

Java Functional Programming

I think functional programming is an exceptional approach. Because, if we do normal Java programming then it is much bigger but if we program using Java functional programming then it is much smaller and the code capacity increases a lot.

And for functional programming we have to use functional interface, lambda expression, method reference, stream, filter, collector. The functional interface has a single abstract method. There may be default method and static method as well. This single abstract method is called functional method.

Functional Interfaces provide target types for lambda expressions and method references. Each functional interface has a single abstract method, called the functional method for that functional interface, to which the lambda expression's parameter and return types are matched or adapted. Functional interfaces can provide a target type in multiple contexts, such as assignment context, method invocation, or cast context:

// Assignment Context

```
Predicate<String> p=String::isEmpty;
```

//Method invocation context

```
Stream.filter(e->e.getSize()>10)..
```

//Cast context

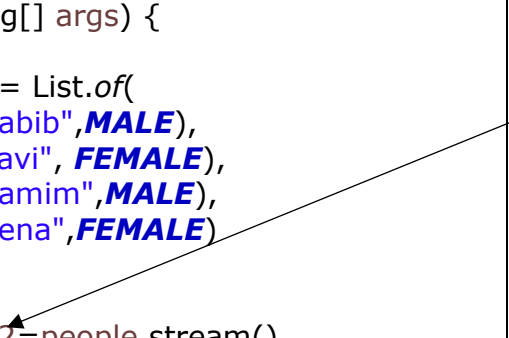
```
Stream.map((ToIntFunction) e->e.getSize())....
```

The following examples are normal Java programming examples. So you have to code a lot more: this program we find out the name which gender is FEMALE.

Example_1	Imperative approach which is normal java program.
<pre>package java_imperative_example; import java.util.ArrayList; import java.util.List; import static java_imperative_example.Example_1.Gender.*; public class Example_1 { public static void main(String[] args) { List<Person> people = List.of(new Person("Habib",MALE), new Person("Ravi", FEMALE), new Person("Hamim",MALE), new Person("Hena",FEMALE)); //Imperative Approach List<Person> females=new ArrayList<>(); for (Person person : people) { if(FEMALE.equals(person.gender)) { females.add(person); } } } }</pre>	

<pre> for (Person female : females) { System.out.println(female); } static class Person { private final String name; private final Gender gender; public Person(String name, Gender gender) { this.name = name; this.gender = gender; } @Override public String toString() { return "Person [name=" + name + ", gender=" + gender + "]"; } } enum Gender { MALE, FEMALE; } } </pre>	
OUTPUT : Person [name=Ravi, gender=FEMALE] Person [name=Hena, gender=FEMALE]	

In the next example we will see how example_1 can be minimized using Java functional programming. Which will be much more efficient than a normal Java program.

Example_1	Lambda Expression Example : Declarative Approach
<pre> package java_declarative_example; import java.util.List; import java.util.stream.Collectors; import static java_declarative_example.Example_1.Gender.*; public class Example_1 { public static void main(String[] args) { List<Person> people = List.of(new Person("Habib",MALE), new Person("Ravi", FEMALE), new Person("Hamim",MALE), new Person("Hena",FEMALE)); List<Person> females2=people.stream() .filter(person->FEMALE.equals(person.gender)) .collect(Collectors.toList()); females2.forEach(System.out::println); } static class Person { private final String name; private final Gender gender; } </pre>	<div>output</div> <p>The only difference is that in the previous example we used the normal Java function but here we have used the Java functional programming.</p> 

<pre> public Person(String name, Gender gender) { this.name = name; this.gender = gender; } @Override public String toString() { return "Person [name=" + name + ", gender=" + gender + "];" } enum Gender { MALE, FEMALE; } </pre>	
<p>Output:</p> <pre> Person [name=Ravi, gender=FEMALE] Person [name=Hena, gender=FEMALE] </pre>	

Function:

Example_1	Lambda Expression Example : Normal_Java_Function	
<pre> package java_function; public class Example_1 { public static void main(String[] args) { int increment=<i>increment</i>(1); System.out.println(<i>increment</i>); } static int <i>increment</i>(int number) { return number+1; } } </pre> <p>Output: 2</p>	output	<p>In this example, we will learn to increment a number using a simple Java function. So first let's declare a function inside the class but outside the main method. Whose name is increment. The main purpose of using this function is that no matter what number we give, if we use this function, the output will be increased by one.</p> <p>The main function of the Java function is to display the output based on a single input.</p> <p>In the next example we will do the same program using the function of Java 8.</p>

Example_2	Lambda Expression Example : Java_Function	
<pre> package java_function_example; import java.util.function.*; public class Example_2 { public static void main(String[] args) { int output1 = <i>incremntByOne</i>.apply(1); System.out.println(output1); } static Function<Integer, Integer> <i>incremntByOne</i> = number -> number + 1; } </pre>	output	<pre> 2 Function<T,R> Function Takes 1 argument and produces 1 Result Function<Integer, Integer> Here First Integer Mean Data Type Second Integer Represents a results </pre>

Example_1	Java_Function and Then
<pre> package java_function_example; import java.util.function.*; public class Example_2 { public static void main(String[] args) { int output1 = <i>incrementByOne</i>.apply(1); System.out.println(output1); } static Function<Integer, Integer> <i>incrementByOne</i> = number -> number + 1; } </pre>	output 2 Function<T,R> Function<Integer, Integer> Here First Integer Mean Data Type Second Integer Represents a results

Example_3	Lambda Expression Example : Java_Function
<p>In this example we learn , after increasing a number , how we can use that number multiplication by using function. Also in this example we will learn , how to use two method used by andThen Function .</p>	
<pre> package java_function_example; import java.util.function.*; public class Example_3 { public static void main(String[] args) { int increment1 = <i>incrementByOne</i>.apply(1); System.out.println(increment1); int multiply = <i>multiplyBy10</i>.apply(increment1); System.out.println(multiply); // andThen Function uses Function<Integer, Integer> increment1AndThenMuliBy10 = <i>incrementByOne</i>.andThen(<i>multiplyBy10</i>); int andthenoutput = increment1AndThenMuliBy10.apply(4); System.out.println(andthenoutput); } static Function<Integer, Integer> <i>incrementByOne</i> = number -> { return number + 1; }; static Function<Integer, Integer> <i>multiplyBy10</i> = number -> number * 10; } </pre>	
Output is: 2 20 50	

BiFunction:

Function and BiFunction is same . but have simple Difference , Function Takes 1 arguments and produces 1 result. And BiFunction Takes 2 arguments and produces 1 results.

Syntax: BiFunction<T,U,R> ; Here T is First and U is Second input and R is produces output.

Example_4

Lambda Expression Example : Java_Function

```

package java_function_example;
import java.util.function.*;

public class Example_4 {

    public static void main(String[] args) {

        int output1=increment1.apply(1);
        System.out.println("after increment: "+output1);

        int output2=multiby10.apply(output1);
        System.out.println("output1 multiplication :"+output2);

        //BiFunction takes 2 arguments and produces 1 result
        int output3=bifunction.apply(4, 100);
        System.out.println("BiFunction Output:"+output3);
    }

    static Function<Integer,Integer> increment1=number->number+1;
    static Function<Integer,Integer> multiby10=number->number*10;
    static BiFunction <Integer,Integer,Integer> bifunction=(input1,input2)->(input1+1)*input2;
}

```

Output is:

```

after increment: 2
output1 multiplication :20
BiFunction Output:500

```

Consumer:

Java *Consumer* is a functional interface which represents an operation that accepts a single input argument and returns no result. Consumer is different other Functional Interface. It accepts only data.

Example_1

Lambda Expression Example : Java_Function

```

package consumer_biconsumer_example;

public class Example_1 {
    public static void main(String[] args) {
        Customer cust_value = new Customer("Wornoz", "01900000000");
        CustomerFunction(cust_value);
    }

    static class Customer {
        private final String CustomerName;
        private final String CustomerPhoneNumber;

        public Customer(String customerName, String customerPhoneNumber) {
            CustomerName = customerName;
            CustomerPhoneNumber = customerPhoneNumber;
        }
    }

    static void CustomerFunction(Customer customer) {
        System.out.println("Hello " + customer.CustomerName + ", "
            + "Thanks Registering for Java By Phone Number "
            + customer.CustomerPhoneNumber);
    }
}

```

Output is:

Hello Wornoz, Thanks Registering for Java By Phone Number 01900000000

This program will be a normal java function program. We will do this same program using the Consumer Functional Interface. Which is a kind of functional programming. Now we see next example Example_2.

Example_2

Consumer

```

package consumer_biconsumer_example;
import java.util.function.Consumer;
public class Example_2 {
    public static void main(String[] args) {
        Customer cust_value = new Customer("Wornoz", "01900000000");
        CustomerFunction.accept(cust_value);
    }

    static class Customer {
        private final String CustomerName;
        private final String CustomerPhoneNumber;

        public Customer(String customerName, String customerPhoneNumber) {
            CustomerName = customerName;
            CustomerPhoneNumber = customerPhoneNumber;
        }
    }

    static Consumer<Customer> CustomerFunction =
        customer->System.out.println("Hello "
            +customer.CustomerName+
            ",Thanks Registering for Java By Phone Number "
            +customer.CustomerPhoneNumber);
}

```

Output:

Hello Wornoz, Thanks Registering for Java By Phone Number 01900000000

BiConsumer:

Consumer and BiConsumer almost same but have a simple difference. In consumer interface we have receipt only one argument but do not produces any output. In BiConsumer interface we receipt value and at a same time we can justify the value using "Boolean"..Now we show BiConsumer Example. Here we use Example_2

Example_2

BiConsumer

```

package consumer_biconsumer_example;
import java.util.function.BiConsumer;

public class Example_3 {

    public static void main(String[] args) {
        Customer cust_value = new Customer("Wornoz", "01900000000");

        biconsumerFun.accept(cust_value, true);
    }
}

```

```

static BiConsumer<Customer,Boolean> biconsumerFun=
    (customer,showPhone)->
    System.out.println("Hello "
    +customer.Cust_Name+", "
    + "Thanks Registering for java by number "
    +(showPhone?customer.Cust_Phone:"*****"));

static class Customer {
    private final String Cust_Name;
    private final String Cust_Phone;

    public Customer(String cust_Name, String cust_Phone) {
        Cust_Name = cust_Name;
        Cust_Phone = cust_Phone;
    }
}

```

Hello Wornoz, Thanks Registering for java by number 01900000000 if
biconsumerFun.accept(cust_value, **true**); is true

Hello Wornoz, Thanks Registering for java by number *****
biconsumerFun.accept(cust_value, **false**);

Predicate:

Predicate is Functional Interface . It is accepts an argument and a returns Boolean valued. Such as : true or false. Mainly it is used for checking the condition and value for a collection type object.

Firstly, we see normal java program which is used for checked the condition. Now we see example_1.

Example_1	Normal Java Program that is used for checked condition
<pre> package predicate_example; public class Example_1 { public static void main(String[] args) { System.out.println(isNumberValid("01720128755")); System.out.println(isNumberValid("01920128755")); } static boolean isNumberValid(String number) { return number.startsWith("017") && number.length() == 11; } } </pre>	<pre> true false </pre>

Next Example this same program we will do using Predicate.

Example_2	Functional Programming : using Predicate
<pre> package predicate_example; import java.util.function.Predicate; public class Example_2 { public static void main(String[] args) { System.out.println(isNumberValid.test("01720128755")); System.out.println(isNumberValid.test("01920128755")); } static Predicate<String> <i>isNumberValid</i>=number-> number.startsWith("017") && number.length()==11; } </pre>	<pre> true false </pre>

Supplier:

The Supplier interface represents an operation that takes no arguments but returns a result. For this work , have to use get() method.

Firstly , we see normal java program.

Example_1	Normal java program	
<pre>package supplier_example; public class Example_1 { public static void main(String[] args) { GetUrlConnect(); } static String GetUrlConnect() { return "jdbc://localhost::5432/user"; } }</pre>		
		Output
		jdbc://localhost::5432/user

Now we will see same example using supplier:

Example_2	Functional Program : with Supplier	
<pre>package supplier_example; import java.util.List; import java.util.function.Supplier; public class Example_2 { public static void main(String[] args) { System.out.println(getUrlConnection1.get()); System.out.println(getUrlConnection2.get()); } static Supplier<String> getUrlConnection1 = () -> "jdbc://localhost::3455/customers"; static Supplier<List<String>> getUrlConnection2 = () ->List.of("jdbc://localhost::3456/users", "jdbc://localhost::3455/customers"); }</pre>	Output	<pre>jdbc://localhost::3455/customers [jdbc://localhost::3456/users, jdbc://localhost::3455/customers]</pre>

Stream:

Java provides a new additional package in java 8 called java.util.stream. This Package consists of classes , interfaces and enum to allow functional style operation on the elements. We can use stream by importing java.util.stream package.

Stream does not store elements. It easily collect data from such as data structure , array or I/O Channel , and also that collected data through a pipeline of computational operations.

Stream is functional in nature. It can not modify the source . without removing collectors data, we can filter the elements.

It is lazy and evaluates code only when required.

We can use stream to filter , collect , print and convert one data structure to other because of stream is a more powerful function.

Example_1	Functional Programming : using stream
<pre>package streams; import java.util.List; import java.util.stream.Collectors; import static streams.Example_1.Gender.*; public class Example_1 { public static void main(String[] args) { List<Person> people = List.of(new Person("Mizan", MALE), new Person("Joti", FEMALE), new Person("Manik", MALE), new Person("Himel", MALE), new Person("Himu", FEMALE), new Person("Nodi", PREFER_NOT_TO_SAY)); /*-System1-*/ System.out.println("---different types Gender---"); people.stream() .map(person -> person.gender) .collect(Collectors.toSet()) .forEach(gender -> System.out.println(gender)); // others way .forEach(System.out::println); /*---*/ /*-System2-*/ System.out.println("---People Names---"); people.stream() .map(person->person.name) .collect(Collectors.toList()) .forEach(System.out::println); /*-System2-*/ System.out.println("--name sorting by ascending---"); people.stream().map(person->person.name).sorted() .forEach(System.out::println); System.out.println("----we count every name length----"); people.stream() .map(person -> person.name) .mapToInt(name -> name.length()) .forEach(System.out::println); } static class Person { private final String name; private final Gender gender; public Person(String name, Gender gender) { this.name = name; this.gender = gender; } } }</pre>	

```
enum Gender {
    MALE, FEMALE, PREFER_NOT_TO_SAY;
}
}
```

Output:

```
---different types Gender---
PREFER_NOT_TO_SAY
MALE
FEMALE
---People Names---
Mizan
Joti
Manik
Himel
Himu
Nodi
--name sorting by ascending---
Himel
Himu
Joti
Manik
Mizan
Nodi
----we count every name length----
5
4
5
5
4
4
```

Another Program Example_2

```
package streams;
import java.util.List;
import java.util.stream.Collectors;

public class Example_2 {
    public static void main(String[] args) {
        List<Person> people = List.of(
            new Person("Wornoz", 20000),
            new Person("Qurishe", 25000),
            new Person("Omi", 15000),
            new Person("Sonia", 18000),
            new Person("Jobayeer", 12000));
        people.stream().filter(p -> p.salary > 20000).map(p -> p.name).collect(Collectors.toList())
            .forEach(System.out::println);
    }
    static class Person {
        private final String name;
        private final double salary;

        public Person(String name, double salary) {
            this.name = name;
            this.salary = salary;
        }
    }
}
```

output: Qurishe

Optional:

Optional is a container object which is used to contain not-null objects. Optional objects null value with absent of value . this is class has different type of methods. Also it is used for NullPointerException.

Example_1	Optional
<pre>package java_optionals_example; import java.util.Optional; public class Example_1 { public static void main(String[] args) { //case:1 Optional<Object> val1=Optional.empty(); System.out.println("Case1 Output : " +val1); //Case:2 Optional<Object> val2=Optional.of(10); System.out.println("Case2 Output : " +val2); //Case:3 Optional<Object> val3=Optional.ofNullable(null); System.out.println("Case3 Output : " +val3); //Case:4 Object val4=Optional.ofNullable("optional program").orElse("default 1"); System.out.println("Case4 Output : " +val4); //if we use null value then it will be show "default" //Case:5 Object val5=Optional.ofNullable(null).orElseGet(()->" default 2"); System.out.println("Case5 Output : " +val5); //Case:6 //Object val6=Optional.ofNullable(null).orElseThrow(()-> new IllegalStateException("exception")); //System.out.println(val6); //in here case:6 has a problem , as a result case7 can not execute. //but we need case 7 //how we can solve this problem? if we solve case6 problem then //we have to use optional methods. //now we see case : 6 alternative. //Case:6 problem solving by optional method Object val6_Alter=Optional.ofNullable(null).orElseGet(()->"Case 6 , removing nullpointer exception"); System.out.println(val6_Alter); //Case :7 Optional.ofNullable("Hi").ifPresent(System.out::println); } }</pre>	

Output:

Case1 Output : Optional.empty
 Case2 Output : Optional[10]
 Case3 Output : Optional.empty
 Case4 Output : optional program
 Case5 Output : default 2
 Case 6 , removing nullpointer exception
 Hi

Example_2**Functional Program : with Supplier**

```
package java_optionals_example;
import java.util.Optional;

public class Example_2 {

    public static void main(String[] args) {
        // case 1:
        Optional.ofNullable("wornoz@gmail.com")
            .ifPresent(System.out::println);

        // case 2:
        Optional.ofNullable("wornoz@gmail.com")
            .ifPresentOrElse((email) -> System.out.println("sending email to " + email),
                () -> {
                    System.out.println("can not send email");
                });

        // case 3:
        Optional.ofNullable(null)
            .ifPresentOrElse((email) -> System.out.println("sending email to " + email),
                () -> {
                    System.out.println("can not send email");
                });
    }
}
```

Output:

wornoz@gmail.com
 sending email to wornoz@gmail.com
 can not send email

Example_3**Functional Program : stream**

```
package java_optionals_example;
import java.util.List;
import java.util.function.Predicate;
import static java_optionals_example.Example_3.Gender.*;

public class Example_3 {

    public static void main(String[] args) {
        List<Person> people = List.of(
            new Person("Wornoz", MALE),
            new Person("Laila", FEMALE),
            new Person("Qurishe", MALE));

        Predicate<Person> person_pre = person -> FEMALE.equals(person.gender);
    }
}
```

```
        boolean container_only_female = people.stream().anyMatch(person_pre);
        System.out.println(container_only_female);

    }

    static class Person {
        private final String name;
        private final Gender gender;

        public Person(String name, Gender gender) {
            this.name = name;
            this.gender = gender;
        }
    }

    enum Gender {
        MALE, FEMALE;
    }
}
```

Output:

true