**Why Java 8?**

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

**Features of Java 8?**

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

**Lambda Expression**

* It is an Anonymous Function

**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.

**Default Methods in Interface**

* If avoid ambiguity in this so there is two ways
* Override those methods and write own implementation
* Else in this call super method

**Static Method in Interface**

* It contain the complete definition of the function.
* Cannot be override or change in the implementation class.

**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

**Predicate<T>** - T – Generic Type return Boolean

boolean test(T t);

**Function<T,R>** - T – One Generic Argument , R - return Result type

R apply(T t);

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

void accept(T t);

**Supplier<T> -** Can’t take any argument only Return.

T get();

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();

**Map and FlatMap**

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

For example we have list of employee and in employee two variables name and list of city’s.

Now we will apply the map opration 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.