Sorting the List using LE:
==========

```
package com.masai;

import java.util.ArrayList;
import java.util.Collections;import java.util.Comparator;
import java.util.List;

public class Demo {


	public static void main(String[] args) {


		List<Student> students= new ArrayList<>();

		students.add(new Student(1, "N1", 500));
		students.add(new Student(4, "N2", 400));
		students.add(new Student(5, "N3", 600));
		students.add(new Student(3, "N4", 800));
		students.add(new Student(2, "N5", 700));


		//using Annonymous inner class
		Collections.sort(students, new Comparator<Student>() {

			@Override
			public int compare(Student s1, Student s2) {

				return s1.getMarks() > s2.getMarks() ? +1 : -1;
			}

		} );

		//using LE

		Collections.sort(students,(s1,s2) -> s1.getMarks() > s2.getMarks() ? +1 : -1);




	}
```

```
}
```

LE as a method parameter:
==========

```
package com.masai;

public class Demo {


	void fun1(Intr i1) {

		if(i1 != null) {
		System.out.println("inside fun1 of Demo");

		i1.printSomeThing("Amit");

		}else
			System.out.println("Null is not allowed");
	}

	public static void main(String[] args) {


		Demo d1 = new Demo();



		//d1.fun1(new IntrImpl());
		//d1.fun1(null);

		//Intr i1 = s -> System.out.println("Welcome Using LE "+s);
		//d1.fun1(i1);

		d1.fun1(s -> System.out.println("Welcome Using LE "+s));

	}

}
```

LE as a return type:
==========

```
package com.masai;

public class Demo {


	Intr fun1() {
		System.out.println("inside fun1 of Demo");

		//return null;


		//return new IntrImpl();

		//Intr i1 = new IntrImpl();
		//return i1;

//		Intr i1 = s -> System.out.println("Welcome Using LE "+s);
//		return i1;

		return s-> System.out.println("Welcome to LE :"+s);

	}


	public static void main(String[] args) {

		Demo d1 = new Demo();

		Intr i2=  d1.fun1();

		i2.printSomeThing("Ravi ");




	}

}
```

Some of the FI introduced in Java 1.8 to perform functional programming :
==========

--these interfaces belongs to "java.util.function" package.

1.Predicate(I)

2.Consumer(I)

3.Supplier(I)

4.Function(I)

--these all interfaces are generic type:

1. Predicate<T> :
==========

this interface has only one abstract method :

public Predicate<T>

public boolean test(T t);

}

---this test() method checks wheter supplied object is satisfying a condition or not.

ex: - test a Student object whether his marks is less than 800.

ex1:

MyPredicate.java:

```
package com.masai;

import java.util.function.Predicate;

public class MyPredicate implements Predicate<Integer>{
```

```
	@Override
	public boolean test(Integer i) {

		return i >= 0 ;

	}



}
```

Demo.java:

```
----------

package com.masai;

import java.util.function.Predicate;

public class Demo {


	public static void main(String[] args) {


		Predicate<Integer> p1 = new MyPredicate();
		System.out.println(p1.test(-10));

		Predicate<Integer> p2= i ->  i >= 0;
		System.out.println(p2.test(-10));


	}
}
```

ex2:

MyPredicate.java:

```
----------------------

package com.masai;
```

```
import java.util.function.Predicate;

public class MyPredicate implements Predicate<Student>{

	@Override
	public boolean test(Student s) {

//		if(s.getMarks() > 700)
//			return true;
//		else
//			return false;



		return s.getMarks() > 700 ;

	}

}
```

Demo.java:

```
package com.masai;

import java.util.function.Predicate;

public class Demo {


	public static void main(String[] args) {


		Predicate<Student> p1= new MyPredicate();
		System.out.println(p1.test(new Student(10, "N1", 600)));


		Predicate<Student> p2 = s -> s.getMarks() > 700;
		System.out.println(p2.test(new Student(10, "N1", 800)));


	}
}
```

--From java 1.8 onwards inside the Collection interface one new method is added called:

public boolean removeIf(Predicate filter);

--based on the condition of Predicate , this method will remove/filter the elementts from the any type of Collection object.

example:

Demo.java:

```
package com.masai;

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

public class Demo {


	public static void main(String[] args) {

		List<Student> students = new ArrayList<>();

		students.add(new Student(10, "N1", 780));
		students.add(new Student(12, "N2", 480));
		students.add(new Student(13, "N3", 380));
		students.add(new Student(14, "N4", 880));
		students.add(new Student(15, "N5", 680));
		students.add(new Student(16, "N6", 720));


		//students.removeIf(new MyPredicate());

		students.removeIf(s -> s.getMarks() < 500);

		for(Student s: students) {
			System.out.println(s);
		}
```

```
	}
}
```

2.java.util.function.Consumer<T>:
==========

public abstract void accept(T t);

--this method accept the object of generic type and it does not return anything.

ex1:

MyConsumer.java:

```
package com.masai;

import java.util.function.Consumer;

public class MyConsumer implements Consumer<String>{

	@Override
	public void accept(String s) {

		System.out.println("Welcome :"+s);

	}

}
```

Demo.java:

```
package com.masai;

import java.util.function.Consumer;

public class Demo {
```

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

		Consumer<String> c1 = new MyConsumer();
		c1.accept("Ravi");

		Consumer<String> c2 = s -> System.out.println("Welcome Using LE "+s);
		c2.accept("Amit");

	}
}
```

example2:

MyConsumer.java:

```
package com.masai;

import java.util.function.Consumer;

public class MyConsumer implements Consumer<Student>{

	@Override
	public void accept(Student s) {

		System.out.println("Roll is :"+s.getRoll());
		System.out.println("Name is :"+s.getName());
		System.out.println("Marks is :"+s.getMarks());
	}


}
```

Demo.java:

```
package com.masai;

import java.util.function.Consumer;

public class Demo {
```

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

		Consumer<Student> c1 = new MyConsumer();
		c1.accept(new Student(10, "Ram", 800));


		Consumer<Student> c2= s -> {

			System.out.println("Roll is :"+s.getRoll());
			System.out.println("Name is :"+s.getName());
			System.out.println("Marks is :"+s.getMarks());

		};

		c2.accept(new Student(10, "Ram", 800));

	}
}
```

forEach method:
==========

public void forEach(Consumer action); //action for each element of a collection

--this method we can call from any Collection class object.

example1:

```
package com.masai;

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

public class Demo {


	public static void main(String[] args) {


		List<String> names=
Arrays.asList("Amit","Ravi","mohit","aanand","vinay");
```

```
                names.forEach(name -> System.out.println(name.toUpperCase()));

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

                names.forEach(System.out::println);


        }
}
```

example2:

Demo.java:
==========

```
package com.masai;

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

public class Demo {


        public static void main(String[] args) {


                List<Student> students = new ArrayList<>();

                students.add(new Student(10, "N1", 780));
                students.add(new Student(12, "N2", 480));
                students.add(new Student(13, "N3", 380));
                students.add(new Student(14, "N4", 880));
                students.add(new Student(15, "N5", 680));
                students.add(new Student(16, "N6", 720));


                students.forEach(s -> {

                        System.out.println("Roll is :"+s.getRoll());
                        System.out.println("Name is :"+s.getName());
```

```
                System.out.println("Marks is :"+s.getMarks());

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

            });


        }
}
```

3. java.util.function.Supplier<T>:
==========================

```
public abstract T get();
```

example:

MySupplier.java:

```
package com.masai;

import java.util.function.Supplier;

public class MySupplier implements Supplier<String>{

    @Override
    public String get() {


        return "this message from the external class";


    }

}
```

Demo.java:

```
package com.masai;

import java.util.function.Supplier;
```

```
public class Demo {


	public static int getANumber() {

		return 1000;
	}


	public static void main(String[] args) {

	Supplier<String> s1 = new MySupplier();

	String str= s1.get();

	System.out.println(str);


	Supplier<String> s2 = () -> "This message from LE";
	System.out.println(s2.get());

	Supplier<Student> s3 = () -> new Student(10, "N1", 800);
	System.out.println(s3.get());

	Supplier<Integer> s4 = Demo::getANumber;
	System.out.println(s4.get());



	}
}
```

4. java.util.function.Function<T,R>:
==========

public R apply(T t);

example:

--Getting a Student object and returning the result of that student if marks > 500 then return Pass otherwise return Fail.

MyFunction.java:

```
--------------------

package com.masai;

import java.util.function.Function;

public class MyFunction implements Function<Student, String>{

	@Override
	public String apply(Student s) {

//		if(s.getMarks() > 500)
//			return "Pass";
//
//		else
//			return "Fail";

		return s.getMarks() > 500 ? "Pass": "Fail";



	}

}
```

Demo.java:

```
---------------

package com.masai;

import java.util.function.Function;

public class Demo {



	public static void main(String[] args) {


		Function<Student, String> f1 = new MyFunction();
```

```
                String result= f1.apply(new Student(10, "Ravi", 450));

                System.out.println(result);


                Function<Student, String> f2 = s -> s.getMarks() > 500 ? "Pass" : "Fail";

                System.out.println(f2.apply(new Student(20, "Amit", 900)));

        }
}
```

Stream API:
=========

--this api is also introduced in java 1.8

--this api belongs to "java.util.stream" package.

--this api is different from IO-Stream, this IO-Stream api belongs to java.io package.

--this java.util.stream package contains some library classes and interfaces by using which
we can perform functional style of programming on a group of objects (Collection objects)

java.util.stream package classes represents flow of data in the form of objects.

*****This API has one main interface:**

java.util.stream.Stream(I)

Note: object of this Stream interface represents flow/sequence of objects from a source like collection objects.

Features of Stream:
==========

1. stream does not store the elements, it only represents the elements in a sequence.

ex: wire does not hold/store the electricity.

2.It represents only flow of objects , not the primitives.

3.operations (filtering/transforming,etc) performed on the stream object does not modify its source.

ex: filtering a stream obtained from a source (collection) produces a new stream with the filtered elements rather than removing the elements from the source collection. (filter chaining)

4. with the help of stream object we can perform various usefull operation on the collection data in functional style, like filterning some elements , printing some elements, transforming some elements etc.

Collection interface provides 2 methods to get a Stream object.

1. Stream<T> stream();

2. Stream<T> parrellalStream(); // this stream obj is used in multithreaded application.

Methods of the Stream(I) interface:
==========

there are 2 types of methods in the Stream interface:

1. Intermediate method

2. terminal method

1. Intermediate method: these methods returns a new Stream object, instead of final output.

--these methods never gives the final result.

some of the commonly used intermediate methods are:

map(), filter() method.

2. terminal method: stream object returns the final output only when terminal methdo is called on the stream object.

these methods consumes that stream object, and after that we can not re-use

that stream object again.

Note: If we try to use a consumed stream object once again, then it will throw an exception.

some of the commonly used terminal methods are:

forEach(Consumer c);
collect()
min()
max()
count()
get()
anyMatch()
AllMatch()