**JAVA 8**

**Functional Interface:**

It is first introduced in java 8.

An interface with **exactly only one abstract** method is known as functional interface.

It can **any number of static and default methods** but can contain only **one abstract** method.

**Abstract Method**: a method without any implementation.

Eg:

Interface something{

public void execute();

}

**NOTE:**

Inorder to make a interface as functional interface we have to annotate it with **@FunctionalInterface**

There are several types of functional interface :

Function

Consumer

Supplier etc.,

**Function:**

It is a type of functional interface it has two fields one to represent the input and other to represent the output type

Eg:

**Function<String,Integer>**

**String-input**

**Integer-output or return type**

**Class implement {**

**Psvm(String[] args){**

**Function fun =(String s)->s.length();**

**Sop(Fun.apply(“Manimaran”))**

**}}**

**CONSUMER:**

It is a type of functional interface it has only one part it takes only **input** and it **doesn’t return** anything. So it is mostly used for assignation part .

**STRUCTURE:CONSUMER<>**

Eg:

Class imp{

Psvm(String[] args){

Consumer<Integer> con=(String s)->SOP(“This is the input”,+s);

Con.accept(“Mani”);

}

}

**SUPPLIER:**

It is type of functional interface . It will not take anything but it will only has return type.it doesn’t take anything as input.

Eg:

Class imp{

Psvm(String[] args){

Supplier<Integer> sup=()->LocalDateTime.now()

SOP(Sup.get())

**PREDICATE:**

It is a type of predefined functional interface where it has only input stream type and it will return a type of Boolean.

**Structurea and Eg:**

Class Some{

Psvm(String[] args){

Predicate<Integer> pred=num->num%2==0;

Sop(pred.test(4));

}}

**RUNNABLE**:

Eg:

Class some{

Psvm(String[] args){

Thread thr=new Thread(()->SOP(“Runnable interface”));

Thr.start();

}

}

**LAMBDA EXPRESSION:**

it is introduced in java8.

It is an anonymous function means it is a function without any name and does not belongs to particular class.

It is mainly used in functional interface.

**Advantages of Java 8:**

->No need to write a name for the method.

->Body of the code is important

->no need to declare the return type of the code.

**Syntax for Lambda expression:**

**() -> {}**

()-lambda input parameters

->Arrow denoting Lambda

{}-denoting the lambda body

Eg: **WITHOUT PARAMETERS**

Interface Shape{

Public void draw();

}

Public shapes implements shape{

Public static void main(String[] args){

Shape shape=()->System.out.println(“There are several shapes.”);

Shape.draw()

}

**WITH PARAMETERS:**

Interface addition{

Int addable(inta ,intb);

}

Public implementation{

Psvm(String[] args){

Addition add=(a,b)-> a+b;

Add.addable(10,20);

}}

**NOTE:**

**In order to create a lambda expression we have to create a reference variable for the interface after that we can implement the body in it.**

**After assigning it to the body we have to call the interface class to execute the code.**

**METHOD REFERENCE:**

It is used to refer method of the functional interface.it is a compact and easy form of lambda expression.

Whenever you are using a lambda expression to just referring a method, we can replace our lambda expression with a method reference.

Method reference is of four types:

1. Reference to static method:

Syntax: **class::static methodname**

**Eg:**

Class some{

Psvm(String[] args){

Function<Integer,Double> fun=Math::sqrt;

Sop(fun.apply(4));

}

}

1. Reference to the instance of the particular object:

Syntax:**Object::instance method name**

**Eg:**

interface printable{

void print();

}

Class some{

Public void disp(String something){

String msg=something.tolowercase();

SOP(msg);

}

Psvm(String[] args){

Some so=new Some();

Printable pr=so::disp;

Sop(pr.print(“Manimaran”));

}

}

1. Reference to the instance method of the arbitrary object of specific type:

Syntax: **class::instancemethodname**

**It is mostly used to call a method of the argument**

Eg:

Class Some{

Psvm(String[] args){

Function<String,Integer> fun=String::length;

Sop(fun.apply(“Man”));

}

1. Reference to constructor:

Syntax: **classname::new**

Eg:

Class Some{

Psvm(String[] args){

List<String> fruits=new ArrayList<>();

Fruits.add(“Banana”);

Fruits.add(“Banana”);

Fruits.add(“Apple”);

Function<List<String>,Set<String>> fun=Hashset::new;

Fun.apply(fruits);

**///lambda expression**

**Function <List<String>,Set<String>> fun=(fruits)->new Hashset<>();**

**///**

}}

**OPTIONAL CLASS:**

It is specially introduced in java 8 to avoid null checks and nullpointerexception exception.

Optional is a single value container it either contains a value or not.

By using optional we can specify alternate values to return.

There are several ways in creating optional object. we can use the below mentioned methods to create a optional object.

Methods:wh

1. Of
2. Empty
3. ofNullable

1.empty:

Where empty is a static method which is predefined in the optional class. We can create a empty optional class object by using this empty method.

Syntax:

Optional<Object> option=Optional.empty();

SOP(option);

The above code will return an empty Optional class.

2.Of

When we need to create an optional class object to store the value of the other class if only when it contains some value in it else it will return null pointer exception.

Syntax and eg:

String name=”Manimaran”;

Optional <Object> option=Optional.of(name);

SOP(option);

**Answer:** It will return an optional with value present in it.

3.OfNullable:

It is used to create an optional with value if it is not null else it will ,create an empty optional value.

Syntax and eg:

String name=”Manimaran”

Optional<Object> option=Optional.ofNullable(name);

SOP(option);

**Answer:**It will return an Optional class with a value if is not null else it will return an empty Optional.

**.get()**

It is used to get the value of the Optional class created. It will return the value if it is present else it will throws **nosuchelement** exception.

**.isPresent()**

It is used to check whether the optained obtained class as value in it or not.it returns a Boolean value.

Eg:

If(option.isPresent()) {

}

Else{

}

**.orElse()**

It is used to return a default value when the optional class does not contain any value present in it.

It takes the type stream value as parameter.

Eg:

Sop(option.orElse(“default values”))

**Answer:** it will check whether a optional value is present else it will return the default value written in the orElse block.

**orElseGet(Supplier)**

It is used to return a default value when the optional class does not contain any value present in it.

It takes the type supplier(functional interface) as parameter.

Eg:

Sop(option.orElseGet(()->”default value”))

**orElseThrow()**:

instead of returning a default value for the value of the optional class we can throw a exception for that

It takes the type supplier(functional interface) as parameter.

Eg:

Sop(option.orElseThrow(()-> new illegalArgument Exception(“its illegal”)))

**ifPresent()**

it is used to check whether a optional class object as value in it or not. It wil not return anything if it doesn’t contains any value.

It takes the type consumer(functional interface) as parameter.

Eg:

Sop(option.ifPresent(()->sop(“value is there”)));

**Filter():**

It is used for ,conditional checking on optional class object.

It will take **predicate** as input type.

Syntax and eg:

String result=”aabbcc”;

Optional<String> op=Optional.of(result);

Op.filter(res->res.contains(“a”)).ifPresent(res-> System.out.println(res))

**.map()**

The map method in optional is providing a way to transform one optional type to another.

It will take **function** as input type.

Syntax and eg:

String result=”aabbcc”;

Optional<String> op=Optional.of(result);

Op.filter(res->res.contains(“a”)).map(res-> res.length()).ifPresent(res-> System.out.println(res))

**DEFAULT AND STATIC METHOD IN INTERFACES:**

**DEFAULT:**

The default method in java helps to add new features in the interface. We can override the default method if we want else we can.

Eg:

Interface Mani{

Void disp();

Default status(){

SOP(“this is status”);

}

}

Class Maran implements Mani{

@Override

Void disp(){

Sop(“This is going to be displayed”)

}

}

Class Main{

Psvm(String[] args){

Maran mar= new Maran();

SOP(mar.disp());

SOP(mar.status());

}

}

**STATIC METHODS:**

An Interface can have any number of static methods the static methods can be accessed only by the interface class name.

**We cannot override a static method.**

Eg:

Interface Mani{

Void disp();

Static void status(){

SOP(“this is status”);

}

}

Class Maran implements Mani{

@Override

Void disp(){

Sop(“This is going to be displayed”)

}

}

Class Main{

Psvm(String[] args){

Maran mar= new Maran();

SOP(mar.disp());

SOP(Mani.status());

}

}

Static 🡪 compile time pomorphism

Others🡪 runtime polymorphism

**STREAM API**

It represents a sequence of object from the source , which supports aggregate operations.

**Aggregate operation**- the operation which are used to compute a single value from a collection of values.

**Working:**

On top of stream we can perform **aggregate operations** and collecting it in the form of collection.

We can create stream using the below syntax:

**Syntax:**

1. **creating stream directly:**

Stream<i/p type > str=Stream.of(“a”,”b”,”c”);

Str.forEach(System.out::println)—> print the elements of the stream in the console.

1. **Creating stream from Source:**

Int[] a={1,2,3,4,55,6};

List<Integer> lst=Arrays.asList(a);

Stream<Integer> str=lst.stream();

1. **Creating stream from array directly using array.stream**

Int[] a={1,2,3,4,55,6};Arrays.stream(a);

Stream<Integer> str= Arrays.stream(a);

**OPERATIONS IN STREAM:**

**Filter():**

It is used to do conditional operations on streams. It contains predicate as input type.

Eg:

Int[] arr={33,2,99,34,23,11,12}

List<Integer> lst=Array.asLists(arr);

SOP(Lst.stream.filter(a->a>15))

Or

SOP(Lst.stram.filter(a->a>5).collect(Collectors.tolist))

**SORTING IN STREAM API:**

**Sorting in ascending order:**

**For Collections:**

**Syntax :**

.sorted(Comparator.naturalorder)

Or

.Sorted()

Or

.sorted(a1.a2->a1.compareTo(a2))

Eg:

Class Some{

Psvm(String[] args){

List<String> ls=new ArrayList<>();

Ls.add(“Mani”);

Ls.add(“RAja”);

Ls.add(“Bala”);

Sop(ls);

List<String> ls1=ls.stream.sorted().collect(Collector.tolist)

List<String> ls2=ls.stream.sorted(Comparator.naturalOrder).collect(Collector.tolist)

List<String> ls3=ls.stream.sorted((o1,o2)->o1.compareTo(o2)).collect(Collector.tolist)

}

}

**For collection objects:**

Class Students{

Private int rollno;

Private String name;

Private int rank;

Getters and setters method;

constructors;

}

Class Main{

Psvm(String args[])[

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

Ls.add(new Student(data));

Ls.add(new Student(data));

Ls.add(new Student(data));

List<student>ls1=ls.stream.sorted((o1,o2)>o1.getrank().compareTO(o2.getrank())).collect(Collector.tolist);

Or

List<student>ls1=ls.stream.sorted((o1,o2)>(int)(o2.getrank()-o1.getrank())).collect(Collector.tolist);

Or

List<student>ls1=ls.stream.sorted(comparator.comparing(Student::getrank)).collect(Collector.tolist);

}

}

**Sorting in descending order:**

**Syntax :**

.sorted(Comparator.reverseorder)

Or

.sorted(a1.a2->a2.compareTo(a1))

Eg:

Class Some{

Psvm(String[] args){

List<String> ls=new ArrayList<>();

Ls.add(“Mani”);

Ls.add(“RAja”);

Ls.add(“Bala”);

Sop(ls);

List<String> ls2=ls.stream.sorted(Comparator.reverseOrder).collect(Collector.tolist)

List<String> ls3=ls.stream.sorted((o1,o2)->o2.compareTo(o1)).collect(Collector.tolist)

}

}

**For collection objects:**

Class Students{

Private int rollno;

Private String name;

Private int rank;

Getters and setters method;

constructors;

}

Class Main{

Psvm(String args[])[

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

Ls.add(new Student(data));

Ls.add(new Student(data));

Ls.add(new Student(data));

List<student>ls1=ls.stream.sorted((o1,o2)>o1.getrank().compareTO(o2.getrank())).collect(Collector.tolist);

Or

List<student>ls1=ls.stream.sorted((o1,o2)>(int)(o1.getrank()-o2.getrank())).collect(Collector.tolist);

Or

List<student>ls1=ls.stream.sorted(comparator.comparing(Student::getrank).**reversed()).**collect(Collector.tolist);

}

}

**MAPPING IN STREAMS:**

Generally we can say that we use map to do operations of the streams or collections i.e., manipulating the values of the collections or streams.

It takes consumer type interface

**Syntax:**

**Map(Function<T,T>)**

Eg:

Class Main{

Psvm(String[] args){

List<integer> ls=new ArrayList<>();

Ls.add(1);

Ls.add(2);

Ls.add(4);

List<integer> ls1=ls.stream.map(n->n\*4).collect(collectors.tolist).forEach(System.out::println);

Print(ls1)

}

}

**FINDFIRST AND FINDANY:**

We use both of these methods to find the elements in the stream.

It returns a optional object.

Findfirst-> returns first element of the stream

findAny-> returns any element of the stream

**SYNTAX and eg:**

**FindFirst()**

**findAny()-🡪 it is mostly suitable for parallel stream**

**Eg:**

Class Some{

Psvm(String[] args){

List<Integer> ls=new ArrayList<>();

Ls.add(10);

Ls.add(19);

Ls.add(20);

Ls.add(23);

Ls.add(22);

Optional<Integer> ls1=ls.stream.findFirst()

If(ls.isPresent()){

SOP(ls.get());

}

Else{

SOP(“Empty optional class”);

}

Optional<Integer> ls11=ls.stream.findAny()

If(ls11.isPresent()){

SOP(ls.get());

}

Else{

SOP(“Empty optional class”);

}

}

}

**COUNT,MIN and MAX methods:**

By using this methods we can return the count(total number of elements),minimum and maximum of the elements in the stream.

**SYNTAX:**

Class Some{

Psvm(String[] args){

Stream<Integer> st=Stream.of(1,2,3,4,55,66);

SOP(st.count());

Integer min=st.min(Comparator.comparing(Integer.valueOf)).get()

Integer max= st.max(Comparator.comparing(Integer.valueOf)).get()

SOP(max+” ”+min);

}

}

**forEach method:**

we can use the forEach method to method to iterate over the elements and it is defined inside the **Iterable** class and also in stream class.

It takes consumer interface.

**Syntax and eg:**

forEach(consumer<>)

class Main{

psvm(String args[]){

int[] arr={1,2,3,4,5,6,6};

List<Integer> ls=Array.aslists(arr);

Ls.stream.forEach(System.out::println);

}

}

**forEach method in map:**

class Main{

psvm(String args[]){

Map<Integer,String> map=new HashMap<Integer,String>();

Map.put(key,value);

Map.put(key,value);

Map.put(key,value);

Map.put(key,value);

Map.forEach((k,v)-> {sop(map.getKey();sop(map.getvalue();))}

}

}

**Series and parallel streams:**

|  |  |
| --- | --- |
| **series** | **parallel** |
| Runs in single core | It utilizes all the cores |
| Order is maintained | Order is not maintained |
| Only single iteration takes place at a time | Multiple iterations takes place simultaneously. |

**Date and time in stream():**

Java Date Time classes are not defined consistently.

There are no clearly defined classes for time, timestamp, formatting, and parsing

Date class doesn’t provide internationalization, there is no timezone support.

**To overcome the above mentioned problems we need JAVA 8 date and time API.**

[**1. LocalDate**](https://www.digitalocean.com/community/tutorials/java-8-date-localdate-localdatetime-instant#1-localdate)

LocalDate is an immutable class that represents Date with default format of yyyy-MM-dd. We can use now() method to get the current date. We can also provide input arguments for year, month and date to create LocalDate instance.

This class provides an overloaded method for now() where we can pass ZoneId for getting dates in a specific time zone

Syntax:

LocalDate.now()-> to get the current date in the format (yyyy-mm-dd)

LocalDate.of(2022,Month.January,14)-> to get the mentioned format date.

localDate.now(ZoneId .of(zone))-> to get the date of a particular zone

localDate.ofYearDay(year,days)🡪 it returns the date after the mentioned number of day in the year

### [2. LocalTime](https://www.digitalocean.com/community/tutorials/java-8-date-localdate-localdatetime-instant#2-localtime)

LocalTime is an immutable class whose instance represents a time in the human readable format. It’s default format is hh:mm:ss.zzz. Just like LocalDate, this class provides time zone support and creating instance by passing hour, minute and second as input arguments.

**Syntax:**

LocalTime.now()🡪 to get the current time in the format(hh:mm:ss.zzz)

LocalTime.of(12,25,45,44)🡪to get the mentioned time in the format

localTime.now(ZoneId.of(zone))🡪to get the time of the mentioned zone

localTime.ofSecondofDay(seconds)🡪it returns the time after seconds.

### [3. LocalDateTime](https://www.digitalocean.com/community/tutorials/java-8-date-localdate-localdatetime-instant#3-localdatetime)

LocalDateTime is an immutable date-time object that represents a date-time with default format as yyyy-MM-dd-HH-mm-ss.zzz. It provides a factory method that takes LocalDate and LocalTime input arguments to create LocalDateTime instance.

**SYNTAX:**

localDateTime.now()🡪 to get the current date and time in the format yyyy-mm-ddThh:mm:ss.zzz

localDateTime.of(2016,Month.March,21,1,10,20,30,)

localDateTime.of(LocalDate.now(),LocalTime.now())🡪to get the current date and time in the format yyyy-mm-ddThh:mm:ss.zzz

localDateTime.now(ZoneId.of(zone))🡪 to get the date and time of the zone mentioned

**Manipulating the date and time:**

Most of the Date Time principle classes provide various utility methods such as plus/minus days, weeks, months etc. There are some other utility methods for adjusting the date using **TemporalAdjuster**and to calculate the period between two dates.

**Syntax:**

Date.getYear()🡪 to get the year from the date

Date.isLeapYear()🡪to check it is leap year or not

Date.atTime(LocalTime.now())🡪 to get the time from the mentioned date

Date.plusDays(30)🡪 to add mentioned number of days to the date

Date.plusWeeks(3)🡪 to add mentioned number of weeks to the date

Date.plusMonths(10)🡪 to add the mentioned number of months to the date

Date.minusDays(30)🡪 to minus the mentioned number of days to the date

Date.minusWeeks(3)🡪 to minus the mentioned number of weeks to the date

Date.minusMonths(10)🡪 to minus the mentioned number of months to the date

Date.with(TemporalAdjusters.lastDayofYear())🡪 to get the last day of the date mentioned

Date.with(TemporalAdjusters.FirstDayofMonth())🡪to get the first day of the month mentioned in the date

**COLLECTION API Improvements:**

The collection API helps in performing operation on the collection objects. Collection is a architecture which is used to store and manipulate a group of objects.

There are several improvements that are made in the collection .which represents the addition of several methods in the Java API collections. The additional methods are:

**replaceAll():**

by using this method we can replace the data In the collection (based on some condition).

It takes functional interface as argument.

Eg:

Class Student {

Private int id;

Private String name;

Private float marks;

Constructors;

Getters and setter methods;

}

Class Main{

Psvm(String args[]){

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

Ls.add(new Student(data));

Ls.add(new Student (data));

Ls.add(new Student (data));

Ls.add(new Student (data));

Ls.add(new Student (data));

forEach(System.out::println);

ls.replaceAll(e->{

if(e.getmarks()>80){

return e;

}

Else{

Return e.setmarks(e.getmarks()+5);

}

});

forEach(System.out::println);

}

}

**getOrDefault():**

This method is used in Maps. The main purpose of this is to get a default value for a key when the key value is not present in the map or it doesn’t contains any value for the key.it does not add the key and value to the map.

Eg:

Class Main{

Psvm(){

Map<Integer,String> map=new HAshMap<Integer,String>();

Map.put(k,va);

Map.put(k,va);

Map.put(k,va);

Map.put(k,va);

String ans=map.getOrDefault(key,defaultvalue);

SOP(ans);

}

}

**putIfAbsent():**

This method is used in Maps. The main purpose of this is to add the value for a key when the key value is not present in the map .it will add the key and value to the map.

Eg:

Class Main{

Psvm(){

Map<Integer,String> map=new HAshMap<Integer,String>();

Map.put(k,va);

Map.put(k,va);

Map.put(k,va);

Map.put(k,va);

String ans=map.putIfAbsent(key,defaultvalue);

SOP(ans);

}

}

//function joining

//predicate joining

//grouping

Register for blood donor

End point to store the blood groups.

1 Endpoint to Add the blood groups to database

2 Endpoint to get the list of all blood groups

searching

3 endpoint to return the list of countries

4 endpoint to return the list of states based on the country code.

5 endpoint to return the districts based on the state.

6 endpoint to return the list of donors based on the above search parameters

(blood group,country code, state and other things)🡪 make blood group and country as mandatory

7. endpoint to store the datas of donors

Fullname,bloodgroup,mobile number,country, state ,district,city, availability(boolean)