Day13: Functional Interface in java8, Lambda Expression, Method reference, Java Stream API with functional programming

Functional programming in Java:

In computer science, **functional programming** is a programming paradigm where programs are constructed by applying and composing functions, i.e. decompose the problem into ‘functions.

It is a declarative type of programming style. Its main focus is on “what to solve” in contrast to an imperative style where the main focus is “how to solve”. the functional programming method focuses on results, not the process.

In functional programming, functions are treated as first-class citizens, meaning that they can be bound to names (including local identifiers), passed as arguments, and returned from other functions, just as any other data type can.

.

Functional Interface in Java:

- An interface which has only one abstract method is called as functional interface.
- This functional interface can have any number of **default, static** methods, but can contains only **one abstract** method.
- A functional interface can have methods of the **Object** class also.
- Functional Interface is also known as **Single Abstract Method Interfaces or SAM Interfaces**. It is a new feature in Java 8, which helps to achieve functional programming approach.

Some of the predefined interfaces in java which can be consider as functional interface :

java.lang.Comparable: public int compareTo(Object obj);

java.util.Comparator: public int compare(Object obj1, Object obj2);

java.lang.Runnable: public void run();

etc.

Note: It's recommended that all functional interfaces have an informative ***@FunctionalInterface*** annotation. This clearly communicates the purpose of the interface, and also allows a compiler to generate an error if the annotated interface does not satisfy the conditions. This interface belongs to **java.lang** package

Example:

```
@FunctionalInterface
interface Intr{
    void sayHello(String name);
}

class X implements Intr{
    public void sayHello(String name){
        System.out.println("Welcome "+name);
    }
    public static void main(String[] args) {
      Intr i1 = new X();
        i1.sayHello("Admin");
    }
}
```

Example: functional interface with default, static, and Object class methods

```
@FunctionalInterface
interface Intr{
    //one abstract method
    void sayHello(String name);

    //Object class method
    boolean equals(Object obj);

    //default method
    default void fun1(){
        System.out.println("inside the default method fun1 of Intr");
    }

    //static method
    static void fun2(){
        System.out.println("inside the static method fun2 of Intr");
    }


}

 class X implements Intr{


    @Override
    public void sayHello(String name){
        System.out.println("Welcome "+name);
    }

    public static void main(String[] args) {
        Intr i1 = new X();

        i1.sayHello("Admin");
        i1.fun1();
```

```
        System.out.println(i1.equals(i1));

        Intr.fun2();

    }
}
```

Note: A functional interface can extends another interface only when it does not have any abstract method.

Implementing a functional interface using Anonymous Inner class:

We can provide the implementation of an interface using Anonymous inner class also

Example:

```
@FunctionalInterface
interface Intr{
    void sayHello(String name);

}

class Main{

    public static void main(String[] args) {

        Intr i1 = new Intr() { //anonymous inner class
            @Override
            public void sayHello(String name) {
                System.out.println("Welcome "+name);
            }
        };

        i1.sayHello("Amit");
    }
}
```

Note:- to use the anonymous inner class the Interface need not be functional, i.e. an interface may have more than one abstract method also.

With the help if functional interface ,we can achieve functional programming in java using Lambda expression.

Lambda Expression in Java:

Lambda expression is, essentially, an anonymous or unnamed method. The lambda expression does not execute on its own. Instead, it is used to implement a method defined by a functional interface.

Using Lambda Expression we can represent an object of a functional interface in much more concise way.

Lambda Expression saves a lot of code. In case of lambda expression, we don't need to define the method again for providing the implementation. Here, we just write the implementation code.

Java lambda expression is treated as a function, so compiler does not create .class file.

Why use Lambda Expression:

- To provide the implementation of Functional interface.
- To write a short and concise code.

Java Lambda Expression Syntax:

Syntax:

```
(argument-list) -> {body}
```

Java lambda expression is consisted of three components.

1) Argument-list: It can be empty or non-empty as well.

2) Lambda operator: → It is used to link arguments-list and body of expression.

3) Body: It contains expressions and statements for lambda expression.

Let’s implement on functional interface using Lambda Expression:

```
@FunctionalInterface
interface Intr{
    void sayHello(String name);

}

public class Main {

        public static void main(String[] args) {

           Intr i1 = (String name) -> {
               System.out.println("Welcome "+name);
           };

           i1.sayHello("User");

        }
}
```

Note: in the argument list, mentioning data type is optional. and in case of single parameter the () small bracket is also optional.

Example:

```
 Intr i1 = (name) -> {
               System.out.println("Welcome "+name);
           };

// we can omit the () small bracket also

 Intr i1 = n -> {
               System.out.println("Welcome "+n);
           };
```

In case of zero or more than one parameter the () small bracket is mandatory.

Example:

```
@FunctionalInterface
interface Intr{
    void sayHello();
}


public class Main {

        public static void main(String[] args) {

           Intr i1 = () -> {
               System.out.println("Welcome User");
           };

           i1.sayHello();
        }
}
```

Example: Using multiple parameter:

```
@FunctionalInterface
interface Intr{

    void add(int num1, int num2);

}

public class Main {

        public static void main(String[] args) {

           Intr i1 = (n1,n2) -> {
               System.out.println("The Result is " +(n1+n2));
           };
```

```
            i1.add(10,20);

        }
}
```

Lambda Expression Body:

- The body of a lambda expression can contain zero, one or more statements.
- When there is a single statement curly brackets are not mandatory and the return type of the anonymous function is the same as that of the body expression.
- When there are more than one statements, then these must be enclosed in curly brackets (a code block) and the return type of the anonymous function is the same as the type of the value returned within the code block, or void if nothing is returned.
- If an abstract method of the functional interface has return type other than void, and if we want to implement using single statement and we don't use the {} curly bracket then **return** keyword is not allowed. but if take a {} curly bracket then return keyword is mandatory.

Example

```
@FunctionalInterface
interface Intr{

    int add(int num1, int num2);

}

public class Main {

        public static void main(String[] args) {

           Intr i1 = (n1,n2) -> {
               return n1+n2;
           };

      //if only a single statment body is there then {} is optional
      //and if we don't take the {} then return keyword is not allowed

          Intr i2 = (n1,n2) -> n1+n2;


            System.out.println(i1.add(10,20));
            System.out.println(i2.add(50,60));

        }
}
```

Matching a Lambda Expression against the functional interface is divided into four steps:

1. Interface should have only one abstract method.

2. Number of parameters of the Lambda Expression must match with the number of parameters of the method inside the interface.
3. Type of parameters should also match.
4. return type of the Lambda Expression should also match with the method defined in the interface.

Example: defining a and implementing a functional interface with an abstract method which will take Student object and print the student details.

```
@FunctionalInterface
interface Intr{

    void printDetails(Student student);

}

public class Main {

        public static void main(String[] args) {

          Intr i1 = (s) ->{

              System.out.println("Roll :"+s.getRoll());
              System.out.println("Name :"+s.getName());
              System.out.println("Marks :"+s.getMarks());

          };

          i1.printDetails(new Student(10,"Ram",850));
        }
}
```

Example: Passing Lambda Expression as an argument to a method:

```
interface Calculator {

    public int calculate(int a, int b);
}

 class MyClass {

    public static void main(String javalatte[]) {
        // this is lambda expression
        Calculator plusOperation = (a, b) -> a + b;
```

```
        Calculator minusOperation= (a, b) -> a*b;
        System.out.println(plusOperation.calculate(10, 34));
        System.out.println(minusOperation.calculate(12,5));
    }
}
```

```
@FunctionalInterface
interface Intr{

    int doMultiply(int num1,int num2);

}

public class Main {


    public static void fun1(Intr i1){

        System.out.println(i1.doMultiply(10,20));
    }

        public static void main(String[] args) {

            fun1( (n1,n2) -> n1*n2 );
        }
}
```

Example: returning Lambda Expression from a method:

```
@FunctionalInterface
interface Intr{

    void sayHello(String name);

}

public class Main {

   //returning LE from fun1 as object of functional interface Intr
    public static Intr fun1(){

        return  n -> System.out.println("Welcome "+n);
    }



        public static void main(String[] args) {

             Intr i1 = fun1();

             i1.sayHello("Admin");

        }
}
```

I Problem:

Let’s sort the List of Student according to their marks using Lambda Expression:

```
//Student.java
public class Student {

    private int roll;
    private String name;
    private int marks;

    public Student() {
    }

    public Student(int roll, String name, int marks) {
        this.roll = roll;
        this.name = name;
        this.marks = marks;
    }

    @Override
    public String toString() {
        return "Student{" +
                "roll=" + roll +
                ", name='" + name + '\'' +
                ", marks=" + marks +
                '}';
    }

    public int getRoll() {
        return roll;
    }

    public void setRoll(int roll) {
        this.roll = roll;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getMarks() {
        return marks;
    }

    public void setMarks(int marks) {
        this.marks = marks;
    }
}


//Main.java

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Main {
```

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

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

            students.add(new Student(10,"name1",780));
            students.add(new Student(12,"name2",480));
            students.add(new Student(14,"name3",680));
            students.add(new Student(15,"name4",580));

            Collections.sort(students, (s1,s2) -> s1.getMarks() > s2.getMarks() ? +1 : -1 );

            System.out.println(students);

        }
}
```

Method Reference:

It is a simplified form (short-cut) of Lambda Expression.

Java provides a new feature called method reference in Java 8. Method reference is used to refer method of functional interface. It is compact and easy form of lambda expression.

Instead of creating a Lambda Expression with all the details, with the help of method reference we can refer an existing class method to the functional interface implementation, which matches the condition of Lambda Expression.

Types of Method References:

There are following types of method references in java:

1. Reference to a static method.
2. Reference to an instance (non-static) method.
3. Reference to a constructor.

1.Reference to a Static Method:

You can refer to static method defined in the class as an implementation of a functional interface.

Syntax:

```
ClassName::methodName
```

Example:

```
@FunctionalInterface
interface Intr{

    void sayHello(String name);

}

public class Main {

        public static void fun1(String s){
            System.out.println("Using static Method reference Welcome "+s);
        }


        public static void main(String[] args) {

           Intr i1 = Main::fun1;

           i1.sayHello("Admin");

        }
}
```

Example2:

```
@FunctionalInterface
interface Intr{

    int convertToNumber(String number);

}

public class Main {

        public static int fun1(String s){
            System.out.println("Using static Method reference Welcome "+s);
            return Integer.parseInt(s);
        }


        public static void main(String[] args) {

           //Using static method reference
           Intr i1 = Main::fun1;
           System.out.println(i1.convertToNumber("200"));

            //Using static method reference
           Intr i2 = Integer::parseInt;
           System.out.println(i2.convertToNumber("500"));

           //using Lambda Expression
           Intr i3 = s -> Integer.parseInt(s);
           System.out.println(i2.convertToNumber("1000"));

        }
}
```

2.Reference to an instance (non-static) method:

Syntax:

```
object::methodName
```

Example:

```
@FunctionalInterface
interface Intr{

    void  printNumber(int number);

}


public class Main {

        public void fun1(int num){
            System.out.println("Using non-static Method reference Welcome "+num);
        }


        public static void main(String[] args) {

            Intr i1 = new Main()::fun1;
            i1.printNumber(12);

            Intr i2 = System.out::print;
            i2.printNumber(15);
        }
}
```

3.Reference to a constructor:

Syntax:

```
ClassName::new
```

Example:

```
@FunctionalInterface
interface Intr{

    void  sayHello();

}

public class Main {

    Main(){
        System.out.println("Method reference using Constructor");
    }
        public static void main(String[] args) {
```

```
            Intr i1 = Main::new;

            i1.sayHello();
        }
}
```

Some of the new functional interfaces introduced in java 8 to perform functional style of programming. these interfaces belongs to java.util.function package.

1. **Predicate<T>**
2. **Consumer<T>**
3. **Supplier<T>**
4. **Function<T,R>**

1. **java.util.function.Predicate<T>:**

   This interface contains only one abstract method called:

   ```
   public boolean test(T t);
   ```

   This method test() checks whether supplied obj satisfying a condition or not.

   Example:

   ```
   import java.util.function.Predicate;
   public class Main {

           public static void main(String[] args) {

               Predicate<Integer> p = i -> i > 0;

               System.out.println(p.test(10));//true
               System.out.println(p.test(-10));//false

           }
   }
   ```

   In java 8 Collection interface defines a method called:

   `public boolean removeIf(Predicate filter)`

   Based on the condition of Predicate, this method will remove/filter the elements from the Collection classes

Example: Removing the Students from the List whose marks is less than 700

```
import java.util.List;
import java.util.ArrayList;
public class Main{

  public static void main(String[] args)   {

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

    students.add(new Student(10, "name1", 650));
    students.add(new Student(12, "name2", 750));
    students.add(new Student(13, "name3", 550));
    students.add(new Student(14, "name4", 820));
    students.add(new Student(15, "name5", 720));
    students.add(new Student(16, "name6", 620));


    System.out.println(students);

    students.removeIf( student -> student.getMarks() < 700 );

    System.out.println(students);
}
}
```

2. **java.util.function.Consumer<T>:**

   It represents a function which takes in one argument and produces a result. However these kind of functions don’t return any value.

```
public void accept(T t);
```

Example:

```
import java.util.function.Consumer;
public class Main {

        public static void main(String[] args) {

            Consumer<Student> c = s -> {

                System.out.println("Roll is "+s.getRoll());
                System.out.println("Name is "+s.getName());
                System.out.println("Marks is "+s.getMarks());
            };

            c.accept(new Student(10,"Amit",850));
        }
}
```

Note :- from java 8 each collection classes contains a method called

```
default void forEach(Consumer c);
```

This forEach method defined as **default** method inside the **java.lang.Iterable** interface.

Example: iterating elements of a List using forEach method.

```
import java.util.ArrayList;
import java.util.List;
public class Main{
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("Football");
        list.add("Cricket");
        list.add("Chess");
        lList.add("Hocky");


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

        //or using Method referensce
        //list.forEach(System.out::println);

    }
}
```

3.java.util.function.Supplier<T>:

It represents a function which does not take in any argument but produces a value of type T.

method:

```
public T get();
```

Example:

```
import java.util.function.Supplier;
public class Main {

        public static void main(String[] args) {

            Supplier<String> s = () -> "This is from Lambda Expression";

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

            Supplier<Student> s2 = () -> new Student(10,"Ram",850);

            System.out.println(s2.get().getName());
```

```
        }
}
```

4.java.util.function.Function<T,R>

This interface defines an abstract method which will takes T type of object as parameter and returns R type of object.

```
public R apply(T t);
```

Example:

```
import java.util.function.Function;

public class Main {

        public static void main(String[] args) {

            Function<Integer,String> f = i -> "This is a numner "+i;

            System.out.println(f.apply(10));

            Function<String,Integer> f2 = s -> Integer.parseInt(s);

            System.out.println(f2.apply("200")+500);

            Function<String,Integer> f3 = Integer::parseInt;
            System.out.println(f3.apply("400")+200);
        }
}
```

Example2: Getting a Student object and returning greeting message with Student Name.

```
import java.util.function.Function;
public class Main {

        public static void main(String[] args) {


            Function<Student,String> f = s -> "Welcome "+s.getName().toUpperCase();

            String msg= f.apply(new Student(10,"Amit",850));

            System.out.println(msg);

        }
}
```

Java Stream API:

This API is also introduced in java 8. This API belongs to **java.util.stream** package.

The Stream API is used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.

java.util.stream package contains some library classes and interfaces by using which we can perform functional style of programming on the group of objects(collection of data).

This API has one main interface:

```
java.util.stream.Stream
```

Note:- Object of this **Stream** interface represents sequence of object from a source like collections.

The feature of java stream:

- The stream does not stores the elements, it only represents elements in a sequence.

  Example: wire does not store the electricity.
- It holds only objects, primitives are not allowed.
- Operation(filtering) performed on the stream does not modify its source.

  Example: filtering a stream obtained from a source(collection) produces a new stream with the filtered element rather than removing the elements from the source collection.
- With the help of stream obj we can perform various operations on the collection of objects in functional style, like filtering some elements, transform some elements, manipulate, sort, etc.
- Stream is lazy and evaluates code only when required.
- The elements of a stream are only visited once during the life of a stream. a new stream must be generated to revisit the same elements of the source.

From java 8, Collection interface provides following method to gets a Stream obj.

public default Stream<T> stream();

Note: We can use Stream as generic type for example:

Stream<String>: represents sequence of String object from a source(Collection object).

Stream<Employee>: represents sequence of Employee object from a source(Collection object).

Methods in Stream interface:

There are two types of methods in **Stream** interface:

1. Intermediate methods
2. Terminal methods

1. **Intermediate methods:** these methods returns a new stream object, these intermediate methods never gives the final result.
   most commonly used intermediate methods are:

   public Stream map(Function f);

   public Stream filter(Predicate p);

2. **Terminal methods:**- stream object returns a result only when terminal methods are called on the stream object, the terminal methods consumes that stream object and after that we can't use that stream object again.

   some of the terminal methods are:

   public void forEach(Consumer c);

   public R collect(Collector c);

   public Optional<T> min(Comparator c);

   public Optional<T> max(Comparator c);

   public boolean anyMatch(Predicate p)

   public boolean allMatch(Predicate p)

   public long count();

   Example1: forEach method:

```
ArrayList<String> al = new ArrayList<String>();

al.add("one");
al.add("one1");
al.add("one2");
al.add("one3");

Stream<String> ss=al.stream();

ss.forEach(i->System.out.println(i));
```

Difference Between Collection.stream().forEach() and Collection.forEach()

Collection.stream().forEach()	Collection.forEach()
Collection.stream().forEach() is also used for iterating the collection but it first converts the collection to the stream and then iterates over the stream of collection.	Collection.forEach() uses the collections iterator.
Unlike Collection.forEach() it does not execute in any specific order, i.e. the order is not defined.	If always execute in the iteration order of iterable, if one is specified.
During structure modification in the collection, the exception will be thrown later.	If we perform any structural modification in the collection by using the collection.forEach() it will immediately throw an exception.
If iteration is happening over the synchronized collection, then it does not lock the collection.	If iteration is happening over the synchronized collection, then it locks the collection and holds it across all the calls.

Example 2: print all the Student marks from the List of Student using stream.

```
public class Main {

        public static void main(String[] args) {

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

            students.add(new Student(10,"Name1",850));
            students.add(new Student(12,"Name2",750));
            students.add(new Student(13,"Name3",650));
            students.add(new Student(14,"Name4",950));
            students.add(new Student(15,"Name5",750));

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

        }
}
```

Example: filter() method:

This method take Predicate object as an argument and filter the stream based on the Predicate condition, and returns the filtered elements in a another stream object.

It will not filter the elements from the source collection object.

Example: filter the students from the List of Student whose marks is greater than 800.

```
import java.util.ArrayList;
import java.util.List;

public class Main {

        public static void main(String[] args) {

            List<Student> students = new ArrayList<>();
            students.add(new Student(10,"Name1",850));
            students.add(new Student(12,"Name2",750));
            students.add(new Student(13,"Name3",650));
            students.add(new Student(14,"Name4",950));
            students.add(new Student(15,"Name5",750));

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

            //Stream<Student> str2 = str1.filter(s -> s.getMarks() > 800);

            //str2.forEach( s -> System.out.println(s.getName()));

      students.stream()
          .filter(s -> s.getMarks() > 800)
            .forEach(s -> System.out.println(s.getName()));


        }
}
```

We Problem:

Create another List of Student whose marks is greater than 800 from a List of Student

```
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;
public class Main {

        public static void main(String[] args) {

            List<Student> students = new ArrayList<>();
            students.add(new Student(10,"Name1",850));
            students.add(new Student(12,"Name2",750));
            students.add(new Student(13,"Name3",650));
            students.add(new Student(14,"Name4",950));
            students.add(new Student(15,"Name5",750));

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

            //Stream<Student> str2 = str1.filter(s -> s.getMarks() > 800);

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

            List<Student> anotherList= students.stream()
                                            .filter(s -> s.getMarks() > 800)
```

```
                                                .collect(Collectors.toList());
        }
}
```

map() method:

This method takes a Function object as an argument and map the element to the new element and returns the newly mapped elements in another stream object.

Example1: from a list of name, generate a new list which has name with welcome msg:-

```
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;
public class Main {

        public static void main(String[] args) {

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

            al.add("ramesh");
            al.add("suresh");
            al.add("mukesh");
            al.add("ajay");

            Stream<String> ss=al.stream();


            //List list=ss.map(s->{return "welcome "+s;}).collect(Collectors.toList());

            //or without using return keyword

            List list=ss.map(s-> "welcome "+s).collect(Collectors.toList());


            list.stream().forEach(s->System.out.println(s));

        }
}
```

Example: getting List of Uppercase String from the List of lowercase string:

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class Main {

    public static void main(String[] args) {

        List<String> citiesL= Arrays.asList("delhi","mumbai","chennai","kolkata");
```

```
        List<String> citiesU =  citiesL.stream().map( city -> city.toUpperCase()).collect(Collectors.toList());

        System.out.println(citiesL);
        System.out.println(citiesU);


    }
}
```

You Problem:

Convert a list of String into a list of Integer(length of that string) and then filter all even numbers inside another List.

Adding all the marks of students:

```
import java.util.ArrayList;
import java.util.stream.Collectors;
public class Main {

    public static void main(String[] args) {

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

        al.add(new Student(10, "n1", 852));
        al.add(new Student(12, "n2", 854));
        al.add(new Student(13, "n3", 851));
        al.add(new Student(14, "n4", 856));
        al.add(new Student(15, "n5", 858));

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

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

Using Allmatch, Anymatch, noneMatch methods:

These methods take a Predicate object as an argument.

Example:

```
public class Main {

    public static void main(String[] args) {

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

        al.add(new Student(10, "n1", 852));
        al.add(new Student(12, "n2", 854));
        al.add(new Student(13, "n3", 851));
```

```
        al.add(new Student(14, "n4", 856));
        al.add(new Student(15, "n5", 858));


        boolean b=al.stream().allMatch(s->s.getMarks()>800);

        System.out.println(b);

    }
}
```

References:

https://www.geeksforgeeks.org/

https://www.javatpoint.com/

https://docs.oracle.com/javase/tutorial/java