```
class Sample
{
	private String s;
	Sample(String s){
		this.s = s;
		System.out.println("Constructor executed...."+s);
	}
}
@FunctionalInterface
interface Interf
{
	public Sample get(String s);
}
public class Test {
	public static void main(String[] args){
			Interf i  = s -> new Sample(s);
			i.get("from lambda expression...");

			System.out.println();

			//constructor reference
			Interf i1 = Sample::new;
			i1.get("from constructor reference....");
	}
}
```

Example of Method reference

```
@FunctionalInterface
interface Interf
{
	public void m1(int i);
}
public class Test {

	//logic coded by other developer
	public void m2(int i){
		System.out.println(i*i);
		System.out.println("logic coming from method reference...");
	}

	public static void main(String[] args){
			Interf i  = x-> System.out.println(x);
			i.m1(10);

			System.out.println();

			//method reference(binding the body of m2() to abstract method
m1)
			Interf i1 = new Test()::m2;
			i1.m1(20);

	}
}
```

Eg:To demonstrate the usage of forEach() to print the elements of Arraylist

```
import java.util.*;
import java.util.function.*;
```

```
// public void forEach(java.util.function.Consumer<? super E>);
// public abstract void accept(T t)

class MyConsumer implements Consumer<String>
{
      @Override
      public void accept(String name){
            System.out.println("accept method got called...");
            System.out.println(name);
      }
}
public class Test {
      public static void main(String[] args){

                  ArrayList<String> names = new ArrayList<String>();
                  names.add("sachin");
                  names.add("dhoni");
                  names.add("kohli");
                  names.add("dravid");

                  //Traditional approach
                  Consumer<String> consumer = new MyConsumer();
                  names.forEach(consumer);
                  System.out.println();

                  //lambda expression
                  names.forEach(name->System.out.println(name));
                  System.out.println();

                  //method reference
                  names.forEach(System.out::println);
      }
}
```

Stream API
==========
Stream -----> Channel through which there is a free flow movement of data.

Streams
To process objects of the collection, in 1.8 version Streams concept introduced.

What is the differences between Java.util.streams and Java.io streams?
java.util streams meant for processing objects from the collection. Ie, it represents a stream of objects from the collection
but Java.io streams meant for processing binary and character data with respect to file.
i.e it represents stream of binary data or character data from the file .
hence Java.io streams and Java.util streams both are different.

What is the difference between collection and stream?
 => If we want to represent a group of individual objects as a single entity then We should go for collection.
 => If we want to process a group of objects from the collection then we should go for streams.
 => We can create a stream object to the collection by using stream() method of Collection interface. stream()
       method is a default method added to the Collection in 1.8 version.

```
import java.util.*;
import java.util.stream.*;
public class Test {
	public static void main(String[] args){

		ArrayList<Integer> al = new ArrayList<Integer>();
			al.add(0);
			al.add(5);
			al.add(10);
			al.add(15);
			al.add(20);
			al.add(25);

			System.out.println(al);//[0,5,10,15,20,25]

			//till jdk1.7v
			ArrayList<Integer> evenList = new ArrayList<Integer>();
			for ( Integer i1: al )
				if (i1%2==0)
					evenList.add(i1);
			System.out.println(evenList);//[0,10,20]

			//From JDK1.8V we use Streams
			//1. Configuration ===> al.stream()
			//2. Processing       ===> filter(i->i
%2==0).collect(Collectors.toList())
			List<Integer> streamList=al.stream().filter(i->i
%2==0).collect(Collectors.toList());
			System.out.println(streamList);
			streamList.forEach(System.out :: println);

	}
}

eg#2.
import java.util.*;
import java.util.stream.*;
public class Test {
	public static void main(String[] args){
		ArrayList<Integer> al = new ArrayList<Integer>();
			al.add(0);
			al.add(5);
			al.add(10);
			al.add(15);
			al.add(20);
			al.add(25);

			System.out.println(al);

			// till JDK1.7V
			ArrayList<Integer> doubleList = new ArrayList<Integer>();
			for ( Integer i1: al )
				doubleList.add(i1*2);

			System.out.println(doubleList);


			// from JDK1.8V
			// map-> for every object, if a new object has to be created then
```

```
go for Map
                  List<Integer> streamList = al.stream().map(obj-
>obj*2).collect(Collectors.toList());
                  System.out.println(streamList);
                  streamList.forEach(i-> System.out.println(i));

                  System.out.println();

                  streamList.forEach(System.out::println);
      }
}
```
=> Stream is an interface present in java.util.stream. Once we got the stream, by using that we can process objects of that collection.

We can process the objects in the following 2 phases

1.Configuration

2.Processing

1) Configuration:

We can configure either by using filter mechanism or by using map mechanism.

Filtering:

We can configure a filter to filter elements from the collection based on some boolean condition by using

filter()method of Stream interface.

```
                  public Stream filter(Predicate<T> t)
```
here (Predicate<T > t) can be a boolean valued function/lambda expression

Ex:
```
Stream s = c.stream();
Stream s1 = s.filter(i -> i%2==0);
```
Hence to filter elements of collection based on some Boolean condition we should go for filter() method.

Mapping:

If we want to create a separate new object, for every object present in the collection based on our requirement then we should go for

map() method of Stream interface.

```
                  public Stream map (Function f);
```
It can be lambda expression also

Ex:
```
 Stream s = c.stream();
 Stream s1 = s.map(i-> i+10);
```
Once we performed configuration we can process objects by using several methods.

2) Processing

processing by collect() method

Processing by count()method

Processing by sorted()method

Processing by min() and max() methods

forEach() method

toArray() method

Stream.of()method

eg#1.
```
import java.util.*;
import java.util.stream.*;
public class Test {
      public static void main(String[] args){
```

```
            ArrayList<String>names = new ArrayList<String>();
            names.add("sachin");
            names.add("saurav");
            names.add("dhoni");
            names.add("dravid");
            names.add("kohli");
            names.add("raina");

            System.out.println(names);

            List<String> reslut = names.stream().filter(name-
>name.length()>5).collect(Collectors.toList());
            System.out.println(reslut.size());

            long count= names.stream().filter(name->name.length()>5).count();
            System.out.println("The no of objects whose string length > 5
is ::"+count);

      }
}

import java.util.*;
import java.util.stream.*;

//Comparable(Predefined API for natural sorting order) -> compareTo(Object obj)
//Comparator(for userdefined class for customized sorting order)->
compare(Obj1,Obj2)

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

            ArrayList<Integer> al =new ArrayList<Integer>();
                  al.add(10);
                  al.add(0);
                  al.add(15);
                  al.add(5);
                  al.add(20);
            System.out.println("Before sorting :: "+al);

            //using stream api
            List<Integer> resultList=
al.stream().sorted().collect(Collectors.toList());
            System.out.println("After sorting  :: "+resultList);

            List<Integer> customizedResult = al.stream().sorted((i1,i2)-
>i2.compareTo(i1)).collect(Collectors.toList());
            System.out.println("After sorting  :: "+customizedResult);

      }
}
import java.util.*;
import java.util.stream.*;

//Comparable(Predefined API for natural sorting order) -> compareTo(Object obj)
//Comparator(for userdefined class for customized sorting order)->
compare(Obj1,Obj2)

public class Test {
```

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

		ArrayList<Integer> al =new ArrayList<Integer>();
			al.add(10);
			al.add(0);
			al.add(15);
			al.add(5);
			al.add(20);
		System.out.println("Array List is ::"+al);

		Object[] objArr = al.stream().toArray();
		for(Object obj: objArr)
			System.out.println(obj);

		System.out.println();

		Integer[] objArr1 = al.stream().toArray(Integer[]::new);
		for(Integer obj1: objArr1)
			System.out.println(obj1);


	}
}

eg
import java.util.*;
import java.util.stream.*;
public class Test {
	public static void main(String[] args){

		//Stream API ===> Collections(group of objects)

		Stream s= Stream.of(9,99,999,9999,99999);
		s.forEach(System.out::println);

		System.out.println();

		Double[] d = {10.0,10.1,10.2,10.3,10.4};
		Stream s1= Stream.of(d);
		s1.forEach(System.out::println);

	}
}
collect()
=======
This method collects the elements from the stream and adding to the specified to
the collection indicated (specified) by argument.


eg#1.
import java.util.*;
import java.util.stream.*;
public class Test {
	public static void main(String[] args) {

			ArrayList<String> names = new ArrayList<String>();
			names.add("sachin");
			names.add("saurav");
			names.add("dhoni");
```

```
                  names.add("yuvi");


                  System.out.println(names);//[sachin,saurav,dhoni,yuvi]


      //Predicate(I)

//    public abstract boolean test(T);
                  List<String> result=names.stream().filter(name->name.length()>5).
                        collect(Collectors.toList());
                  System.out.println(result);


//Function(I)<T,R>

      //    public abstract R apply(T);
                  List<String> mapResult = names.stream().map(name->
name.toUpperCase()).
                                    collect(Collectors.toList());
                  System.out.println(mapResult);

      }
}


count()
======
This method returns number of elements present in the stream.
      public long count()

import java.util.*;
import java.util.stream.*;
public class Test {
      public static void main(String[] args) {

                  ArrayList<String> names = new ArrayList<String>();
                  names.add("sachin");
                  names.add("saurav");
                  names.add("dhoni");
                  names.add("yuvi");


                  System.out.println(names);//[sachin,saurav,dhoni,yuvi]

                  long count = names.stream().filter(name-
>name.length()>5).count();
                  System.out.println(count);
      }
}

III.Processing by sorted()method
     If we sort the elements present inside stream then we should go for sorted()
method.
     The sorting can either default natural sorting order or customized sorting
order specified by comparator.
            sorted()- default natural sorting order
            sorted(Comparator c)-customized sorting order.
```

```
import java.util.*;
import java.util.stream.*;

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

                  ArrayList<Integer> al = new ArrayList<Integer>();
                  al.add(10);
                  al.add(20);
                  al.add(0);
                  al.add(5);
                  al.add(25);
                  al.add(15);
                  System.out.println(al);

                  List<Integer> result =
al.stream().sorted().collect(Collectors.toList());
                  System.out.println(result);

                  List<Integer> customizedResult=al.stream().sorted((i1,i2) ->-
i1.compareTo(i2)).collect(Collectors.toList());
                  System.out.println(customizedResult);
      }
}
```

IV.Processing by min() and max() methods

```
            min(Comparator c)
                        returns minimum value according to specified comparator.
            max(Comparator c)
                        returns maximum value according to specified comparator.
```

```
import java.util.*;
import java.util.stream.*;

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

                  ArrayList<Integer> al = new ArrayList<Integer>();
                  al.add(10);
                  al.add(20);
                  al.add(0);
                  al.add(5);
                  al.add(25);
                  al.add(15);
                  System.out.println(al);

                  Integer minValue = al.stream().min((i1,i2)->
i1.compareTo(i2)).get();
                  System.out.println(minValue);

                  Integer maxValue = al.stream().max((i1,i2)->
i1.compareTo(i2)).get();
                  System.out.println(maxValue);
      }
}
```

V.forEach() method

This method will not return anything.

This method will take lambda expression as argument and apply that lambda

expression for each element present in the stream.

```
import java.util.*;
import java.util.stream.*;

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

                  ArrayList<String>names = new ArrayList<String>();
                  names.add("AAA");
                  names.add("BBB");
                  names.add("CCC");
                  names.add("DDD");

                  names.stream().forEach(name -> System.out.println(name));
                  names.stream().forEach(System.out::println);
      }
}
```

VI.
toArray() method
We can use toArray() method to copy elements present in the stream into specified array

```
import java.util.*;
import java.util.stream.*;

public class Test {
      public static void main(String[] args) {
                  ArrayList<Integer> al =new ArrayList<Integer>();
                  al.add(0);
                  al.add(10);
                  al.add(5);
                  al.add(20);
                  al.add(15);
                  System.out.println(al);

                  Integer[] array = al.stream().toArray(Integer[]::new);
                  for (Integer element : array)
                        System.out.println(element);
      }
}
```

VII.Stream.of()method
We can also apply a stream for group of values and for arrays.

Ex:

```
Stream s=Stream.of(99,999,9999,99999);
s.forEach(System.out:: println);

Double[] d={10.0,10.1,10.2,10.3};
Stream s1=Stream.of(d);
s1.forEach(System.out :: println);
```