Stream API Notes :
---
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);
    }
}
```

---

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.

Comparator intrerface methods :

Comparator.comparingInt(ToIntFunction<T>)

Comparator.comparingDouble(ToDoubleFunction<T>)

Comparator.comparingLong(ToLongFunction<T>)

All these methods return type is : 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);

    }

}
```

---

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 It is 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);
   }

}
```