employee(111, "Juber", 90000D);
	  Employee e2 = new Employee(222, "Aryan", 40000D);
	  Employee e3 = new Employee(333, "Scott", 60000D);
	  Employee e4 = new Employee(444, "Rahul", 70000D);
	  Employee e5 = new Employee(555, "Aakash",85000D);
	  Employee e6 = new Employee(666, "Manav", 92000D);

	  List<Employee> list = Stream.of(e1,e2,e3,e4,e5,e6).filter(emp ->
emp.empSalary()>50000).collect(Collectors.toList());
```

```
        list.forEach(System.out::println);

      }
}
```

31-01-2025
----

public Stream map(Function<? super T,? extends R> mapper) :

It is a predefined method of Stream interface.

It takes Function (Predefined functional interafce) as a parameter.

It performs intermediate operation and consumes single element from input Stream and produces single element to output Stream. (1:1 transformation)

Here mapper function is functional interface which takes one input and provides one output.

```
package com.ravi.basic;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class StreamDemo7
{
   public static void main(String[] args)
   {
      List<Integer> listOfNumbers = Arrays.asList(1,2,3,4,5,6,7,8,9,10);
      //add a constant value 10 to all the numbers

      List<Integer> numbers = listOfNumbers.stream().map(num ->
num+10).collect(Collectors.toList());

      System.out.println(numbers);

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

     List<Integer> immutableList = List.of(1,2,3,4,5,6,7,8,9,10,2,3,4,6,8);
```

```
    //Fetch all the unique even numbers and find the cube of those numbers
    System.out.println("Cube of all the even numbers :");
     immutableList.stream().distinct().filter(num -> num%2==0).map(n ->
n*n*n).forEach(System.out::println);


  }
}

-----------------------------------------------------------
package com.ravi.basic;

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

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

}

public class MapDemo1
{
	public static void main(String[] args)
	{
		ArrayList<MyEmp> listOfEmp = new ArrayList<>();
		listOfEmp.add(new MyEmp(1, "Scott", 800D));
		listOfEmp.add(new MyEmp(2, "Smith", 1200D));
		listOfEmp.add(new MyEmp(3, "Alen", 1500D));
		listOfEmp.add(new MyEmp(4, "Martin", 1800D));
		listOfEmp.add(new MyEmp(5, "John", 2000D));

		System.out.println("Original Employee Data with Old Salary");
		listOfEmp.forEach(System.out::println);

		//add 500D in the salary for all the Employees
		List<Double> collect = listOfEmp.stream().map(emp ->
emp.salary()+500).collect(Collectors.toList());

		System.out.println("Employee Data after Salary updation");
		collect.forEach(System.out::println);
```

```
          }

}
------------------------------------------------------------
//Program on map(Function<T,R> mapped)
package com.ravi.basic;

import java.util.ArrayList;
import java.util.List;
import java.util.Set;
import java.util.stream.Collector;
import java.util.stream.Collectors;

public class StreamDemo8
{
          public static void main(String args[])
          {
                    //Get the name of the Player in upper-case from Player Object

                     List<Player> playerList = createPlayerList();

                    Set<String> playerName = playerList.stream().map(player ->
player.name().toUpperCase()).collect(Collectors.toSet());

                    System.out.println(playerName);




          }

          public static List<Player> createPlayerList()
          {
                    List<Player> al = new ArrayList<>();
                    al.add(new Player(18, "Virat"));
                    al.add(new Player(45, "Rohit"));
                    al.add(new Player(7, "Dhoni"));
                    al.add(new Player(18, "Virat"));
                    al.add(new Player(90, "Bumrah"));
                    al.add(new Player(67, "Hardik"));

                    return al;
          }
```

```
}


record Player(Integer id, String name)
{

}

------------------------------------------------------------
package com.ravi.basic;

import java.util.HashSet;
import java.util.List;
import java.util.Set;
import java.util.stream.Collectors;

public class MapDemo2
{
	public static void main(String args[])
	{
		//Get the name of the Player in upper-case from Player Object
		Set<Player> playerList = createPlayerList();

		List<String> collect = playerList.stream().map(player ->
player.name().toUpperCase()).collect(Collectors.toList());
		System.out.println(collect);

	}

	public static Set<Player> createPlayerList()
	{
		Set<Player> player = new HashSet<>();
		player.add(new Player(18, "Virat"));
		player.add(new Player(45, "Rohit"));
		player.add(new Player(7, "Dhoni"));
		player.add(new Player(18, "Virat"));
		player.add(new Player(90, "Bumrah"));
		player.add(new Player(67, "Hardik"));

		return player;
	}
}
```

```
record Player1(Integer id, String name)
{

}
---------------------------------------------------------------
//Find the length of the name

package com.ravi.stream_demo;

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

public class FindLegthOfName {

	public static void main(String[] args)
	{
		List<String> listOfName =
Arrays.asList("Rahul","Scott","Raj","Elina","Aaarti","Puja");

		listOfName.stream().map(str -> str.length()).forEach(System.out::println);

	}

}
--------------------------------------------------------------
//Retrieve first character of all the given name

package com.ravi.stream_demo;

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

public class RetrieveFirstCharacter {

	public static void main(String[] args)
	{
	  List<String> listOfName = Arrays.asList("Jaya","Arnav","Virat","Aryan");

	  List<Character> collect = listOfName.stream().map(str ->
str.charAt(0)).collect(Collectors.toList());
	  System.out.println(collect);
```

```
        }

}
------------------------------------------------------------
//Retrieve the employee salary from employee object

package com.ravi.stream_demo;

import java.util.ArrayList;

record Employee(Integer empId, String empName, Double empSalary, Integer age)
{

}

public class RetrieveSalary {

        public static void main(String[] args)
        {
                ArrayList<Employee> listOfEmployees = new ArrayList<>();
                listOfEmployees.add(new Employee(111, "A", 70000D,24));
                listOfEmployees.add(new Employee(222, "B", 60000D,26));
                listOfEmployees.add(new Employee(333, "C", 45000D,23));
                listOfEmployees.add(new Employee(444, "D", 65000D,28));
                listOfEmployees.add(new Employee(555, "E", 55000D,29));

                System.out.println("Salary of all the employees :");
                listOfEmployees.stream().map(emp ->
emp.empSalary()).forEach(System.out::println);

        }

}
------------------------------------------------------------
//Retrieve the name whose length is > 3 and convert those
//names in uppercase

package com.ravi.stream_demo;

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

public class FilterNameAndUpperCase {
	public static void main(String[] args)
	{
		List<String> listOfName =
Arrays.asList("Rahul","Scott","Raj","Elina","Ram","Puja");

		listOfName.stream().filter(str -> str.length()>3).map(name ->
name.toUpperCase()).forEach(System.out::println);



	}

}
```
----
public Stream flatMap(Function<? super T,? extends Stream<? extends R>> mapper)

It is a predefined method of Stream interface.

The map() method produces one output value for each input value in the stream So if there are n elements in the stream, map() operation will produce a stream of n output elements.

flatMap() is two step process i.e. map() + Flattening. It helps in converting Collection<Collection<T>> into Collection<T> [to make flat i.e converting Collections of collection into single collection or merging of all the collection into single Collection]

```
package com.ravi.testing;

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

public class FlatMapDemo {

	public static void main(String[] args)
	{
		List<String> indPlayer = Arrays.asList("Surya", "Tilak", "Akshar","Pandaya");

	List<String> engPlayer = Arrays.asList("Salt","Butler","Archer","Rashid");
```

```
        List<List<String>> icc = Arrays.asList(indPlayer , engPlayer);

        System.out.println(icc);

        //use flatMap for mapping + flattening

        List<String> collect = icc.stream().flatMap(list -> list.stream()).collect(Collectors.toList());

        System.out.println(collect);

        }

}
```

```
//flatMap()
//map + Flattening [Converting Collections of collection into single collection]
package com.ravi.basic;

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

public class StreamDemo9
{
        public static void main(String[] args)
        {
                List<String> list1 = Arrays.asList("A","B","C");
                List<String> list2 = Arrays.asList("D","E","F");
                List<String> list3 = Arrays.asList("G","H","I");

                List<List<String>> nestedColl = Arrays.asList(list1, list2, list3);
                System.out.println("Original Nested Collection :"+nestedColl);

                List<String> collect = nestedColl.stream().flatMap(list ->
list.stream()).collect(Collectors.toList());
                System.out.println(collect);

        }
}
```

```
//Flattening of prime, even and odd number
package com.ravi.basic.flat_map;
```

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

public class FlatMapDemo1
{
	public static void main(String[] args)
	{
     List<Integer> primeNumbers = Arrays.asList(5,7,11);
     List<Integer> evenNumbers = Arrays.asList(2,4,6);
     List<Integer> oddNumbers = Arrays.asList(1,3,5);

     List<List<Integer>> nestedColl = List.of(primeNumbers,evenNumbers,oddNumbers);
     System.out.println(nestedColl);

    List<Integer> flatList = nestedColl.stream().flatMap(num ->
num.stream()).collect(Collectors.toList());

    System.out.println(flatList);


	}

}
```

```
//Fetching first character using flatMap()
package com.ravi.basic.flat_map;

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

public class FlatMapDemo2
{
	public static void main(String[] args)
	{
	   List<String> listOfNames = Arrays.asList("Jaya","Aryan","Virat","Aakash");

		List<Character> list = listOfNames.stream().flatMap(str ->
Stream.of(str.charAt(0))).toList(); //toList() java16V
```

```
		System.out.println(list);
	}

}
--------------------------------------------------------------------
package com.ravi.basic.flat_map;

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

class Product
{
	private Integer productId;
	private List<String> listOfProducts;

	public Product(Integer productId, List<String> listOfProducts)
	{
		super();
		this.productId = productId;
		this.listOfProducts = listOfProducts;

	}

	public Integer getProductId()
	{
		return productId;
	}

	public List<String> getListOfProducts()
	{
		return listOfProducts;
	}
}

public class FlatMapDemo3
{
	public static void main(String[] args)
	{
		List<Product> listOfProduct = Arrays.asList(
	new Product(1, Arrays.asList("Camera", "Mobile", "Laptop")),
	new Product(2, Arrays.asList("Bat", "Ball", "Wicket")),
        new Product(3, Arrays.asList("Chair", "Table", "Lamp")),
        new Product(4, Arrays.asList("Cycle", "Bike", "Car"))

                );

                List<String> collect = listOfProduct.stream().flatMap(product ->
product.getListOfProducts().stream()).collect(Collectors.toList());

                System.out.println(collect);

        }

}
```

01-02-2025

Working with Primitive Streams :

Streams works with collections of objects and not primitive types.

Now, to provide a way to work with the three most used primitive types – int, long and double, Java provides three primitive specialized implementations of Stream.

IntStream (represents sequence of primitive int elements)
LongStream (represents sequence of primitive long elements)
DoubleStream (represents sequence of primitive double elements)

Method of IntStream, LongStream and DoubleStream :

IntStream, LongStream and DoubleStream are the predefined interfaces available in java.util.stream sub package.

These interfaces contain static method of(T ...values) through which we can create corresponding type of element.

Arrays which is a predefined class in java.util package provides a predefined method called stream() which will also convert corresonding array object into Stream type

```
public static IntStream stream(int [] array);
public static LongStream  stream(long [] array);
public static DoubleStream stream(double [] array);
```

Note : By using above methods we can convert the array into corresponding Stream Type.

```
package com.ravi.testing;

import java.util.Arrays;
import java.util.stream.DoubleStream;
import java.util.stream.IntStream;
import java.util.stream.LongStream;

public class PrimitiveToStreamDemo1
{
        public static void main(String[] args)
        {
                IntStream intStream = IntStream.of(1,2,3,4,5,6,7,8);
                LongStream longStream = LongStream.of(1L,2L,3L,4L,5L);
                DoubleStream doubleStream = DoubleStream.of(1.1,1.2,1.3,1.4,1.5);
                intStream.forEach(System.out::println);
                System.out.println();
                longStream.forEach(System.out::println);
                System.out.println();
                doubleStream.forEach(System.out::println);

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


                int a[] = {1,2,3,4,5};
                IntStream intStream2 = Arrays.stream(a);

                long l[] = {1L, 2L, 3L, 4L};
                LongStream longStream2 = Arrays.stream(l);

                double d[] = {1.2, 2.6, 3.9, 8.9};
                DoubleStream doubleStream2 = Arrays.stream(d);

                intStream2.forEach(System.out::print );
                System.out.println();
                longStream2.forEach(System.out::print);
                System.out.println();
                doubleStream2.forEach(System.out::print);

        }
```

}

IntStream flatMapToInt(Function<? super T, ? extends IntStream> mapper)

It is a predefined method of Stream interface which comes under flattening.

It allows us to transform each element of the stream into an IntStream (a stream of primitive int values) and then flattens these resulting streams into a single IntStream.

Note : IntStream is a specialized stream for working with int values avilable in java.util.stream sub package.

```
package com.ravi.basic.flat_map;

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

public class FlatMapToIntDemo1 {

        public static void main(String[] args)
        {
                int []a1 = new int[] {1,2,3};
                int []a2 = new int[] {4,5,6};
                int []a3 = new int[] {7,8,9};

          List<int[]> nestedArray = Arrays.asList(a1,a2,a3);

                IntStream intStream = nestedArray.stream().
                                flatMapToInt(array-> IntStream.of(array));
                intStream.forEach(System.out::print);




        }

}
```

LongStream flatMapToLong(Function<? super T, ? extends LongStream> mapper) :

It is a predefined method of Stream interface which comes under flattening.

It allows us to transform each element of the stream into a LongStream (a stream of primitive long values) and then flattens these resulting streams into a single LongStream.

Note : LongStream is a specialized stream for working with long values avilable in java.util.stream sub package.

```
package com.ravi.basic.flat_map;

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

public class FlatMapToLongDemo1 {

	public static void main(String[] args)
	{
		long []arr1 = new long[] {23,33,43};
		long []arr2 = new long[] {53,63,73};
		long []arr3 = new long[] {83,93,103};

		List<long[]> longArray = Arrays.asList(arr1, arr2,arr3);
		LongStream flatMapToLong = longArray.stream().
					flatMapToLong(array -> Arrays.stream(array));

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

}
```

DoubleStream flatMapToDouble(Function<? super T, ? extends DoubleStream> mapper)

It is a predefined method of Stream interface which comes under flattening.

It allows us to transform each element of the stream into an DoubleStream (a stream of primitive double values) and then flattens these resulting streams into a single DoubleStream.

Note : DoubleStream is a specialized stream for working with double values avilable in java.util.stream sub package.

```
package com.ravi.basic.flat_map;
```

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.DoubleStream;

public class FlatMapToDoubleDemo1
{
   public static void main(String[] args)
   {
         double d1[] = new double[]{1.1, 1.2, 1.3};
         double d2[] = new double[]{2.1, 2.2, 2.3};
         double d3[] = new double[]{3.1, 3.2, 3.3};

         List<double[]> listOfDoubleArrays = Arrays.asList(d1,d2,d3);

         DoubleStream doubleStream = listOfDoubleArrays.stream()
            .flatMapToDouble(array -> DoubleStream.of(array));

         // Print each double value in the flattened stream
         doubleStream.forEach(System.out::println);
      }
}
```

**Difference between map() and flatMap()
----

map() method transforms each element into another single element.

flatMap() transforms each element into a stream of elements and then flattens those streams into a single stream.

We should use map() when you want a one-to-one transformation, and we should use flatMap() when dealing with nested structures or when you need to produce multiple output elements for each input element.

----

public Stream sorted() :

It is a predefined method of Stream interface.
It provides default natural sorting order.
The return type of this method is Stream.
It has an overloaded method which accept Comparator<T> as a parameter
through which we can provide user-defined sorting logic

public Stream distinct() :

It is a predefined method of Stream interface.

If we want to return stream from another stream by removing all the duplicates then we should use distinct() method.

```
package com.ravi.basic;

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

public class StreamDemo10
{
        public static void main(String[] args)
        {
                //Print the numbers in ascending order
                List<Integer> listOfNum = List.of(89,67,56,45,23,15);
                listOfNum.stream().
                  sorted((i1,i2)-> i1.
                                compareTo(i2)).
                        forEach(System.out::println);
                System.out.println("================================");

                //Print the numbers in descending order
                List<Integer> listOfNumber = List.of(89,67,56,45,23,15);
                listOfNumber.stream().
                  sorted((i1,i2)-> i2.compareTo(i1)).
                    forEach(System.out::println);
                System.out.println("================================");

                //Print the names in Ascending order
                Stream<String> strOfName = Stream.of("Ankit","Scott","Smith","James");
                strOfName.sorted((s1,s2)-> s1.compareTo(s2)).forEach(System.out::println);

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

                //Print the names in Descending order
                Stream<String> strmOfName = Stream.of("Ankit","Scott","Smith","James");
                strmOfName.sorted((s1,s2)-> s2.compareTo(s1)).forEach(System.out::println);

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

```
                    Stream<String> s = Stream.of("Virat", "Rohit", "Dhoni", "Virat",
"Rohit","Aswin","Bumrah");
                            s.distinct().
                            sorted((s1,s2)-> s2.compareTo(s1)).
                            forEach(System.out::println);

          }

}
```

```
package com.ravi.basic;
import java.util.stream.Stream;
public class StreamDemo11
{
          public static void main(String[] args)
          {
                    Stream<Integer> numbers = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 3, 4, 5);

                    numbers.distinct().forEach(System.out::println);


          }
}
```
----
public Stream<T> limit(long maxSize) :

It is a predefined method of Stream interface to work with sequence of elements.

The limit() method is used to limit the number of elements in a stream by providing maximum size.

It creates a new Stream by taking the data from original Stream.

Elements which are not in the range or beyond the range of specified limit will be ignored.

----
public Stream<T> skip(long n) :

It is a predefined method of Stream interface which is used to skip the elements from begning of the Stream.

It returns a new stream that contains the remaining elements after skipping the specified number of elements which is passed as a parameter.

```
package com.ravi.basic;
import java.util.stream.Stream;
public class StreamDemo12
{
	public static void main(String[] args)
	{
		Stream<String> s = Stream.of("Virat", "Rohit", "Dhoni", "Zaheer",
"Raina","Sahwag","Sachin","Bumrah");
		s.limit(7).forEach(System.out::println);

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

		 Stream<String> of = Stream.of("Virat", "Rohit", "Rahul","Gill",
"Pant","Bumrah","Nitish");
		 of.skip(3).forEach(System.out::println);

	}
}
```

How many ways we can create Stream :

There are 4 ways to craete the Stream as shown in the Program

```
package com.ravi.advanced;

import java.util.Arrays;
import java.util.List;
import java.util.stream.DoubleStream;
import java.util.stream.Stream;

public class WaysOfStreamCreation {

	public static void main(String[] args)
	{
		//Case 1:
		List<Integer> list = List.of(1,2,3);
		list.stream().forEach(System.out::println);

		//Case 2
		double arr[] = {1.2, 3.6, 8.9};
		DoubleStream stream = Arrays.stream(arr);
	   stream.forEach(System.out::println);
		//Case 3
		Stream.of(12,90,78).forEach(System.out::println);


			//Case 4 [How to generate Infinite Stream]
		//generate(Supplier<T> g)
			Stream.generate(() -> Math.random()).limit(10).forEach(System.out::println);

			Stream.iterate(1, num -> num+1).limit(10).forEach(System.out::println);

	}

}
```

03-02-2025
----
public Stream<T> peek(Consumer<? super T> action) :

It is a predefined method of Stream interface which is used to perform a side-effect operation on each element in the stream while the stream remains unchanged.

It is an intermediate operation that allows us to perform operation on each element of Stream without modifying original.

The peek() method takes a Consumer as an argument, and this function is applied to each element in the stream. The method returns a new stream with the same elements as the original stream.

```
package com.ravi.basic;

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

public class StreamDemo13
{
	public static void main(String[] args)
	{
		Stream<String> streamOfFruits = 
Stream.of("Apple","Mango","Grapes","Kiwi","pomogranate");

		List<Integer> fruitLength = streamOfFruits
```

```
                    .peek(str -> System.out.println("Peeking from Original: " + str.toUpperCase()))
                    .map(fruit -> fruit.length())
                    .collect(Collectors.toList());
                System.out.println("-----------------");
                System.out.println(fruitLength);

        }

}
```

Note :- peek(Consumer<T> cons) will not modify the Original Source.
----
public Stream<T> takeWhile(Predicate<T> predicate) :

It is a predefined method of Stream interface introduced from java 9v which is used to perform a side-effect operation on each element in the stream while the stream remains unchanged.

*It is used to create a new stream that includes elements from the original stream only as long as they satisfy a given predicate.

```
package com.ravi.basic;

import java.util.stream.Stream;

public class StreamDemo14
{
 public static void main(String[] args)
 {
          Stream<Integer> numbers = Stream.of(10,11,9,13,2,1,100);

     numbers.takeWhile(n -> n > 9).forEach(System.out::println);

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


     numbers = Stream.of(12,2,10,3,4,5,6,7,8,9);

     numbers.takeWhile(n -> n%2==0).forEach(System.out::println);


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

     numbers = Stream.of(1,2,3,4,5,6,7,8,9);
```

```
    numbers.takeWhile(n -> n < 9).forEach(System.out::println);

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


    numbers = Stream.of(11,2,13,4,5,6,7,8,9);

    numbers.takeWhile(n -> n > 9).forEach(System.out::println);

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

    Stream<String> stream = Stream.of("Ravi", "Ankit", "Rohan", "Aman", "Ravish");

    stream.takeWhile(str -> str.charAt(0)=='R').forEach(System.out::println);



 }
}
```

----
public Stream<T> dropWhile(Predicate<T> predicate) :

It is a predefined method of Stream interface introduced from java 9 which is used to create a new stream by excluding elements from the original stream as long as they satisfy a given predicate.

```
package com.ravi.basic;

import java.util.stream.Stream;

public class StreamDemo15 {

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

                numbers.dropWhile(num -> num < 7).forEach(System.out::println);

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

                numbers = Stream.of(15, 8, 7, 9, 5, 6, 7, 8, 9, 10);
```

```
                    numbers.dropWhile(num -> num > 5).forEach(System.out::println);


          }

}
```

Optional<T> class in Java :

It is a predefined final and immutable class available in java.util package from java 1.8v.

It is a container object which is used to represent an object (Optional object) that may or may not contain a non-null value.

If the value is available in the container, isPresent() method will return true and get() method will return the actual value.

It is very useful in industry to avoid NullPointerException.

Methods of Optional<T> class :
----
1) public static Optional<T> ofNullable(T x) :

It will return the object of Optional class with specified value. If the specified value is null then this method will return an empty object of the optional class.

2) public boolean isPresent() :

It will return true, if the value is available in the container otherwise it will return false.

3) public T get() :

It will get/fetch the value from the container, if the value is not available then it will throw java.util.NoSuchElementException.

4) public T orElse(T defaultValue) :

It will return the value, if available otherwise it will return the specified default value.

5) public static Optional<T> of (T value) :

It will return the optional object with the specified value that is non- null value becuase it does

not contain any container.

6) public static Optional<T> empty() :

It will return an empty Optional Object.

7) public java.util.stream.Stream stream()

It will Convert optional to Stream.

8) public void ifPresent(Consumer<T> cons) :

It Used to consume/accept the value from optional container if the value is not null.

```
package com.ravi.optional_demo;

import java.util.Optional;

public class OptionDemo1 {

        public static void main(String[] args)
        {
                String str = null;

                Optional<String> cont = Optional.ofNullable(str);


                //orElse method
           String val = cont.orElse("No value in the container by orElse");
           System.out.println(val);

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

           if(cont.isPresent())
           {
                 String value = cont.get();
                 System.out.println("Value of the container :"+value);
           }
           else
           {
                System.err.println("No Value in the container isPresent()");
           }
```

```
	}

}
-----------------------------------------------------------
//Writing different style of setter and getter

package com.ravi.optional_demo;

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

class Employee
{
	private Integer employeeId;
	private String employeeName;

	public Employee()
	{
		super();
	}

	public Employee(Integer employeeId, String employeeName)
	{
		super();
		this.employeeId = employeeId;
		this.employeeName = employeeName;
	}

	public Optional<Integer> getEmployeeId()
	{
		return Optional.ofNullable(employeeId);
	}

	public Optional<String> getEmployeeName()
	{
		return Optional.ofNullable(employeeName);
	}
}

public class OptionalDemo2 {
	public static void main(String[] args)
	{
		//Employee emp1 = new Employee();
		Employee emp1 = new Employee(111,"Scott");


		//Approach 1
		Optional<Integer> employeeId = emp1.getEmployeeId();

		if(employeeId.isPresent())
		{
			System.out.println("Id is :"+employeeId.get());
		}
		else
		{
			System.err.println("employeeid is not available");
		}

		//Approach2
		  Optional<String> employeeName = emp1.getEmployeeName();
		  String empName = employeeName.orElse("employee name is not available");
	System.out.println("Name is :"+empName);



	}

}
```

```
//Replcing null by uisng Optional.empty()

package com.ravi.optional_demo;

import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

public class OptionalDemo3 {

	public static void main(String[] args)
	{
		List<Optional<String>> listOfcity = new ArrayList<>();
		listOfcity.add(Optional.of("Hyderabad"));
		listOfcity.add(Optional.of("Chennai"));
		listOfcity.add(Optional.of("Mumbai"));
		listOfcity.add(Optional.of("Nagpur"));
		listOfcity.add(Optional.empty());

		for(Optional<String> opt : listOfcity)
		{
			if(opt.isPresent())
			{
				System.out.println(opt.get());
			}
			else
			{
				System.out.println("No Value in the List");
			}

		}

	}

}
```

04-02-2025

```
//Immutability of Optional class

package com.ravi.optional_class_demo;

import java.util.Optional;
public class OptionalDemo4
{
   public static void main(String[] args)
   {

      Optional<String> optl = Optional.of("India");
      System.out.println(optl.hashCode());

      Optional<String> newOptnl = modifyOptional(optl);
      System.out.println(newOptnl.hashCode());

      // Check if the original Optional is still the same
      System.out.println("Address is :" + (optl == newOptnl));
```

```
    }

    public static Optional<String> modifyOptional(Optional<String> optional)
    {
        if (optional.isPresent())
        {
            return Optional.of("Modified: " + optional.get());
        }
        else
        {
            return Optional.empty();
        }
    }
}
```

Note : India object created is immutable object so when we create
"Modified india" by using of method of Optional class it is created in different memory location so immutable.

Method Reference :

Method Reference in java :

It is a new feature introduced from java 1.8 onwards.

It is mainly used to write concise coding.

By using method reference we can refer an existing method which is available at API level or Project level.

We can use this technique in the body of Lambda expression just to call method defination.

The enitire method body will be automatically placed into Lambda Expression.

It is used to enhance the code reusability.

It uses :: (Double Colon Operator)

While working with Lambda expression we need to write the Lambda Method Body but while working with Method reference we can refer an existing method which is already available in the package or Project.

There are 4 types of method reference

1) Static Method Reference(ClassName::staticMethodName)
2) Instance Method Reference(objectReference::instanceMethodName)
3) Constructor Reference (ClassName::new)
4) Arbitrary Referenec (ClassName::instanceMethodName)

```
package com.ravi.testing;

@FunctionalInterface
interface Worker
{
	void work();
}

public class MethodRefDemo1
{
	public static void main(String[] args)
	{
	  //Lambda Expression
		Worker w1 = () -> System.out.println("Worker is working");
		w1.work();

		//Method Reference
		Worker w2 = new Employee()::work;
		w2.work();

	}

}


class Employee
{
	public void work()
	{
		System.out.println("Employee is Working");
	}
}
```

```
-----------------------------------------------------------------
package com.ravi.testing;

@FunctionalInterface
interface Worker
{
	void work();
}

public class MethodRefDemo2
{
	public static void main(String[] args)
	{
	  Worker w1 = Employee::salary;
	  w1.work();

	}

}


class Employee
{
	public static void salary()
	{
		System.out.println("Employee is Working for Salary");
	}
}
-----------------------------------------------------------------
package com.ravi.testing;

@FunctionalInterface
interface Worker
{
	void work(double salary);
}

public class MethodRefDemo3
{
	public static void main(String[] args)
	{
	   Worker w1 = new Employee()::salary;
	   w1.work(55000);
```

```
        }
}

class Employee
{
        public void salary(double salary)
        {
                System.out.println("Employee Salary is :"+salary);
        }
}
------------------------------------------------------------------
//Program on static Method Reference :
---------------------------------------
package com.ravi.static_method_reference;

import java.util.Vector;
import java.util.function.Consumer;

class EvenOrOdd
{
        public static void isEven(int number)
  {
      if (number % 2 == 0)
      {
          System.out.println(number + " is even");
      }
      else
      {
          System.out.println(number + " is odd");
      }
  }
}
public class StaticMethodReferenceDemo1
{
        public static void main(String[] args)
  {
     Vector<Integer> numbers = new Vector<>();
     numbers.add(5);
     numbers.add(2);
     numbers.add(9);
     numbers.add(12);
```

```
      //By using Lambda Expression
            numbers.forEach(num -> EvenOrOdd.isEven(num));

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

            //By using Method Reference
            numbers.forEach(EvenOrOdd::isEven);

    }
}
----------------------------------------------------------------
//Program on instance Method reefernce :

package com.ravi.instance_method_reference;

@FunctionalInterface
interface Trainer
{
  void getTraining(String name, int experience);
}

class InstanceMethod
{
   public void getTraining(String name, int experience)
   {
            System.out.println("Trainer name is :"+name+" having "+experience+" years of
experience.");
   }
}

public class InstanceMethodReferenceDemo
{
    public static void main(String[] args)
    {
          //Using Lambda Expression
          Trainer t1 = (name,  exp)-> System.out.println("Trainer name is :"+name+" and total
experience is :"+exp+ " years");
          t1.getTraining("Smith", 5);


          //By using Method reference
          Trainer t2 = new InstanceMethod()::getTraining;
          t2.getTraining("Scott", 10);
    }
}
--------------------------------------------------------------------
Constructor reference (ClassName::new)
----------------------------------------
package com.ravi.constructor_reference;

@FunctionalInterface
interface A
{
    Test createObject();
}

class Test
{
    public Test()
    {
        System.out.println("Test class Constructor invoked");
    }
}
public class ConstructorReferenceDemo1
{
    public static void main(String[] args)
    {

            //By using Lambda Expression
            A a1 = () -> new Test();
            a1.createObject();

            System.out.println(".........");
            //By using Method Reference
            A a2 = Test::new;
            a2.createObject();

    }
}
--------------------------------------------------------------------
package com.ravi.constructor_reference;

import java.util.function.Function;

class Accept
```

```
{
            private int x;

            public Accept(int x)
            {
                        this.x = x;
            }

            public int getX()
            {
                        return this.x;
            }
}

public class ConstructorReferenceDemo2
{
            public static void main(String[] args)
            {
      Function<Integer,Accept> fn1 = Accept::new;
      Accept obj = fn1.apply(90);

      System.out.println("The value of x :"+obj.getX());
            }

}
--------------------------------------------------------------------
//How to create Array Object using Constructor reference :
-------------------------------------------------------------
package com.ravi.constructor_reference;

import java.util.Arrays;
import java.util.function.Function;

class Person
{
    private String name;   //Person persons[] = new Person[5];

    public Person(String name)
    {
        this.name = name;
    }

    public String getName()
```

```
   {
      return name;
   }

         @Override
         public String toString() {
                  return "Person [name=" + name + "]";
         }


}

public class ConstructorReferenceDemo3
{
   public static void main(String[] args)
   {
     Function<Integer,Person[]> fn2 =  Person[]::new;

     Person []persons = fn2.apply(3);  //3 is the size of the array

     persons[0] = new Person("Scott");
     persons[1] = new Person("Smith");
     persons[2] = new Person("Martin");

     System.out.println(Arrays.toString(persons));
   }
}

--------------------------------------------------------------
package com.ravi.constructor_reference;

import java.util.Scanner;
import java.util.function.Function;

class Student
{
         private Integer studentId;
         private String studentName;

         public Student(Integer studentId, String studentName)
         {
                  super();
                  this.studentId = studentId;
```

```
		this.studentName = studentName;
	}

	@Override
	public String toString()
	{
		return "Student [studentId=" + studentId + ", studentName=" + studentName + "]";
	}
}

public class ConstructorReferenceDemo4
{
	public static void main(String[] args)
	{
	 Scanner sc = new Scanner(System.in);

	 //Creating Student Array Object
	 Function<Integer,Student[]> fn1 = Student[]::new;

	 System.out.print("Enter the size of the Array :");
	 int size = sc.nextInt();

	 Student[] students = fn1.apply(size);

	 for(int i=0; i<students.length; i++)
	 {
		 System.out.print("Enter the Student id :");
		 int id = sc.nextInt();

		 System.out.print("Enter Student Name :");
		 String name = sc.nextLine();
		 name = sc.nextLine();

		 students[i] = new Student(id, name);
	 }


	 System.out.println("Fetching the data from Array Object :");
	 for(Student std : students)
	 {
		 System.out.println(std);
	 }
```

```
        }

}
------------------------------------------------------------------
```
Arbitrary Reference type :

By using Arbitrary reference type we can call non static method with the class name.

Here internally JVM will pass this keyword to the method which referes the current Object.

It will work with the interfaces which are working as a method parameter because this keyword is available as a method parameter.

```
------------------------------------------------------------------
package com.ravi.arbitary_reference;

import java.util.TreeSet;

public class Test {

        public static void main(String[] args)
        {
                TreeSet<String> ts = new TreeSet<>(String::compareTo);
                ts.add("C");
                ts.add("B");
                ts.add("A");

                System.out.println(ts);

        }

}
------------------------------------------------------------------
package com.ravi.arbitary_reference;

import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class ArbitaryRefDemo1
{
  public static void main(String[] args)
```

```
 {
          List<Integer> listOfNumbers = Arrays.asList(9,5,6,2,4,1);

          //By Using Lambda Expression
          Collections.sort(listOfNumbers, (i1,i2)-> i1.compareTo(i2));
          System.out.println(listOfNumbers);

          //By using Method Reference
          Collections.sort(listOfNumbers, Integer::compareTo);
          System.out.println(listOfNumbers);


         //By Using Lambda Expression
         String [] players = {"Virat", "Rohit", "Zaheer", "Rishab", "Abhishek"};
         Arrays.sort(players,(s1, s2)-> s2.compareTo(s1));
         System.out.println(Arrays.toString(players));

         //By using Method Reference
         String [] players1 = {"Virat", "Rohit", "Zaheer", "Rishab", "Abhishek"};
         Arrays.sort(players1, String::compareTo);
         System.out.println(Arrays.toString(players1));

 }
}
----------------------------------------------------------------
package com.ravi.arbitary_reference;

import java.util.Arrays;
import java.util.Comparator;

class Person
{
   String name;

   public Person(String name)
   {
      this.name = name;
   }

   public int personInstanceMethod1(Person person)
   {
      return  this.name.compareTo(person.name);
   }
	@Override
	public String toString() {
		return "Person [name=" + name + "]";
	}


}

public class ArbitraryRefDemo2
{
   public static void main (String[] args) throws Exception
   {

       Person[] personArray = {new Person("Zuber"),new Person("Raj"), new Person("Ankit"),
new Person("Abhishek")};

       Arrays.sort(personArray, Person::personInstanceMethod1);

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

   }

}
------------------------------------------------------------
package com.ravi.arbitary_reference;

@FunctionalInterface
interface MyInterface<T,U,V,R>
{
	R myApply(T t, U u, V v);
}

class Addition
{
	public Integer doSum(String x, String y)
	{
		return Integer.parseInt(x) + Integer.parseInt(y);
	}
}

public class ArbitaryRefDemo3
{
        public static void main(String[] args)
        {
   //By Using Lambda expression
        MyInterface<Addition,String,String,Integer> fn1 = (a, t, v) -> a.doSum(t, v);
        Integer result = fn1.myApply(new Addition(), "100", "200");
        System.out.println("Result is :"+result);

        //By Using Method Reference
        MyInterface<Addition,String,String,Integer> fn2 = Addition::doSum;
        Integer res = fn2.myApply(new Addition(), "500", "500");
        System.out.println("Result is :"+res);

        }

}
```
==

New Date and Time :

LocalDate :

It is a predefined final class which represents only Date. The java.util.Date class is providing Date and Time both so, only to get the Date we need to use LocalDate class available in java.time package.

```
        LocalDate d = LocalDate.now();
```

Here now is a static method of LocalDate class and its return type is LocalDate class. (static Factory Method)

LocalTime :

It is also a final class which will provide only time.

```
        LocalTime d = LocalTime.now();
```

Here now is a static Factory method of LocalTime class and its return type is LocalTime (static Factory Method).

LocalDateTime :

It is also a final class which will provide Date and Time both without a time zone. It is a combination of LocalDate and LocalTime class.

```
         LocalDateTime d = LocalDateTime.now();
```

```
package com.ravi.new_date_time;

import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.ZoneId;
import java.time.ZonedDateTime;

public class Demo1
{
	public static void main(String[] args)
	{
	   LocalDate d = LocalDate.now();
	   System.out.println(d);

	   LocalTime t = LocalTime.now();
	   System.out.println(t);

	    LocalDateTime dt = LocalDateTime.now();
	    System.out.println(dt);


	}
}
```

ZonedDateTime : (Date and time + Time zone)

It is a final class available in java.time package.

It is also provides date and time along with time zone so, by using this class we can work with different time zone in a global way.

```
ZonedDateTime x = ZonedDateTime.now();
ZoneId zone = x.getZone();
```

getZone() is a predefined non static method of ZonedDateTime class which returns ZoneId class which is abstract and sealed class, this ZoneId class provides the different zones, by using getAvailableZoneIds() static method we can find out the total zone available using this ZoneId class.

```
package com.ravi.new_date_time;
```

```
import java.time.ZoneId;
import java.time.ZonedDateTime;

public class Demo2
{
        public static void main(String[] args)
        {
                ZonedDateTime z = ZonedDateTime.now();
                System.out.println(z);

           ZoneId zone = z.getZone();
           System.out.println(zone);

           System.out.println(ZoneId.getAvailableZoneIds());


        }

}
```

Different of() static methods :

```
List.of();
Set.of();
Map.of();
Stream.of();
Optional.of();
ZoneId.of();
```

```
package com.ravi.new_date_time;

import java.time.ZoneId;
import java.time.ZonedDateTime;

public class Demo3
{
   public static void main(String[] args)
   {
      ZoneId ausTimeZone = ZoneId.of("Australia/Sydney");
      ZonedDateTime aus = ZonedDateTime.now(ausTimeZone);
      System.out.println("Current Date and Time in Australia Time Zone: " + aus);
```

```
        ZoneId canadaTimeZone = ZoneId.of("Canada/Atlantic");
        ZonedDateTime canada = ZonedDateTime.now(canadaTimeZone);
        System.out.println("Current Date and Time in Canada Time Zone: " + canada);


    }
}
```

Note : The abstract class ZoneId provides a static method of() which accept String ZoneId as a parameter and returns ZoneId class.

DateTimeFormatter :

It is a predefined final class available in java.time.format sub package.

It is mainly used to formatting and parsing date and time objects according to Java Date and Time API.

Method :
----
public static DateTimeFormatter ofPattern(String pattern) :

It is a static method of DateTimeFormatter class, It creates a
formatter using user specified String pattern ("dd-MM-yyyy HH:mm:ss")
and LocalDateTime class contains format method which takes
DateTimeFormatter as a parameter and returns the String value.

```
package com.ravi.new_date_time;

import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

public  class Demo4
{
  public static void main(String[] args)
  {
        LocalDateTime now = LocalDateTime.now();
        System.out.println(now);

        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd-MM-YYYY");
        String formattedDateTime = now.format(formatter);
        System.out.println("Formatted DateTime: " + formattedDateTime);
  }
}
```

Terminal Operation in Stream in java:

In Java's Stream API, a terminal operation is an operation that produces a result or a side-effect operation.

Unlike intermediate operations, which returns a new stream, terminal operation consumes the elements of the stream.

Once a terminal operation is applied to a stream, the stream is considered consumed and cannot be reused.

It is a final operation.

Methods of Terminal Operation :

----

1) public long count()

2) public Optional<T> min(Comparator<? super T> comparator)

3) public Optional<T> max(Comparator<? super T> comparator)

4) public Optional<T> findAny()

5) public Optional<T> findFirst()

6) public boolean allMatch(Predicate<? super T> predicate)

7) public boolean anyMatch(Predicate<? super T> predicate)

8) public boolean noneMatch(Predicate<? super T> predicate)

9) public void forEach(Consumer<T> cons)

10) public Optional<T> reduce(BinaryOperator<T> accumulator)

11) public R collect(Collector<? super Integer,A,R> collect)

----

public long count() :

The count operation returns the number of elements available in the stream.

It is a terminal operation that terminates the stream after its execution.

Basically it used to count the number of elements after filter() method.

```
package com.ravi.advanced.count_demo;

import java.util.stream.Stream;

public class CountDemo1
{
	public static void main(String[] args)
	{
		long count = Stream.of("Ravi","Raj","Elina","Aryan","Sachin").count();
		System.out.println(count);

	}

}
```

```
package com.ravi.advanced.count_demo;

//Count the name whose length is greater than 3

import java.util.List;

public class CountDemo2
{
    public static void main(String[] args) {
        List<String> listOfName = List.of("Raj","Ravi","Virat","Rohit","Ram","Bumrah","Sachin");

        long names = listOfName.stream().filter(name -> name.length()>3).count();
        System.out.println("Names whose length is > 3 are :"+names);
    }
}
```

```
package com.ravi.advanced.count_demo;

//Count Unique elements by using Stream API
```

```
import java.util.List;

public class CountDemo3
{
   public static void main(String[] args) {
         List<String> listOfName = List.of("Raj","Raj","Ravi","Virat","Raj");

      long count = listOfName.stream()
                   .distinct()
                   .count();

      System.out.println("Count of unique elements: " + count);
   }
}

package com.ravi.advanced.count_demo;

//Count the names which are containing the character A
import java.util.Arrays;
import java.util.List;

public class CountDemo4
{
   public static void main(String[] args) {
      List<String> list = Arrays.asList("Raj","Ravi","Rohit","Virat","Raj","Aradhya","scott");

      long count = list.stream()
                   .map(String::toUpperCase)
                   .filter(s -> s.contains("A"))
                   .distinct()
                   .count();

      System.out.println("Count of distinct strings containing 'A': " + count);
   }
}
```

Working with Predefined functional interfaces :

UnaryOperator<T> functional interface :

It is a predefined functional interface available in java.util.function sub package.

It is a functional interface in Java that represents an operation on a single operand that produces a result of the same type as its operand. This is a specialization of Function for the case where the operand and result are of the same type.

It has a single type parameter, T, which represents both the operand type and the result type.

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T,R>
{
  public abstract T apply(T x);
}
```

```
package com.ravi.advanced.count_demo;

import java.util.function.UnaryOperator;

public class UnaryOperatorDemo1 {

	public static void main(String[] args)
	{

		UnaryOperator<String> fn1 = str -> str.concat(" World");
		String concat = fn1.apply("Hello");
		System.out.println(concat);

		UnaryOperator<Integer> square = x -> x * x;
        System.out.println(square.apply(5));

	}

}
```

BinaryOperator<T> Functional interface :

It is a predefined functional interface available in java.util.function sub package.

It is a functional interface in Java that represents an operation upon two operands of the same type, producing a result of the same type as the operands.

This is a specialization of BiFunction for the case where the operands and the result are all of the same type.

It has two parameters of same type, T, which represents both the operand types and the result type.

```
@FunctionalInterface
public interface BinaryOperator<T> extends BiFunction<T,U,R>
{
  public abstract T apply(T x, T y);
}
-------------------------------------------------------------
import java.util.function.*;
public class Lambda16
{
	public static void main(String[] args)
	{
		BinaryOperator<Integer> add = (a, b) -> a + b;
        System.out.println(add.apply(3, 5));
	}
}
------------------------------------------------------------------
```

ToIntFunction<T> functional interface :

It is a predefined functional interface available in java.util.function sub package.

It is a functional interface in Java that represents a function that takes an argument of type T and returns an int value. This is typically used in streams when you need to perform operations that result in primitive int value.

```
@FunctionalInterface
public interface ToIntFunction<T>
{
   int applyAsInt(T value);
}
```

IntStream mapToInt(ToIntFunction<T> x)

```
package com.ravi.advanced.count_demo;

import java.util.ArrayList;
import java.util.function.ToDoubleFunction;
import java.util.function.ToIntFunction;

record Employee(String name, Integer experience)
{
}

public class Lambda17
{
  public static void main(String[] args)
  {
          ArrayList<Employee> listOfEmployee = new ArrayList<>();
          listOfEmployee.add(new Employee("Virat",12));
          listOfEmployee.add(new Employee("Rohit",12));
          listOfEmployee.add(new Employee("Bumrah",6));
          listOfEmployee.add(new Employee("Akshar",5));
          listOfEmployee.add(new Employee("Abhishek",4));



    ToIntFunction<Employee> playerExp = employee -> employee.experience();

    int totalYearsOfExperience = listOfEmployee.stream()
                                .mapToInt(playerExp)
                                .sum();


     System.out.println("Total years of experience: " + totalYearsOfExperience);



  }
}
```

Note : IntStream interface has provided a predfined abstract method
     sum()

      int sum(); //Available IntStream interface

--Predefined Functional interface :
----

1) Predicate<T>	boolean test(T x)
2) Consumer<T>	void accept(T x)
3) Function<T,R>	R apply(T x)
4) Supplier<T>	T get()
5) BiPredicate<T,U>	boolean test(T x, U u)
6) BiConsumer<T,U>	void accept(T x, U u)
7) BiFunction<T,U,R>	R apply(T x, U u)

8) UnaryOpartor<T>   T apply(T x)
9) BinaryOperator<T>   T apply(T x, T y);
10) ToIntFunction<T>   int applyAsInt(T x)
11) ToLongFunction<T>   long applyAsLong(T x);
12) ToDoubleFunction<T>   double applyAsDouble(T x);
----
public Optional<T> min(Comparator<? super T> comparator)

It is a predefined method of Stream interface ,It is used to find the minimum element of the stream according to the provided Comparator.

This method is useful when we need to find out the smallest element in a stream, based on a specific comparison criteria using Comparator.

```
package com.ravi.advanced.min_demo;

import java.util.Arrays;
import java.util.List;
import java.util.Optional;
import java.util.OptionalInt;
import java.util.stream.IntStream;

public class MinDemo1
{
   public static void main(String[] args)
   {
       List<Integer> listOfNumbers = Arrays.asList(10, 20, 5, 40, 25, 1);

       Optional<Integer> min = listOfNumbers.stream().min((i1,i2)-> i1.compareTo(i2));

      min.ifPresent(System.out::println);


      int arr[] = {1,7,9, -8};
      IntStream stream = Arrays.stream(arr);
      OptionalInt min2 = stream.min();
      min2.ifPresent(System.out::println);

      }
}


package com.ravi.advanced.min_demo;
```

```
import java.util.Comparator;
import java.util.LinkedList;
import java.util.Optional;
import java.util.stream.Stream;

//Finding the minimum age of Employee

record Employee(Integer age, String name)
{

}


public class MinDemo2
{
        public static void main(String[] args)
        {
                Employee e1 = new Employee(23, "Scott");
                Employee e2 = new Employee(29, "Smith");
                Employee e3 = new Employee(21, "John");
                Employee e4 = new Employee(18, "Martin");




                Stream<Employee> streamOfEmployee = Stream.of(e1,e2,e3,e4);

                Optional<Employee> min =
streamOfEmployee.min(Comparator.comparingInt(Employee::age));



                min.ifPresent(System.out::println);
        }

}

----------------------------------------------------------------

package com.ravi.advanced.min_demo;

import java.util.Comparator;
import java.util.List;
import java.util.Optional;

//Finding the Cheapest Product

record Product(Integer productId, String productName, Double productPrice)
{

}


public class MinDemo3 {

	public static void main(String[] args)
	{
		var p1 = new Product(111, "Camera", 45000D);
		var p2 = new Product(222, "Watch", 23000D);
		var p3 = new Product(333, "HeadPhone", 2000D);
		var p4 = new Product(444, "Keyboard", 500D);

		List<Product> listOfProduct = List.of(p1,p2,p3,p4);

		Optional<Product> min = listOfProduct.stream().
		      min(Comparator.comparingDouble(Product::productPrice));


		min.ifPresent(System.out::println);

	}

}
```

06-02-2025
----
public Optional<T> max(Comparator<? super T> comparator)

It is a predefined method of Stream interface ,It is used to find the maximum element of the stream according to the provided Comparator.

This method is useful when we need to find out the largest element in a stream, based on a specific comparison criteria using Comparator.

```
package com.ravi.advanced.max_demo;
```

```
import java.util.Comparator;
import java.util.List;
import java.util.Optional;

public class MaxDemo1 {

	public static void main(String[] args)
	{
		List<String> listOfFruits =
List.of("Apple","Orange","Mango","Grapes","Pomogranate");

		Optional<String> max =
listOfFruits.stream().max(Comparator.comparingInt(String::length));

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

}

package com.ravi.advanced.max_demo;

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

//Finding the Employee with the Highest Salary

record Employee(Integer employeeId, String employeeName, Double employeeSalary)
{
}

public class MaxDemo2
{
	public static void main(String[] args)
	{
		Employee e1 = new Employee(111, "Aman", 23000D);
		Employee e2 = new Employee(222, "Ramesh", 24000D);
		Employee e3 = new Employee(333, "Suraj", 25000D);
		Employee e4 = new Employee(444, "Raj", 26000D);
		Employee e5 = new Employee(555, "Scott", 46000D);

		Stream<Employee> streamOfEmployees = Stream.of(e1,e2,e3,e4,e5);
		Optional<Employee> max = 
streamOfEmployees.max(Comparator.comparingDouble(Employee::employeeSalary));

		if(max.isPresent())
		{
			System.out.println("Employee Having Maximum Salary is :"+max.get());
		}
		else
		{
			System.out.println("No record Available");
		}

	}

}
```
----
public Optional<T> findAny() :

It is a predefined method of Stream interface ,It is used to return an Optional which describes some element of the stream, or an empty Optional if the stream is empty.

It's useful when we need any element from the stream but don't care which one (Randomly pick an element).

It is better to use parallelStream() method for better output.

```
package com.ravi.advanced.find_any;

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

public class FindAnyDemo1
{
	public static void main(String[] args)
	{
		List<String> listOfNames = List.of("Raj", "Rahul", "Ankit");

		Optional<String> findAny = listOfNames.parallelStream().findAny();

		findAny.ifPresent(System.out::println);

	}
}

package com.ravi.advanced.find_any;

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

public class FindAnyDemo2 {

        public static void main(String[] args)
        {
      List<String> listOfName = List.of("Sachin", "Ankit", "Aman", "Rahul", "Ravi");

      Optional<String> anyElement = listOfName.parallelStream().filter(s ->
s.startsWith("R")).findAny();

      anyElement.ifPresent(System.out::println);
    }

}
```

----
public Optional<T> findFirst()

It is a predefined method of Stream interface ,It is used to return
an Optional which describes the first element of the stream, or an empty Optional if the stream is empty.

```
package com.ravi.advanced.find_first;

import java.util.stream.Stream;

public class FindFirstDemo1
{
  public static void main(String[] args)
  {
        Stream<String> playerName = Stream.of("Virat", "Rohit", "Raj", "Bumrah",
"Arshdeep");


        playerName.findFirst().ifPresent(System.out::println);
  }
}
```

Differences between findAny() and findFirst()
----
findFirst(): Always returns the first element of the stream, which is particularly useful for ordered streams.

findAny(): Returns any element from the stream, and is typically used in unordered streams where the order is not required. It may return elements faster because it does not have to maintain the order.

Note : Collection interface has provided parallelStream() method for fast execution [It uses multiple threads]
----
public boolean allMatch(Predicate<? super T> predicate)

It is a predefined method of Stream interface , It is used to check if all elements of the stream match a given predicate. This method is useful when we need to verify that every element in a stream satisfies a specific condition by using Predicate.

```
package com.ravi.advanced.match;

import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;
import java.util.stream.Stream;

public class AllMatchDemo1 {

        public static void main(String[] args)
        {
                Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);

      boolean allPositive = stream.allMatch(n -> n > 0);
      System.out.println("All elements are positive: " + allPositive);

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

      List<Integer> numbers = Arrays.asList(2, 4, 6, 8, 10, 11);

      Predicate<Integer> isEven =  number -> number % 2 == 0;

      boolean allEven = numbers.stream().allMatch(isEven);
```

```
        if (allEven)
        {
           System.out.println("All numbers are even.");
        }
        else
        {
           System.out.println("Not all numbers are even.");
        }

           }

}
```

----

public boolean anyMatch(Predicate<? super T> predicate)

It is a predefined method of Stream interface, It returns a boolean indicating whether any element of the stream match the given predicate. It is useful when we need to verify that at least one element in the stream that satisfies the provided condition. (Predicate)

```
package com.ravi.advanced.match;

import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;

public class AnyMatchDemo1
{
	public static void main(String[] args)
	{
		List<String> listOfName = List.of("Virat","Rohit","Bumrah","Surya");

	boolean startsWithA = listOfName.stream().anyMatch(name -> name.startsWith("A"));

		System.out.println("Any name starts with letter 'A' : " + startsWithA);


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

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

		Predicate<Integer> isEven = number -> number % 2 == 0;
```

```
                boolean anyEven = numbers.stream().anyMatch(isEven);

                if (anyEven)
                {
                   System.out.println("There is at least one even number.");
                }
                else
                {
                   System.out.println("There are no even numbers.");
                }

             }
}
```

----
public boolean noneMatch(Predicate<? super T> predicate)

It is a predefined method of Stream interface, It is used to check if no elements of the stream match a given predicate. It returns a boolean value true if no elements match the predicate, and false if at least one element matches the predicate or if the stream is empty.

```
package com.ravi.advanced.match;

import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;

public class NoneMatchDemo1
{
  public static void main(String[] args)
  {
            List<Integer> numbers = Arrays.asList(1, 3, 5, 7, 9);

      Predicate<Integer> isEven = number -> number % 2 == 0;


      boolean noneEven = numbers.stream().noneMatch(isEven);



      if (noneEven)
      {
         System.out.println("There are no even numbers.");
      }
      else
      {
         System.out.println("There is at least one even number.");
      }
}
}
```
----

public void forEach(Consumer<T> cons) :

It is a predefined method of Stream interface, the forEach operation allows us to perform an action on each element of a stream. It takes a Consumer as a parameter and executes it for each element of the stream.

```
package com.ravi.advanced;

import java.util.stream.Stream;

public class ForEachDemo1 {

        public static void main(String[] args)
        {
                Stream<Integer> streamOfNumbers = Stream.of(1,2,3,4,5,6,7,8,9,10);
                streamOfNumbers.forEach(System.out::println);
        }

}
```

R collect(Collector<? super T, A, R> collector);

It is a predefined method of Stream interface.It is used to transform the elements of a stream into a different form like collection i.e. List, Set, or Map.

The collect() method takes an argument of type Collector, which is a functional interface that specifies how to perform the collection operation. The Collectors class has provided follwing static methods

a) toList(): Collects the elements of the stream into a List.
b) toSet(): Collects the elements of the stream into a Set.
c) toMap(): Collects the elements of the stream into a Map.
d) joining(charSequenec ch): Concatenates the elements of the stream into a String.
e) groupingBy(Function<K keyMapper, V value<apper>): Groups the elements of the stream by a classifier function.

f) partitioningBy(Predicate<T> p): Partitions the elements of the stream into two groups based on a predicate.

```
package com.ravi.advanced.collect;

//Join the elements by using joining() methods of Collectors class
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

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);

   }
}
```

Map<K, List<T>> Collectors.groupingBy(Function<T,R> classifier) :

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.

```
package com.ravi.advanced.collect;

//Group the city name according to length of the city name
import java.util.*;
import java.util.stream.Collectors;

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

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

        collect.forEach((len, cities)-> System.out.println(len+ " :"+cities));


    }
}
```

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

It is accepting keyMapper and valueMapper as a Function functional interface.

Internally it returns HashMap object so, output is unpredicatable.

```
package com.ravi.advanced.collect;

//Print the length of the country
import java.util.*;
import java.util.stream.Collectors;

public class CollectDemo3
{
    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(
                        countryName -> countryName,
                        countryName -> countryName.length()
                ));

        map.forEach((key, value) ->
        {
            System.out.println(key + ": " + value);
        });
    }
}
```

```
package com.ravi.advanced.collect;
```

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

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));

	}
```

}

Map<Boolean, List<T>> Collectors.partitioningBy(Predicate<T> p)

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.

partitioningBy() method takes two parameter which is overloaded method as shown below

partitioningBy(Predicate<T> p , Collector<T> c)

```
package com.ravi.advanced.collect;

//Partition the given number based on the Predicate

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

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>> partitioned = numbers.stream()
         .collect(Collectors.partitioningBy(n -> n % 2 == 1));

      System.out.println(partitioned);
    }
}
```

```
package com.ravi.advanced.collect;

//Partition the given number based on the Predicate and convert to Set
```

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

public class CollectDemo6
{
    public static void main(String[] args)
    {
        List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1, 3);

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

        System.out.println(partitioned);

    }
}
------------------------------------------------------------

package com.ravi.advanced.collect;

//Count how many elements are partition based on given predicate

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

public class CollectDemo7
{
    public static void main(String[] args)
    {
        List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11);

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

        System.out.println(partitioned);
    }
}
```

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.

```
package com.ravi.advanced.reduce;

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

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

        Stream<Integer> integerStream = Stream.of(1, 2, 3, 4, 5);
        Optional<Integer> reduce = integerStream.reduce(Integer::sum);

        reduce.ifPresent(System.out::println);




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

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


    }
```

```
}
------------------------------------------------------------
package com.ravi.advanced.reduce;

//Finding the maximum number

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

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);
   }
}
------------------------------------------------------------
package com.ravi.advanced.reduce;

//Finding the multiplication of all numbers

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

public class ReduceDemo3
{
   public static void main(String[] args) {
      List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

      Optional<Integer> product = numbers.stream()
                                 .reduce((a, b) -> a * b);

      product.ifPresent(System.out::println);

      System.out.println("=========================");
      Integer reduce = numbers.stream().reduce(1,(a,b)-> a*b);
      System.out.println(reduce);
```

```
    }
}
------------------------------------------------------------
package com.ravi.advanced.reduce;
import java.util.Arrays;
import java.util.List;
import java.util.Optional;

public class ReduceDemo4
{
    public static void main(String[] args) {
        List<String> words = Arrays.asList("Java", "is", "Best");

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

        concatenated.ifPresent(System.out::println);
      }
}
------------------------------------------------------------
package com.ravi.advanced.reduce;

//Finding the total sale amount

import java.util.stream.Stream;

record Sale(String item, Double amount)
{

}

public class ReduceDemo5
{
            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)
                    );

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

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



            }

}
```
==