# Java Lambda Expressions

It provides a clear and concise way to represent one method interface using an expression. It is very useful in collection library. It helps to iterate, filter and extract data from collection.

The Lambda expression is used to provide the implementation of an interface which has functional interface. It 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.

## **Functional Interface**

Lambda expression provides implementation of functional interface. An interface which has only one abstract method is called functional interface. Java provides an anotation @FunctionalInterface, which is used to declare an interface as functional interface.

## **Why use Lambda Expression**

1. To provide the implementation of Functional interface.
2. Less coding.

## **Java Lambda Expression Syntax**

1. (argument-list) -> {body}

****No Parameter Syntax****

1. () -> {
2. //Body of no parameter lambda
3. }

****One Parameter Syntax****

1. (p1) -> {
2. //Body of single parameter lambda
3. }

****Two Parameter Syntax****

1. (p1,p2) -> {
2. //Body of multiple parameter lambda
3. }

## **Without Lambda Expression**

**interface** Drawable{

**public** **void** draw();

}

**public** **class** LambdaExpressionExample {

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

**int** width=10;

        //without lambda, Drawable implementation using anonymous class

        Drawable d=**new** Drawable(){

**public** **void** draw(){System.out.println("Drawing "+width);}

        };

        d.draw();

    }

}

Output:

Drawing 10

## **Java Lambda Expression Example**

Now, we are going to implement the above example with the help of Java lambda expression.

@FunctionalInterface  //It is optional

**interface** Drawable{

**public** **void** draw();

}

**public** **class** LambdaExpressionExample2 {

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

**int** width=10;

        //with lambda

        Drawable d2=()->{

            System.out.println("Drawing "+width);

        };

        d2.draw();

    }

}

Output:

Drawing 10

## **Java Lambda Expression Example: Multiple Parameters**

**interface** Addable{

**int** add(**int** a,**int** b);

}

**public** **class** LambdaExpressionExample5{

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

        // Multiple parameters in lambda expression

        Addable ad1=(a,b)->(a+b);

        System.out.println(ad1.add(10,20));

        // Multiple parameters with data type in lambda expression

        Addable ad2=(**int** a,**int** b)->(a+b);

        System.out.println(ad2.add(100,200));

    }

}

Output:

30

300

## **Java Lambda Expression Example: Foreach Loop**

**import** java.util.\*;

**public** **class** LambdaExpressionExample7{

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

        List<String> list=**new** ArrayList<String>();

        list.add("ankit");

        list.add("mayank");

        list.add("irfan");

        list.add("jai");

        list.forEach(

            (n)->System.out.println(n)

        );

    }

}

Output:

ankit

mayank

irfan

jai

**Java lambda expression can be used in the collection framework. It provides efficient and concise way to iterate, filter and fetch data.**

# Java Functional Interfaces

An Interface that contains exactly one abstract method is known as functional interface.

Methods which are defined inside the interface and tagged with default are known as default methods.

StringJoiner - you can create string by passing delimiters like comma(,), hyphen(-) etc. You can also pass prefix and suffix to the char sequence.

# Java Optional Class

Java introduced a new class Optional in jdk8. It is a public final class and used to deal with NullPointerException in Java application.

### Example: Java Program without using Optional

In the following example, we are not using Optional class. This program terminates abnormally and throws a nullPointerException.

1. **public** **class** OptionalExample {
2. **public** **static** **void** main(String[] args) {
3. String[] str = **new** String[10];
4. String lowercaseString = str[5].toLowerCase();
5. System.out.print(lowercaseString);
6. }
7. }

Output:

*Exception in thread "main" java.lang.NullPointerException*

*at lambdaExample.OptionalExample.main(OptionalExample.java:6)*

To avoid the abnormal termination, we use Optional class. In the following example, we are using Optional. So, our program can execute without crashing.

### Java Optional Example: If Value is not Present

1. **import** java.util.Optional;
2. **public** **class** OptionalExample {
3. **public** **static** **void** main(String[] args) {
4. String[] str = **new** String[10];
5. Optional<String> checkNull = Optional.ofNullable(str[5]);
6. **if**(checkNull.isPresent()){  // check for value is present or not
7. String lowercaseString = str[5].toLowerCase();
8. System.out.print(lowercaseString);
9. }**else**
10. System.out.println("string value is not present");
11. }
12. }

Output:

*string value is not present*

# Java 8 Stream

This package consists of classes, interfaces and enum to allows functional-style operations on the elements.

Stream provides following features:

* Stream is functional in nature. Operations performed on a stream does not modify it's source. For example, filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection.
* Stream is lazy and evaluates code only when required.
* The elements of a stream are only visited once during the life of a stream. Like an Iterator, a new stream must be generated to revisit the same elements of the source.

You can use stream to filter, collect, print, and convert from one data structure to other etc. In the following examples, we have apply various operations with the help of stream.

### **Java Stream Example: Filtering Collection by using Stream**

Here, we are filtering data by using stream. You can see that code is optimized and maintained. Stream provides fast execution.

**import** java.util.\*;

**import** java.util.stream.Collectors;

**class** Product{

**int** id;

    String name;

**float** price;

**public** Product(**int** id, String name, **float** price) {

**this**.id = id;

**this**.name = name;

**this**.price = price;

    }

}

**public** **class** JavaStreamExample {

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

        List<Product> productsList = **new** ArrayList<Product>();

        //Adding Products

        productsList.add(**new** Product(1,"HP Laptop",25000f));

        productsList.add(**new** Product(2,"Dell Laptop",30000f));

        productsList.add(**new** Product(3,"Lenevo Laptop",28000f));

        productsList.add(**new** Product(4,"Sony Laptop",28000f));

        productsList.add(**new** Product(5,"Apple Laptop",90000f));

        List<Float> productPriceList2 =productsList.stream()

                                     .filter(p -> p.price > 30000)// filtering data

                                     .map(p->p.price)        // fetching price

                                     .collect(Collectors.toList()); // collecting as list

        System.out.println(productPriceList2);

    }

}

**Output:**

[90000.0]

### **Java Stream Iterating Example**

You can use stream to iterate any number of times. Stream provides predefined methods to deal with the logic you implement. In the following example, we are iterating, filtering and passed a limit to fix the iteration.

**import** java.util.stream.\*;

**public** **class** JavaStreamExample {

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

        Stream.iterate(1, element->element+1)

        .filter(element->element%5==0)

        .limit(5)

        .forEach(System.out::println);

    }

}

**Output:**

5

10

15

20

25

  // This is more compact approach for filtering data

        Float totalPrice = productsList.stream()

                    .map(product->product.price)

                    .reduce(0.0f,(sum, price)->sum+price);   // accumulating price

        System.out.println(totalPrice);

// count number of products based on the filter

**long** count = productsList.stream()

                    .filter(product->product.price<30000)

                    .count();

        System.out.println(count);

1. / Converting product List into Set
2. Set<Float> productPriceList =
3. productsList.stream()
4. .filter(product->product.price < 30000)   // filter product on the base of price
5. .map(product->product.price)
6. .collect(Collectors.toSet());   // collect it as Set(remove duplicate elements)
7. System.out.println(productPriceList);

   // Converting Product List into a Map

        Map<Integer,String> productPriceMap =

            productsList.stream()

                        .collect(Collectors.toMap(p->p.id, p->p.name));

        System.out.println(productPriceMap);

# Java forEach loop

Java provides a new method forEach() to iterate the elements. It is defined in Iterable and Stream interface. It is a default method defined in the Iterable interface. Collection classes which extends Iterable interface can use forEach loop to iterate elements.

This method takes a single parameter which is a functional interface. So, you can pass lambda expression as an argument.

## **forEach() Signature in Iterable Interface**

1. **default** **void** forEach(Consumer<**super** T>action)

### **Java 8 forEach() example 1**

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** ForEachExample {

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

        List<String> gamesList = **new** ArrayList<String>();

        gamesList.add("Football");

        gamesList.add("Cricket");

        gamesList.add("Chess");

        gamesList.add("Hocky");

        System.out.println("------------Iterating by passing lambda expression--------------");

        gamesList.forEach(games -> System.out.println(games));

    }

}

Output:

------------Iterating by passing lambda expression--------------

Football

Cricket

Chess

Hocky

*Iterating by passing method reference*

gamesList.forEach(System.out::println);   same results as above.

Java provides one more method forEachOrdered(). It is used to iterate elements in the order specified by the stream.

1. Lamda expressions and function & method reference ?
2. Streams in java
3. Optional class and ForEach?