**Why Java 8?**

* Concise and minimal Code.
* To utilize functional programming.
* To enable parallel programming.

**Features of Java 8?**

* Lambda Expression
* Stream API
* Method reference Constructor reference
* Default method and Static method in Interface
* Functional Interface
* Optional Class
* Date and Time API
* Base64 Encode Decode
* Java IO Improvements
* Collection API Improvements

**Lambda Expression**

* It is an Anonymous Function
* There is no modifier, no return type, no method name.
* There is just -> symbol.

**Functional Interface**

* Interface having only and only one Single Abstract method and number of Defaults and Static Methods.
* We can invoke Lambda Expression by using Function Interface.
* Functional Interface act as Type for Lambda Expression.
* Functional interface is used to provide reference to lambda expressions.
* If there is Parent interface is functional interface and if extend this in child interface with having @FunctionalInterface annotation then the child interface also become functional interface.

**Default Methods in Interface**

* Default method having body in Interface.
* It’s have own implementation and also we can override it.
* If avoid ambiguity in this so there is two ways
* Override those methods and write own implementation
* Else in this overridden method call super method
* We can override the default method but we need to remove default keyword in overridden method.
* This overridden method is only in public.

**How Default method in interface handle the Diamond problem?**

interface MyInterface1  
{  
 default int myDiamond()  
 {  
 return 0;  
 }  
}  
interface MyInterface2  
{  
 default int myDiamond()  
 {  
 return 1;  
 }  
}  
public class DiamondProblem implements MyInterface1,MyInterface2{  
 public static void main(String[] args) {  
 }  
 @Override  
 public int myDiamond() {  
 return MyInterface1.super.myDiamond();

// Call super method or Write your own Implementation Code  
 }  
}

**Static Method in Interface**

* It contain the complete definition of the function.
* We Cannot be override or change in the implementation class.
* We can call directly static method using interface name.
* From Java 8 we can write public static void main method in interface.

**Method Reference**

It is like replacement lambda expression.

It used double colon :: for the operations.

Working like calling static method using class name.

For example 🡪 System.Out::println

**Difference between Anonymous Inner Class and Lambda Expression**

**Anonymous Inner Class**

AnonymousInnerClass anonymousInnerClass = new AnonymousInnerClass() {  
 @Override  
 public void sayHi() {  
 System.*out*.println("AnonymousInnerClass Hi");  
 }  
  
 @Override  
 public void sayBy() {  
 System.*out*.println("AnonymousInnerClass Bye");  
 }  
};

**Lambda Expression with Anonymous Inner Class**

MyInterface myInterface = () -> {  
 System.*out*.println("Lambda Expression Hi");  
};  
myInterface.sayHi();

**Predefined Functional Interface**

1. **Predicate<T>** - T – Generic Type and return **Boolean**

boolean test(T t);

**Predicate Joining** – It is nothing but in case if we have more than one predicate then there is a ways to compere this and get single result from it.

* **And**
* **Or**
* **Negate**

Predicate<String> startsWithLetterV = x -> x.toLowerCase().charAt(0) == 'v' ;

Predicate<String> endsWithLetterL = x -> x.toLowerCase().charAt(x.length() - 1) == 'l';

System.*out*.println( startsWithLetterV. and (endsWithLetterL).test("Vipul"));

System.*out*.println( startsWithLetterV. or (endsWithLetterL).test("Vipul"));

System.*out*.println( startsWithLetterV.negate().test("Vipul"));

1. **Function<T,R>** - T – One Generic Argument , R - Return Generic Result type

R apply(T t);

**Functional Chaining** – We can chain multiple function togather with **andThen** and **compose** method.

Function<Integer,Integer> function1 = x -> x \* 2;

Function<Integer,Integer> function4 = x -> x \* x \* x;

System.*out*.println(function1.andThen(function4).apply(3)); // 216

System.*out*.println(function1.compose(function4).apply(3)); // 54

In this example andThen function is work first function1 and process function4

Now other compose method it will processes first function4 and then calculate function1.

1. **Consumer<T> -** Accept one Argument and No Retune.

void accept(T t);

Consumer Chaining – in this we can use andThen method on it.

Consumer<Integer> squareMe = i -> System.*out*.println("Square is = "+i \* i);

squareMe.accept(5);

Consumer<Integer> doubleMe = i -> System.*out*.println("Square is = "+(i + i));

doubleMe.accept(5);

squareMe.andThen(doubleMe).accept(5);

o/p 🡪

Square is = 25

Double is = 10

Square is = 25

Double is = 10

1. **Supplier<T> -** Can’t take any argument but Return Generic Type.

T get();

**In supplier No chaining is required because it take no input it give us Output.**

BiPredicate<String,Integer> biPredicate1 = (str,x) -> str.length() == x;  
System.*out*.println(biPredicate1.test("ABCDE",5));  
  
BiFunction<String,String,Integer> biFunction = (x,y) -> x.length() + y.length();  
System.*out*.println(biFunction.apply("Hi","Hello"));  
  
BiConsumer<Integer,Integer> biConsumer = (x,y) -> System.*out*.println(x + y);  
biConsumer.accept(1,2);

BiFunction<String,String,String> biFunction = (str1,str2) -> str1 + str2;  
BinaryOperator<String> binaryOperator = (str1,str2) -> str1 + str2;  
System.*out*.println(binaryOperator.apply("Hi","Hello"));

UnaryOperator<Integer> unaryOperator = x -> x \* x;  
UnaryOperator<String> unaryOperator1 = str -> str.toLowerCase();

**Stream APIs**

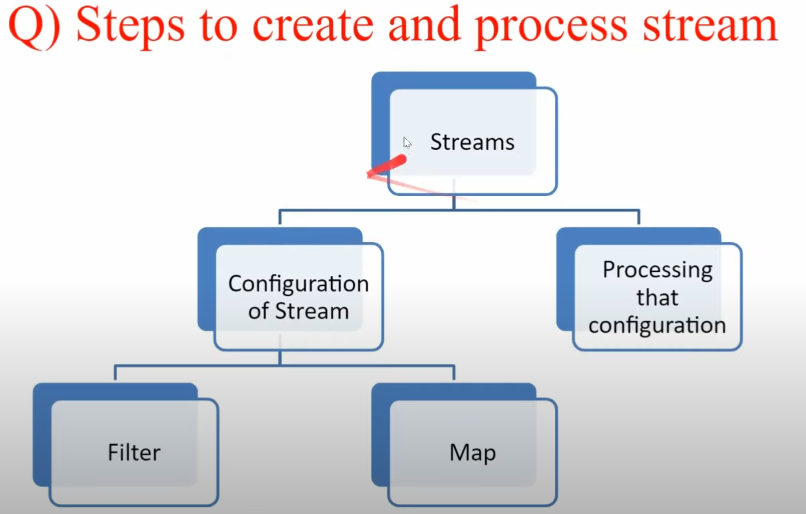
**What are Streams?**

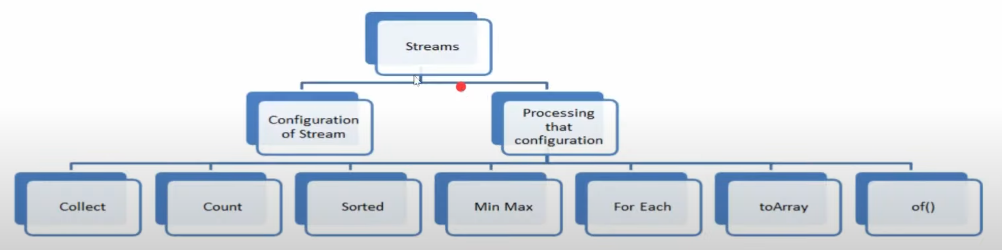
If we want to process the bulk object of collection then we will use Stream.

It is special iterator that allows the process on objects in a functional manner.

**Difference between Streams(Java 8) and java.io.Stream?**

Java 8 streams is use for operation on bulk object in functional manner and java.io.stream is use for file read and write operation.





**Map and FlatMap**

Map is like where we will convert a list into stream and using map it will create a new stream then we need to collect it.

For example we have list of employee and in employee two variables name and list of cities.

Now we will apply the map operation on list

class Employee  
{  
 public String empName;  
 public List<String> city;  
  
 public Employee(String empName, List<String> city) {  
 this.empName = empName;  
 this.city = city;  
 }  
  
 public String getEmpName() {  
 return empName;  
 }  
  
 public void setEmpName(String empName) {  
 this.empName = empName;  
 }  
  
 public List<String> getCity() {  
 return city;  
 }  
  
 public void setCity(List<String> city) {  
 this.city = city;  
 }  
  
 @Override  
 public String toString() {  
 return "Employee{" +  
 "empName='" + empName + '\'' +  
 ", city=" + city +  
 '}';  
 }  
}  
  
public class Map\_FlatMap {  
 public static void main(String[] args) {  
 List<Employee> empList = new ArrayList<>();  
 Employee employee1 = new Employee("ABC",Arrays.*asList*("Nagpur","Pune"));  
 Employee employee2 = new Employee("XYZ",Arrays.*asList*("Pune","Bangalore","Mumbai"));  
 Employee employee3 = new Employee("PQR",Arrays.*asList*("Nagpur","Pune","Delhi","Hyderabad"));  
 Employee employee4 = new Employee("LMN",Arrays.*asList*("Hyderabad","Bangalore"));  
 Employee employee5 = new Employee("GHI",Arrays.*asList*("Delhi","Mumbai"));  
 empList.add(employee1);  
 empList.add(employee2);  
 empList.add(employee3);  
 empList.add(employee4);  
 empList.add(employee5);  
 System.*out*.println(empList);  
  
 // Map  
 System.*out*.println("Print all Employee Name in List");  
 empList.stream().map(Employee::getEmpName).collect(Collectors.*toList*()).forEach(System.*out*::println);  
  
 // Now we need All City in Set order means Nor Repeated  
 // Using Map ->  
 Set<List<String>> cityListMap = empList.stream().map(Employee::getCity).collect(Collectors.*toSet*());  
 System.*out*.println(cityListMap);

// [[Nagpur, Pune, Delhi, Hyderabad], [Hyderabad, Bangalore], [Nagpur, Pune], [Pune, Bangalore, Mumbai], [Delhi, Mumbai]]  
  
 // Now using FlatMap  
 Set<String> cityListFlatMap = empList.stream().flatMap(employee -> employee.getCity().stream()).collect(Collectors.*toSet*());  
 System.*out*.println(cityListFlatMap);

// [Delhi, Nagpur, Pune, Mumbai, Hyderabad, Bangalore]  
 }  
}

**FlatMap will work for List of List**

In Map we pass function to map() it return a single value.

In FlatMap we pass function to flatMap() it return Stream of value then also we need stream on it the it will give us value.

FlatMap is combine Map and flatMap operation.

**Can we use hashcode() method to default implementation in Interface?**

No we can’t use or override this hashcode() in interface as default.

interface MyInterface  
{  
 default int hashCode() // A default Method cannot override a method form

java.lang.Object  
 {  
 return 0;  
 }  
  
}

**StringJoiner**

StringJoiner is final class for construct a sequence of characters separated by delimiters.

There is some method in StringJoiner

add – add new string character value in next element.

merge – it will merge two strings.

length – it will return length of string.

setEmptyValue – it will set the value empty if no elements have yet.

**Optional**

Optional is class which is use for avoid null pointer exception.