examples of Stream:
================

ex1:

Demo.java:

```
package com.masai;

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

public class Demo {


	public static void main(String[] args) {


			List<String> list= Arrays.asList("one","two","three","four");


			Stream<String> str1= list.stream();

			str1.forEach(s -> System.out.println(s.toUpperCase()));//ternminal
method.

			str1.forEach(s -> System.out.println(s)); //exception

	}
}
```

filter() method:

--it is one of the intermediate method.

--this method takes a Predicate obj as an argument, and filter the stream based on the Predicate condition, and returns the filtered elements in another Stream obj.

```
package com.masai;
```

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

public class Demo {


	public static void main(String[] args) {


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


		students.add(new Student(10, "N1",500));
		students.add(new Student(12, "N2",400));
		students.add(new Student(13, "N3",420));
		students.add(new Student(14, "N4",440));
		students.add(new Student(15, "N5",600));
		students.add(new Student(16, "N6",380));


		//from the above list get another list of students whose marks is less than
500.

//		Stream<Student> str1= students.stream();
//
//
//		Stream<Student> str2= str1.filter(s -> s.getMarks() < 500);
//
//		str2.forEach(s -> System.out.println(s));

		students.stream()
				.filter(s -> s.getMarks() < 500)
				.forEach(s -> System.out.println(s));


	}
}
```

--creating another list based on Filtered elements instead of printing them on the console.

ex2:

Demo.java:

```
package com.masai;

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

public class Demo {


	public static void main(String[] args) {


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


		students.add(new Student(10, "N1",500));
		students.add(new Student(12, "N2",400));
		students.add(new Student(13, "N3",420));
		students.add(new Student(14, "N4",440));
		students.add(new Student(15, "N5",600));
		students.add(new Student(16, "N6",380));


		//from the above list get another list of students whose marks is less than
500.

//		Stream<Student> str1= students.stream();
//
//		Stream<Student> str2= str1.filter(s -> s.getMarks() < 500);
//
//		List<Student> filteredList= str2.collect(Collectors.toList());
//

		List<Student> filteredList= students
						.stream()
```

```
                                                  .filter(s -> s.getMarks() < 500)
                                                  .collect(Collectors.toList());


                    filteredList.forEach(s -> System.out.println(s));


          }
}
```

map() method:
=============

--it is also a intermediate method.

--this method is used to transform the object.

--this method takes java.util.function.Function(I) object as an argument and map/transform the elements to a new element and returns teh mapped element in another stream.

ex:
Demo.java:

```
package com.masai;

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

public class Demo {


          public static void main(String[] args) {


                    List<Student> students= new ArrayList<>();
                    students.add(new Student(10, "N1",500));
                    students.add(new Student(12, "N2",400));
                    students.add(new Student(13, "N3",420));
                    students.add(new Student(14, "N4",440));
                    students.add(new Student(15, "N5",600));
                    students.add(new Student(16, "N6",380));

                    // from the above list give the 50 grace marks to all the students
                    //and get the another list of updated students.


                    //Stream<Student> str1= students.stream();

//                  Stream<Student> str2= str1.map(s -> {
//
//                                          Student s2 = new Student(s.getRoll(),
s.getName(), s.getMarks()+50);
//                                          return s2;
//
//                  });


                    //Stream<Student> str2= str1.map(s -> new Student(s.getRoll(),
s.getName(), s.getMarks()+50));


                    //List<Student> modifiedStudents= str2.collect(Collectors.toList());

//
//                  List<Student> modifiedStudents= students
//                                                                  .stream()
//                                                                  .map(s ->
new Student(s.getRoll(), s.getName(), s.getMarks()+50))
//
.collect(Collectors.toList());
//



          List<Student> modifiedStudents= students
                                                                  .stream()
                                                                  .filter(s ->
s.getMarks() < 500)
```

```
                                                                                    .map(s ->
new Student(s.getRoll(),s.getName(),s.getMarks()+50))

.collect(Collectors.toList());


        modifiedStudents.forEach(s -> System.out.println(s));

        }
}
```

min and max method:
=================

--these methods are also terminal methods which will takes a Comparator object, using which we can decide max and min elements.

--this min() and max() method will return the minimum and maximum object in the form of "java.util.Optional" class object.

--this class introduced in java 1.8 version, and it is basically used to avoid the NullPointerExceptiona

--to get the element from the Optional class ,we need to call get() method.

Demo.java:

```
package com.masai;

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

public class Demo {


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

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


                    students.add(new Student(10, "N1",500));
                    students.add(new Student(12, "N2",400));
                    students.add(new Student(13, "N3",420));
                    students.add(new Student(14, "N4",440));
                    students.add(new Student(15, "N5",600));
                    students.add(new Student(16, "N6",380));

Optional<Student> opt = students.stream().min( (s1,s2) -> s1.getMarks() > s2.getMarks() ?
+1 :-1);


                    Student maxStudent= opt.get();

                    System.out.println(maxStudent);


          }
}
```

count() method:
=============

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


                    students.add(new Student(10, "N1",500));
                    students.add(new Student(12, "N2",400));
                    students.add(new Student(13, "N3",420));
                    students.add(new Student(14, "N4",440));
                    students.add(new Student(15, "N5",600));
                    students.add(new Student(16, "N6",380));

                    long num= students.stream().filter(s -> s.getMarks() < 500).count();

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

allMatch(), anyMatch(), nonMatch():
================================

these methods will take the Predicate object and return the boolean value.

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


                students.add(new Student(10, "N1",500));
                students.add(new Student(12, "N2",400));
                students.add(new Student(13, "N3",420));
                students.add(new Student(14, "N4",440));
                students.add(new Student(15, "N5",600));
                students.add(new Student(16, "N6",380));

        boolean b= students.stream().anyMatch(s -> s.getMarks() < 500);


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

summingInt:
============

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


                students.add(new Student(10, "N1",500));
                students.add(new Student(12, "N2",400));
                students.add(new Student(13, "N3",420));
                students.add(new Student(14, "N4",440));
                students.add(new Student(15, "N5",600));
                students.add(new Student(16, "N6",380));


                int x= students.stream().collect(Collectors.summingInt(s -> s.getMarks()));

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

Multithreading:
=============

MultiThreading:-
================

---before learning about multithreading,we should know what nacessiated multithreading,for that we need to know about multitasking :-

consider the follwing program:-

```
class Test{

fun1()
{
--
-- data transfer 50gb (DMA ) 1 hour
--
}

fun2()
{
---
---
---
}
fun3()
{
---
---
---
}

psvm()
{

Test t=new Test();

t.fun1();
t.fun2();
t.fun3();
}
}
```

normally,if the class is compiled and jvm executes the program,then the order of excecution is that,first fun1() is called and after complete excution of fun1() control comebacks to the main() and then fun2() is called and so on...
---now let us assume that fun1() has some statment which involves data transfer,we know that data transfer is not a job of the processsor,it is a duty of a seperate individual circuit (DMA)(direct memory aceess) which functions under the control of the processor.

---since fun1() has data transfer statements,processor assigns that job to DMA.during this time the processor should remain idle.

----this made s/w developers to think that,efficiency of the processor would be increeased if processor is made to do some other useful work during this idle state.
the useful work is nothing but,executing other functions/methods present in the program

---this need lead to the concept of multitasking..

---Remember that processor is also an electric circuit and at any instance of time,it can execute only one statement.it can not execute multiple statement simulteniously...

mutitasking:-it is the concept of executing multiple tasks/functionality simulteniously.(functionality may be from same domain(same application) or from diff domain(diff applications).)

if we apply multitasking then a part of the one fun get executed then control switchs to the another fun now here also some part of second fun get executed then conrol switchs to the third fun
then again control comes back to the fun1 now it continues fun1 from where it had stopped earliar.

now when we see the output we feel that three fun executing simulteneously.

---thus,concept of multitasking come into existence to avoid the idle state of cpu...

---in mutitasking ,a part of a funtionality is executed one at a time
and it that part how many statement will be executed will be decided by scheduling.

Scheduling:-it is the process in which a specific time period is allocated to a fun where the control remains in that particular fun for that specific time period.
once time period lapsed control switches another fun with another time slice and so on.

Scheduling is supervised by the Scheduler(either OS scheduler or ThreadScheduler).

---Thread-Schedular in java is the part of jvm that decides which thread should run....

The time slice allocated will be in nano seconds.
thus by the time our eye recognize the execution of one part control switches to another part of the program.

Adv of multitasking:-
1.it is invented to avoid Idle states of the cpu.
2.make the fun get executed independetly.//mostly used becoz for a small project idle state of a cpu is not a big concern.

---animation along with form submition is a very good example of multitasking.

Multitasking is of two type:-
============================

1.process based multi tasking
2.threadbased multitasking

---java supports thread based multi tasking.

process based multitasking:-

----

concept of executing more than one program simulteniously which r present in diff location of ram is known as process based multitasking.
here processer has to maintained address of both the program since control has to shift from one part of ram to another.
it incerase overhead on processor.

--here OS scheduler will perform the scheduling.

ThreadBased multitasking:-
----

concept of executing more than one fun simul belonging to the same memory domain is known as ThreadBased multitasking:-

---here thread-scheduler will do the scheduling

Note:- the main adv of the mt is increase performance and reduce the response time of the system(reduce the idle time of CPU or proper utilization of resources)

**Application areas to apply multithreading

to develop multimedia graphics

to develop animations

to develop video games

to develop Webserver or ApplicationServer

when compared with other languages,developing multi-threded appl in java is very easy bcoz,java proviedes in-built support with rich api.

**what is thread :-
==================

--An application when it is under execution is called process.

--a thread is a part or sub process of an application.

--a thread is a seperate flow of execution that execute some functionality of a program with other part of program simulteniously.

Multithreaded application:-

--in java,every program/application has a default flow of execution,a defult thread,it is called as a main thread.if we can start another flow of execution(another thread) along with main thread simultenously then it is called a multithreaded application or program.

Implementing thread in java:-
===============================

impl thread in java is two step process:-

1.first of all we have to define a funtionality which can be executed as a thread along with the main thread(define job for a worker)

2.this fun should start as a thread.(assign job to worker)

----the signature of a fun using which we imple a thread(or what job a thread has to do) is defined in an interface by name Runnable

--this interface belongs to java.lang package.

this interface has only one method i.e

public void run();

for which we have to provide a body. (in this method we need to define the task which a thread should execute.)

after providing body we need to execute this funtionality as thread(i.e simulteniously with the other part of the program).

--there is a class by name Thread present in java.lang package,which has a method called start(),this start() method is used to execute a given functionality as a seperate thread.

--this start methods recognize the run() method of the Runnable interface and then run() method is executed as a seperate individual thread.

---here Thread class is like a worker who has to start the job individually defined by run() method..in Runnable interface...

NOTE:-with the help of run() method we define a job that has to execute as thread,and with the help of Thread class start() method we need to start the job as a seperate individual thread.

---Thread class and Runnable interface r the two structure using which we imple Thread based multitasking in java.

```
package java.lang
interface Runnable{
public abstract void run();
}

class Thread implements Runnable{

@Override
public void run(){
//Thread class internally implements Runnable interface and  overrides the run() method
with empty //implementation
}

other methods of Thread class(join, sleep, etc..)

}
```

we imple threads either of the follwing two ways:

1.By implementing Runnable Interface

2.By extending Thread class itself..

```
1.class A imple Runnable{
@Override
public void run(){
--
}
--
}
```

```
2.class A extends Thread
{
@Override
public void run(){
--
}
--
}
```

----Internally thread class imple Runnable interface and override run() method with empty implementaion...

like:-

```
class Thread implements Runnable{

@Override
public void run(){
//it is empty body overriden from Runnable interface
}


public void start(){//this is thread class own method....


}
--
--//other methods of the Thread class

}
```

Note:-wheather we extends Thread class or imple Runnable inface dirctly,we have to use run() method of the Runnable interrface .

Thread class sudo code:
==================

```
public class Thread implements Runnable {

@Override
public void run(){
//empty implementation
}

public void start(){

//registering our thread with the Thread-scheduler,
//perforing all the low level  task to start a separate flow of execution.
//60000
run();

}

class X extends Thread{

@Override
public void run(){
--
--
}


}
//other methods like (sleep, join, getName. getPriority)

}
```