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. |
| **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);  }  }    **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***;  }  } | |
| **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 | |
| **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;  **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***;  }  } | | output |
| The only difference is that in the previous example we used the normal Java function but here we have used the Java functional programming. |
| Output:  Person [name=Ravi, gender=FEMALE]  Person [name=Hena, gender=FEMALE] | |  |

Function:

|  |  |  |
| --- | --- | --- |
| Example\_1 | Lambda Expression Example : Normal\_Java\_Function | |
| **package** java\_function;  **public** **class** Example\_1 {  **public** **static** **void** main(String[] args) {  **int** increment=*increment*(1);  System.***out***.println(increment);  }    **static** **int** increment(**int** number) {  **return** number+1;  }  }  Output:  2 | | output |
| 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.  The main function of the Java function is to display the output based on a single input.  In the next example we will do the same program using the function of Java 8. |

|  |  |  |
| --- | --- | --- |
| Example\_2 | Lambda Expression Example : Java\_Function | |
| **package** java\_function\_example;  **import** java.util.function.\*;  **public** **class** Example\_2 {  **public** **static** **void** main(String[] args) {  **int** output1 = *incremntByOne*.apply(1);  System.***out***.println(output1);  }  **static** Function<Integer, Integer> *incremntByOne* = number -> number + 1;  } | | output |
| 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 |

|  |  |  |
| --- | --- | --- |
| Example\_1 | Java\_Function and Then | |
| **package** java\_function\_example;  **import** java.util.function.\*;  **public** **class** Example\_2 {  **public** **static** **void** main(String[] args) {  **int** output1 = *incremntByOne*.apply(1);  System.***out***.println(output1);  }  **static** Function<Integer, Integer> *incremntByOne* = number -> number + 1;  } | | 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 |
| 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 . | |
| **package** java\_function\_example;  **import** java.util.function.\*;  **public** **class** Example\_3 {  **public** **static** **void** main(String[] args) {  **int** increment1 = *incrementByOne*.apply(1);  System.***out***.println(increment1);  **int** multiply = *multiplyBy10*.apply(increment1);  System.***out***.println(multiply);  // andThen Function uses  Function<Integer, Integer> increment1AndThenMuliBy10 = *incrementByOne*.andThen(*multiplyBy10*);  **int** andthenoutput = increment1AndThenMuliBy10.apply(4);  System.***out***.println(andthenoutput);  }  **static** Function<Integer, Integer> *incrementByOne* = number -> {  **return** number + 1;  };  **static** Function<Integer, Integer> *multiplyBy10* = number -> number \* 10;  } | |
| **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 | |
| **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;  }  } | | Output |
| true  false |

Next Example this same program we will do using Predicate.

|  |  |  |
| --- | --- | --- |
| Example\_2 | Functional Programming : using Predicate | |
| **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> *isNumberValid*=number->  number.startsWith("017") && number.length()==11;  } | | Output |
| true  false |

**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 | |
| **package** supplier\_example;  **public** **class** Example\_1 {  **public** **static** **void** main(String[] args) {  *GetUrlConnect*();  }  **static** String GetUrlConnect() {  **return** "jdbc://localhost::5432/user";  }  } | | Output |
| jdbc://localhost::5432/user |

Now we will see same example using supplier:

|  |  |  |
| --- | --- | --- |
| Example\_2 | Functional Program : with Supplier | |
| **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"  );  } | | Output |
| jdbc:://localhost::3455/customers  [jdbc:://localhost::3456/users, jdbc:://localhost::3455/customers] |

**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 |
| **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;  }  }  **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) { output: Qurishe  **this**.name = name;  **this**.salary = salary;  }  }  } | |

**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 |
| **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);    }  } | |
| **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 | |