**Lambda Expression:-**

Anonymous Function:-

* No name
* No return type
* No modifier

**Example -1:-**

Public void m1()

{

System.out.println(“Hello”);

}

**() ->System.out.println(“Hello”);**

**Example -2:-**

Public void m1(int a,int b)

{

System.out.println(a+b);

}

**(a,b) ->System.out.println(a+b);**

**Example-3:-**

Public void m1(int n)

{

return n\*n

}

n->n\*n;

Cases in return statement:-

n-> return n\*n; 🡺Invalid

n->{return n\*n}; 🡺Invalid

n->{return n\*n;}; 🡺valid

n->n\*n 🡺valid

(n)->{return n\*n;}; 🡺valid

Note:-

* Without curly braces we cannot use return keyword, compile will consider returned value automatically.
* Within curly braces if we want to return some value compulsory we should use return statement.

**Once we write Lambda Expression**

If you call the lambda expression by using function interface.

**Functional Interface: -** Is interface which contains single abstract method (SAM).

**Example of functional interface**

* **Runnable** 🡺**run()**
* **Callable 🡺call()**
* **Comparable🡺compaTo()**
* **ActionListner🡺actionPerfomed()**

* It contains only one abstract method but it contains any number of defaults and static method.

@FunctionalInterface

**interface** A {

**public** **abstract** **void** m1();

**default** **void** m4() {

System.***out***.println("Default m4");

}

**default** **void** m5() {

System.***out***.println("Default m5");

}

}

**class** Test2 **implements** A{

@Override

**public** **void** m1() {

System.***out***.println("m1 method implemented");

}

}

**public** **class** DriverIn{

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

Test2 t=**new** Test2();

t.m4();

}

}

**Lambda Expression with Functional Interface:-**

**Example:-1**

**package** FunLambda;

@FunctionalInterface

**interface** Interf{

**public** **abstract** **void** m1();

}

**public** **class** functionLam {

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

Interf i=()->System.***out***.println("hello lambda expression");

i.m1();

}

}

**Example:-2**

**package** FunLambda;

@FunctionalInterface

**interface** ADD{

**public** **abstract** **void** add(**int** a,**int** b);

}

**public** **class** FunLAM2 {

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

ADD add=(a,b)->System.***out***.println("The sum of number is = "+(a+b));

add.add(10, 20);

add.add(300, -200);

add.add(1000, 2000);

}

}

@FunctionalInterface

**interface** Product{

**public** **abstract** **int** squareIt(**int** n);

}

**public** **class** FuncLam3 {

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

Product i=n->(n\*n);

System.***out***.println("The squre of number is = "+i.squareIt(4));

System.***out***.println("The squre of number is = "+i.squareIt(16));

}

}

Note:-If functional interface there we can use lambda expression, if function interface not there you cannot use lambda expression, Lambda always associate with functional interface.

1. **Lambda Expression using Multithreading:-**

**Example-1**

**package** FunLambda;

**public** **class** Mutilthreading {

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

Runnable rl=()->{

**for**(**int** i=1;i<=5;i++) {

System.***out***.println("Child thread-1");

}

};

Thread t=**new** Thread(rl);

t.start();

**for**(**int** i=1;i<=5;i++) {

System.***out***.println("Main thread-1");

}

}

}

1. **Lambda Expression Using Collection:-**

**Comparator:-**

* It is a functional interface is contain only one abstract method compare() method.

**Int compare (object obj1,object obj2)**

* Return –ve if obj1 has to come **before** obj2
* Return +ve if obj1 has to come **after** obj2
* Return 0 if obj1 and obj2 **equal**

**Example-1**

**package** FunLambda;

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.Comparator;

**public** **class** Collection {

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

ArrayList<Integer> al=**new** ArrayList<Integer>();

al.add(10);

al.add(20);

al.add(1);

al.add(30);

al.add(5);

al.add(100);

al.add(30);

System.***out***.println(al);

Comparator<Integer>c=(I1,I2)->(I1<I2)?+1:(I1>I2)?-1:0;

Collections.*sort*(al,c);

System.***out***.println(al);

Collections.*reverse*(al);

System.***out***.println(al);

al.stream().forEach(System.***out***::println);

List<Integer> l= al.stream().distinct().filter(i->i%2==0).collect(Collectors.*toList*());

System.***out***.println(l);

}

}

**Example-2**

**package** FunLambda;

**import** java.util.ArrayList;

**import** java.util.Collections;

**class** Student{

String name;

**int** id;

**public** Student(String name, **int** id) {

**this**.name = name;

**this**.id = id;

}

@Override

**public** String toString() {

**return** "Student [name=" + name + ", id=" + id + "]";

}

}

**public** **class** EmployeeLam2 {

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

ArrayList<Student> al=**new** ArrayList<Student>();

al.add(**new** Student("Mirza",101352));

al.add(**new** Student("Satya",1780281));

al.add(**new** Student("Chinu",5135154));

al.add(**new** Student("Jaga",353584));

al.add(**new** Student("Soubhagya",151335));

al.add(**new** Student("Bunty",78965465));

System.***out***.println(al);

Collections.*sort*(al,(s1,s2)->(s1.name.compareTo(s2.name)));

System.***out***.println(al);

Collections.*sort*(al,(s1,s2)->(s1.id-(s2.id)));

System.***out***.println(al);

}

}

**Anonymous Inner Classes vs Lambda Expression:-**

* Anonymous Inner class **!=** Lambda Expression.
* If anonymous inner class implements an interface that contains single abstract method then only we can we can replace that anonymous inner class with lambda expression.
* If multiple abstract method is there then no use lambda expression.

**Syntax:-**

package AnonymousClass;

interface Test{

**void** m1();

**void** m2();

}

**public** **class** Annonymousmultiple {

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

Test t =**new** Test() {

@Override

**public** **void** m1() {

System.***out***.println("iam m1 method");

}

@Override

**public** **void** m2() {

System.***out***.println("I am m2 method ");

}

};

}

}

**Syntax:-**

Thread t=new Thread(){

………………

………..

};

Runnable r=new Runnable(){

………………

……………

};

**package** AnonymousClass;

**public** **class** AnnonymousClass {

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

Runnable r=()->{

**for** (**int** i = 0; i <5; i++) {

System.***out***.println("Child Thread-1");

}

};

Thread t=**new** Thread(r);

t.start();

**for** (**int** i = 0; i <5; i++) {

System.***out***.println("Main Thread-1");

}

}

}

**package** AnonymousClass;

**interface** Test{

**void** m1();

**void** m2();

}

**public** **class** Annonymousmuktiple {

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

Test t =**new** Test() {

@Override

**public** **void** m1() {

System.***out***.println("iam m1 method");

}

@Override

**public** **void** m2() {

System.***out***.println("I am m2 method ");

}

};

}

}

**Anonymous Inner Classes vs Lambda Expression:-**

* Anonymous inner class can extend a **normal** class
* Anonymous inner class can extend **abstract** class
* Anonymous inner class can implement an **interface** which contains number of abstract method.
* Lambda Expression can implement an interface which contains a **single abstract** method (**Functional** **Interface**).
* Anonymous Inner class **!=** Lambda Expression.
* Anonymous Inner class **>** Lambda Expression.

**Default Methods:-**

* Without effecting implementation classes if we want to add new method to the interface will go for default method.
* Default can override but in implementation class methods are public compulsory.
* Defaults methods multiple inheritance support.
* **(Until 1.7V)**Every method present inside interface is always : by default public abstract.
  + - **Void m1();**
    - **Public void m1();**
    - **Abstract void m1();**
    - **Public abstract void m1();**

**Related Methods:-**

* **1.8 V:** Defaults methods and static method inside interface you can use
* **1.9 V:** Private methods allows interface

**package** Deafalutmethod;

**interface** P1{

**default** **void** m1() {

System.***out***.println("P1 methods of m1 ");

}

}

**interface** P2{

**default** **void** m1() {

System.***out***.println("P2 methods of m1 ");

}

}

**public** **class** Defaltsmul **implements** P1,P2 {

@Override

**public** **void** m1() {

System.***out***.println("Own default method");

P1.**super**.m1();

P2.**super**.m1();

}

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

Defaltsmul dm=**new** Defaltsmul();

dm.m1();

}

}

**Static Methods:-**

* In interface static methods declare is not visible in implements classes it only visible in parent class.
* In interface allow main method from 1.8 version onwards .
* Main purpose of static method ,If every thing is static not going class it going for interface.

**package** StaticInterface;

**interface** P1{

**public** **static** **void** m1() {

System.***out***.println("Interface static method");

}

}

**public** **class** Test **implements** P1 {

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

P1.*m1*(); 🡺Valid

//m1(); 🡺InValid

//Test.m1(); 🡺Invalid **CT**

Test t=new Test();

//t.m1(); 🡺Invalid **CT**

}

}

**Default method and static method:-**

* From 1.8 version onwards in side interface you can use static methods and defaults method.
* Without effecting implementation classes if we want to add new method to the interface will go for default method.
* To define general utility methods inside interface with static methods.

**Predefined Functional Interfaces:-**

Predicate(I):-

* It is a functional interface ,It contain only one abstract method
* Public abstract boolean test(T t)
* Syntax

Interface Predicate<T>

{

Public boolean test(T t)

}

**Example:-**

**package** PreFunction;

**import** java.util.function.Predicate;

**public** **class** PreidcateLenString {

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

Predicate<String>p=(s)->s.length()>5;

System.***out***.println(p.test("Mirza Faisal"));

System.***out***.println(p.test("Mirza"));

//WAP to find string length whose length greater than 5

String s2[]= {"Mirza","satya","Bunu","Chunky","Sipu"};

Predicate<String>p1=s3->s3.length()>=5;

**for**(String a:s2) {

**if**(p1.test(a)) {

System.***out***.println(a);

}

}

//WAP to find string length whose length it even number

System.***out***.println("Even length string number");

Predicate<String>p2=s3->s3.length()%2==0;

**for**(String a:s2) {

**if**(p2.test(a)) {

System.***out***.println(a);

}

}

}

}

**Example:-**

**package** PreFunction;

**import** java.util.ArrayList;

**import** java.util.function.Predicate;

**class** Employee{

String name;

**int** salary;

**public** Employee(String name, **int** salary) {

**this**.name = name;

**this**.salary = salary;

}

}

**public** **class** PredicateEmployee {

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

ArrayList<Employee> al=**new** ArrayList<>();

al.add(**new** Employee("Mirza",5000));

al.add(**new** Employee("Satya",3000));

al.add(**new** Employee("Tanmaya",2000));

al.add(**new** Employee("Soubhagya",1500));

al.add(**new** Employee("Swagat",6000));

Predicate<Employee> p1=(e)->e.salary>3000;

**for**(Employee e1:al) {

**if**(p1.test(e1)) {

System.***out***.println(e1.name+":"+e1.salary);

}

}

}

}

Predicate Joining:-

* Multiple predicates to combine together to checks very complex condition.

Syntax

* P1.and(p2).text(34)

**package** PreFunction;

**import** java.util.function.Predicate;

**public** **class** PredicatesJoining {

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

**int** a[] = { 5, 165, 16, 516, 13, 1, 84, 67, 8, 7, 9, 8, 48, 79, 88, 68, 98, 99, 77, 43, 0 };

Predicate<Integer> p1 = i -> i % 2 == 0;

Predicate<Integer> p2 = i -> i > 12;

// and() ,or(),negate()

**for** (**int** x : a) {

**if** (p1.and(p2).test(x)) {

System.***out***.println("The numbers which are even and greter than 12 :"+x);

}

}

}

}

* P1.or(p2)

**package** PreFunction;

**import** java.util.function.Predicate;

**public** **class** PredicatesJoining {

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

**int** a[] = { 5, 165, 16, 516, 13, 1, 84, 67, 8, 7, 9, 8, 48, 79, 88, 68, 98, 99, 77, 43, 0 };

Predicate<Integer> p1 = i -> i % 2 == 0;

Predicate<Integer> p2 = i -> i > 12;

// and() ,or(),negate()

**for** (**int** x : a) {

**if** (p1.or(p2).test(x)) {

System.***out***.println("The numbers which are even and greter than 12 :"+x);

}

}

}

}

* P1.negate():-it is which condition will provide its given opposite

**package** PreFunction;

**import** java.util.function.Predicate;

**public** **class** PredicatesJoining {

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

**int** c=0;

**int** a[] = { 5,35, 165, 16, 516, 13, 1, 84, 67, 8, 7, 9, 8, 48, 79, 88, 68, 98, 99, 77, 43, 0 };

System.***out***.println(a.length);

Predicate<Integer> p1 = i -> i % 2 == 0;

Predicate<Integer> p2 = i -> i > 12;

// and() ,or(),negate()

**for** (**int** x : a) {

**if** (p1.negate().test(x)) {

System.***out***.println("The numbers which are not even and greter than 12 :"+x);

c++;

}

}

System.***out***.println(c);

}

}

Function(I):-

* It is a functional interface ,It contain only one abstract method
* Public abstract R apply<T,R> 🡺T stands inputtype ,R stands for returntype
* Syntax

Interface Function<T,R>

{

Public R apply(T t)

}

**Example:-**

**package** PreFunction;

**import** java.util.function.Function;

**public** **class** Function2 {

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

Function<String,Integer>f =s->s.length();

System.***out***.println(f.apply("Mirza"));

System.***out***.println(f.apply("Soubhagya"));

Function<String,String>f1 =s1->s1.toUpperCase();

System.***out***.println(f1.apply("Mirza"));

System.***out***.println(f1.apply("Soubhagya"));

}

}

**Example:-**

**package** PreFunction;

**import** java.util.ArrayList;

**import** java.util.function.Function;

**class** Student{

String name;

**int** id;

**int** marks;

**public** Student(String name, **int** id, **int** marks) {

**this**.name = name;

**this**.id = id;

**this**.marks = marks;

}

}

**public** **class** FunctionGrade {

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

ArrayList<Student> al=**new** ArrayList<Student>();

al.add(**new** Student("Mirza", 101, 65));

al.add(**new** Student("Satya", 102, 85));

al.add(**new** Student("Swagat", 103, 45));

al.add(**new** Student("Chinu", 104, 35));

al.add(**new** Student("Jaga", 108, 25));

al.add(**new** Student("Ano", 203, 95));

al.add(**new** Student("Amin", 100, 50));

Function<Student, String> f=s->{

**int** marks=s.marks;

String grade="";

**if**(marks>=80) grade="A-Top Class";

**else** **if**(marks>=60) grade="B-First Class";

**else** **if**(marks>=50) grade="C-Second Class";

**else** **if**(marks>=35) grade="D-Third Class";

**else** grade="F- Failed ";

**return** grade;

};

**for** (Student student : al) {

System.***out***.println("Student Name: "+student.name);

System.***out***.println("Student Id: "+student.id);

System.***out***.println("Student Marks: "+student.marks);

System.***out***.println("Student Grade: "+f.apply(student));

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

}

}

}

Function Chaining:-

* f1.andThen(f2).apply(i)🡺f1 first and then f2
* f1.compose(f2).apply(i)🡺f2 first and then f1

Example:-

**package** PreFunction;

**import** java.util.function.Function;

**public** **class** FunctionChanning {

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

Function<Integer, Integer> f1=i->3\*i;

Function<Integer, Integer> f2=i->i\*i\*i;

System.***out***.println(f1.apply(2));

System.***out***.println(f2.apply(2));

System.***out***.println(f1.andThen(f2).apply(2));//

System.***out***.println(f1.compose(f2).apply(2));//

}

}

**Consumer:-**

* Consumer interface is form java.util.function package.
* It is functional interface an has one abstract method called accept().
* I t has only one method Accept()
* Accept method is form consumer interface it accept object type of data as an argument but has a return type void.

**Example:-**

package PreFunction;

import java.util.ArrayList;

import java.util.function.Consumer;

import java.util.function.Function;

import java.util.function.Predicate;

class Student{

String name;

int id;

int marks;

public Student(String name, int id, int marks) {

this.name = name;

this.id = id;

this.marks = marks;

}

}

public class FunctionGrade {

public static void main(String[] args) {

ArrayList<Student> al=new ArrayList<Student>();

al.add(new Student("Mirza", 101, 65));

al.add(new Student("Satya", 102, 85));

al.add(new Student("Swagat", 103, 45));

al.add(new Student("Chinu", 104, 35));

al.add(new Student("Jaga", 108, 25));

al.add(new Student("Ano", 203, 95));

al.add(new Student("Amin", 100, 50));

Function<Student, String> f=s->{

int marks=s.marks;

String grade="";

if(marks>=80) grade="A-Top Class";

else if(marks>=60) grade="B-First Class";

else if(marks>=50) grade="C-Second Class";

else if(marks>=35) grade="D-Third Class";

else grade="F- Failed ";

return grade;

};

Predicate<Student> p=s->s.marks>=60;

Consumer<Student> c=s1-> {

System.out.println("Student Name: "+s1.name);

System.out.println("Student Id: "+s1.id);

System.out.println("Student Marks: "+s1.marks);

System.out.println("Student Grade: "+f.apply(s1));

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

};

for (Student student:al) {

if(p.test(student)) {

c.accept(student);

}

}

}

}

**Supplier:-**

Accept Just supply my required objects and it won’t any input .It is functional interface it has one abstract method get().

Syntax:-

Interface Sypplier<R>

{

Public R get();

**Example:-**

package PreFunction;

import java.util.Date;

import java.util.function.Supplier;

public class Supplier1 {

public static void main(String[] args) {

Supplier<Date> s=()->new Date();

System.out.println(s.get());

// Wap 6 digit otp generate

Supplier<String>s1=()->{

String otp="";

for(int i=0;i<6;i++) {

otp=otp+(int)(Math.random()\*10);

}

return otp;

};

System.out.println(s1.get());

}

}

**BiPredicate:-**

Normal predicate contain only one input .In BiPredicate you can give 2 input.

package BiPreFunction;

import java.util.function.BiPredicate;

public class BiPredict {

public static void main(String[] args) {

BiPredicate<Integer, Integer> p=(a,b)->(a+b)%2==0;

System.out.println(p.test(20, 11));

}

}

**BiFunction:-**

**package** BiPreFunction;

**import** java.util.ArrayList;

**import** java.util.function.BiFunction;

**class** Employee{

String name;

**int** id;

**public** Employee(String name, **int** id) {

**super**();

**this**.name = name;

**this**.id = id;

}

}

**public** **class** BiFunction1 {

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

ArrayList<Employee>l=**new** ArrayList<Employee>();

BiFunction<String, Integer, Employee> f=(name,id)->**new** Employee( name,id);

l.add(f.apply("Mirza", 10122));

l.add(f.apply("Satya", 20445));

l.add(f.apply("Swagat", 10356));

l.add(f.apply("Mukesh", 18368));

l.add(f.apply("Jagadish", 12684));

l.add(f.apply("Chinu", 28124));

**for**(Employee employee:l) {

System.***out***.println("Employee Name:"+employee.name);

System.***out***.println("Employee Name:"+employee.id);

System.***out***.println();

}}}

|  |  |  |
| --- | --- | --- |
| **Function** | **Method** | **Joining or Chaining** |
| Predicate | test() | and(),or(),negate() |
| Function | apply() | andThen(),compose() |
| Consumer | accept() | andThen() |
| Supplier | get() | no method |

|  |  |
| --- | --- |
| BiPredicate<T,U> | Represents a predicate (boolean-valued function) of two arguments. |
| BiConsumer<T,U> | Represents an operation that accepts two input arguments and returns no result. |
| BiFunction<T,U,R> | Represents a function that accepts two arguments and produces a result. |

Primitive Version of Functional Interface:-

**Primitive Predicate Interface:-**

* Intpredicate
* LongPredicate
* DoublePredicate

**Primitive Function Interface:-**

* Intpredicate
* LongPredicate
* DoublePredicate

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Property | Predicate | Function | Consumer | Supplier |
| Purpose | To make some input and performance some conditional check | To Take some input and perform required Operation and return the result | To consumer some input and perform required Operation .It won't return anything | To Supply some value base on our requirement |
| Interface Declaration | Interface Predicate<T>{ ………………….. } | Interface Function<T,R>{ ………………….. } | Interface Consumer<T>{ ………………….. } | Interface Supplier<T>{ ………………….. } |
| Single Abstract Method(SAM) | public boolean test(T t); | public R apply(T t); | public void accept(T t); | public R get(); |
| Defaults Methods | and(),or(),negate() | andThen(),compose() | andThen() | no |
| Static Method | isEqual() | identify() | no | no |

Method Reference And Constructor reference :-

**Class Name:: method name**

**Object reference :: method name**

In the case for method reference different return types are allowed.

But only argument type must be matched.

**Example:-Static members**

**package** MethodReferrence;

**public** **class** MethdoReference1 {

**public** **static** **void** m1() {

**for** (**int** i = 0; i < 5; i++) {

System.***out***.println("Child Thread");

}

}

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

MethdoReference1 m=new MethdoReference1();

Runnable r=MethdoReference1::*m1*;

Thread t=**new** Thread(r);

t.start();

**for** (**int** i = 0; i < 5; i++) {

System.***out***.println("Main Thread");

}

}

}

**Example:-Nonstaic members**

**package** MethodReferrence;

**public** **class** Methodrefnonststic {

**public** **void** m1() {

**for** (**int** i = 0; i < 5; i++) {

System.***out***.println("Child Thread");

}

}

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

Methodrefnonststic mn=**new** Methodrefnonststic();

Runnable r=mn::m1;

Thread t=**new** Thread(r);

t.start();

**for** (**int** i = 0; i < 5; i++) {

System.***out***.println("Main Thread");

}

}

}

**Note:- In the method reference return type and modifies will change allowed but cannot be argument changes it throws** compile **time error.**

**Example:-Interface**

**package** MethodReferrence;

**interface** Add{

**public** **void** add(**int** a,**int** b);

}

**public** **class** Interf {

**public** **static** **void** sum(**int** x,**int** y){

System.***out***.println("The sum of two values is: "+(x-y));

}

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

Add i=(a,b)->System.***out***.println(a+b);

i.add(100, 200);

Add i1=Interf::*sum*;

i1.add(1000,2000);

}

}

Constructor reference:-

**Example:-**

**package** ConstrucorReference;

**class** Sample{

Sample(){

System.***out***.println("Sample class costructor");

}

}

**interface** Interf{

**public** Sample get() ;

}

**class** Test1{

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

Interf i=Sample::**new**;

Sample s1=i.get();

}

}

**Example-2:-**

**package** ConstrucorReference;

**class** Sample1{

Sample1(String s){

System.***out***.println("Sample class costructor : "+s);

}

}

**interface** Interf1{

**public** Sample1 get(String s) ;

}

**public** **class** Test2{

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

Interf1 i=Sample1::**new**;

Sample1 s1=i.get("Mirza");

Sample1 s2=i.get("Mukesh");

}

}

**Stream API:-**

* If wants to represents a group of objects as single entity then we should go for collection.
* If we want to process objects from the collection-Stream.
* Stream s=collection.stream();
* Whenever you calling stream method it will return Stream objects.

**Filter:-**

**package** streamAPI;

**import** java.util.ArrayList;

**import** java.util.List;

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

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

**public** **class** Filter {

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

ArrayList<Integer> al=**new** ArrayList<Integer>();

al.add(10);

al.add(20);

al.add(30);

al.add(15);

al.add(11);

System.***out***.println(al);

List <Integer>l2=al.stream().filter(i->i%2==0).collect(Collectors.*toList*());

System.***out***.println(l2);

}

}

**Map:-**

**package** streamAPI;

**import** java.util.ArrayList;

**import** java.util.List;

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

**public** **class** map1 {

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

ArrayList<Integer> marks=**new** ArrayList<Integer>();

marks.add(45);

marks.add(55);

marks.add(68);

marks.add(23);

marks.add(54);

marks.add(44);

marks.add(58);

marks.add(60);

marks.add(87);

System.***out***.println(marks);

List<Integer>marks1=marks.stream().map(i->i+5).collect(Collectors.*toList*());

System.***out***.println(marks1);

}

}

**Count:-**

**package** streamAPI;

**import** java.util.ArrayList;

**import** java.util.List;

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

**public** **class** Count1 {

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

ArrayList<Integer> marks=**new** ArrayList<Integer>();

marks.add(45);

marks.add(55);

marks.add(68);

marks.add(23);

marks.add(54);

marks.add(44);

marks.add(58);

marks.add(60);

marks.add(87);

marks.add(34);

marks.add(25);

System.***out***.println(marks);

System.***out***.println(marks.size());

**long** failedStuden=marks.stream().filter(i->i>=35).count();

System.***out***.println(failedStuden);

}

}

**Sorted(Default Natural Sorting Order (Comparable):-**

**package** streamAPI;

**import** java.util.ArrayList;

**import** java.util.List;

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

**public** **class** Comparable1 {

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

ArrayList<Integer> al=**new** ArrayList<>();

al.add(10);

al.add(20);

al.add(1);

al.add(30);

al.add(5);

al.add(100);

al.add(30);

System.***out***.println(al);

List<Integer> list=al.stream().sorted((i1,i2)->i1.compareTo(i2)).collect(Collectors.*toList*());//Accending Order

List<Integer> list1=al.stream().sorted((i1,i2)->-i1.compareTo(i2)).collect(Collectors.*toList*());//Decending Order

System.***out***.println(list);// [1, 5, 10, 20, 30, 30, 100]

System.***out***.println(list1);// [100, 30, 30, 20, 10, 5, 1]

}}

**Sorted(Default Natural Sorting Order (Comparator):-**

**package** streamAPI;

**import** java.util.ArrayList;

**import** java.util.Comparator;

**import** java.util.List;

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

**public** **class** Comaparetor1 {

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

ArrayList<String> l=**new** ArrayList<String>();

l.add("Mirza");

l.add("Satya");

l.add("Soubhagya");

l.add("Tanmaya");

l.add("Senapati");

l.add("Sarfarajuddin");

l.add("Azhar");

Comparator<String> c=(s1,s2)->{

**int** l1=s1.length();

**int** l2=s2.length();

**if**(l1<l2)**return** -1;

**else** **if** (l1>l2) **return** +1;

**else** **return** s1.compareTo(s2);

};

System.***out***.println(l);

List<String> list=l.stream().sorted(c).collect(Collectors.*toList*());

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

Mix And Max :-

**package** streamAPI;

**import** java.util.ArrayList;

**public** **class** MinMax {

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

ArrayList<Integer> marks=**new** ArrayList<Integer>();

marks.add(45);

marks.add(55);

marks.add(68);

marks.add(23);

marks.add(54);

marks.add(44);

marks.add(58);

marks.add(60);

marks.add(87);

marks.add(5);

marks.add(1);

System.***out***.println(marks);

Integer minresult=marks.stream().min((i1,i2)->i1.compareTo(i2)).get();

System.***out***.println(minresult);

Integer maxresult=marks.stream().max((i1,i2)->i1.compareTo(i2)).get();

System.***out***.println(maxresult);

}

}

ForEach Method:-

**package** streamAPI;

**import** java.util.ArrayList;

**import** java.util.function.Consumer;

**public** **class** forEach1 {

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

ArrayList<Integer> marks=**new** ArrayList<Integer>();

marks.add(45);

marks.add(55);

marks.add(68);

marks.add(23);

marks.add(54);

marks.add(44);

marks.add(58);

marks.add(60);

marks.add(87);

System.***out***.println(marks);

Consumer<Integer> c=(i)->{

System.***out***.println("The square of number is : "+i\*i);

};

marks.stream().forEach(c);

marks.stream().forEach(System.***out***::println);

}

}

Stream.of:-

**package** streamAPI;

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

**public** **class** StreamOf {

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

Stream<Integer> s=Stream.*of*(9,99,999,9999,99999,999999);

s.forEach(System.***out***::println);

Integer []i= {10,20,30,40,50,60};

Stream.*of*(i).forEach(System.***out***::println);

}

}