## Java 8 features with examples

## 1. Java 8 – Lambda Expression

## 2. Java 8 – Method references

## 3. Java 8 – Functional interfaces

## 4. Java 8 – Interface changes: Default and static methods

## 5. Java 8 – Streams

## 6. Java 8 – Stream filter

## 7. Java 8 – forEach()

Lambda expression is a new and important feature of Java which was included in Java SE 8. It is very useful in collection library. It helps to iterate, filter and extract data from collection.

 A lambda expression is an anonymous function. A function that doesn’t have a name and doesn’t belong to any class. The concept of lambda expression was first introduced in LISP programming language. It comprises of a set of parameters, a lambda operator (->) and a function body.

## Why use Lambda Expression

* To provide the implementation of Functional interface(A functional interface is an interface that contains only one abstract method)

for example: Runnable,ActionListener etc.

* Less coding.

## Java Lambda Expression Syntax

To create a lambda expression, we specify input parameters (if there are any) on the left side of the lambda operator ->, and place the expression or block of statements on the right side of lambda operator. For example, the lambda expression (x, y) -> x + y specifies that lambda expression takes two arguments x and y and returns the sum of these.

(argument-list) -> {body}

The syntax of a lambda expression is an argument list enclosed in parentheses, an arrow token (->), and a function body. The body can be either a statement block (enclosed in braces) or a single expression.

Java lambda expression is consisted of three components.

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

2) Arrow-token: It is used to link arguments-list and body of expression.

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

## Lambda expression vs method in Java

A method (or function) in Java has these main parts:  
1. Name  
2. Parameter list  
3. Body  
4. return type.

A lambda expression in Java has these main parts:  
Lambda expression **only has body and parameter list**.  
1. **No** name – function is anonymous so we don’t care about the name  
2. Parameter list  
3. Body – This is the main part of the function.  
4. **No** return type – The java 8 compiler is able to infer the return type by checking the code. you need not to mention it explicitly

## Java Lambda expression Example

**Without using Lambda expression:** Prior to java 8 we used the anonymous inner classe to implement the only abstract method of functional interface.

import java.awt.\*;

import java.awt.event.\*;

public class ButtonListenerOldWay {

public static void main(String[] args) {

Frame frame=new Frame("ActionListener Before Java8");

Button b=new Button("Click Here");

b.setBounds(50,100,80,50);

b.addActionListener(new ActionListener(){

public void actionPerformed(ActionEvent e){

System.out.println("Hello World!");

}

});

frame.add(b);

frame.setSize(200,200);

frame.setLayout(null);

frame.setVisible(true);

}

}

**By using Lambda expression:** Instead of creating anonymous inner class, we can create a lambda expression like this:

import java.awt.\*;

public class ButtonListenerNewWay {

public static void main(String[] args) {

Frame frame=new Frame("ActionListener java8");

Button b=new Button("Click Here");

b.setBounds(50,100,80,50);

b.addActionListener(e -> System.out.println("Hello World!"));

frame.add(b);

frame.setSize(200,200);

frame.setLayout(null);

frame.setVisible(true);

}

}

Note:

1. As you can see that we used less code with lambda expression.

2. Backward compatibility: You can use the lambda expression with your old code. Lambdas are backward compatible so you can use them in existing API when you migrate your project to java 8.

**Example: Iterating collections using foreach loop**

import java.util.\*;

class LambdaExpressionExample{

public static void main(String[] args) {

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

list.add("Samsung A5");

list.add("Iphone 6S");

// using lambda to iterate through collection

list.forEach(

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

);

}

}

## Example:

import java.util.ArrayList;

class Lambda2ndExample

{

public static void main(String args[])

{

ArrayList<Integer> arrL = new ArrayList<Integer>();

arrL.add(1);

arrL.add(2);

arrL.add(3);

arrL.add(4);

// Using lambda expression to print all elements

// of arrL

arrL.forEach(n -> System.out.println(n));

// Using lambda expression to print even elements

// of arrL

arrL.forEach(n -> { if (n%2 == 0) System.out.println(n); });

}

}

----------------------------------------------------------------------------------------

Let's see a scenario. If we don't implement Java lambda expression. Here, we are implementing an interface method without using lambda expression.

## 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();

}

}

Test it Now

Output:

Drawing 10

## Java Lambda Expression Example

Now, we are implementing the above example with the help of 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();

}

}

Test it Now

Output:

Drawing 10

A lambda expression can have zero or any number of arguments. Let's see the examples:

## Java Lambda Expression Example: No Parameter

interface Sayable{

public String say();

}

public class LambdaExpressionExample3{

public static void main(String[] args) {

Sayable s=()->{

return "I have nothing to say.";

};

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

}

}

Test it Now

Output:

I have nothing to say.

## Java Lambda Expression Example: Single Parameter

interface Sayable{

public String say(String name);

}

public class LambdaExpressionExample4{

public static void main(String[] args) {

// Lambda expression with single parameter.

Sayable s1=(name)->{

return "Hello, "+name;

};

System.out.println(s1.say("Sanjoy"));

// You can omit function parentheses

Sayable s2= name ->{

return "Hello, "+name;

};

System.out.println(s2.say("Dutta"));

}

}

Test it Now

Output:

Hello, Sanjoy

Hello, Dutta

## 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));

}

}

o/p:

30

300

## Java Lambda Expression Example: Creating Thread:

You can use lambda expression to run thread. In the following example, we are implementing run method by using lambda expression.

public class LambdaExpressionExample9{

public static void main(String[] args) {

//Thread Example without lambda

Runnable r1=new Runnable(){

public void run(){

System.out.println("Thread1 is running...");

}

};

Thread t1=new Thread(r1);

t1.start();

//Thread Example with lambda

Runnable r2=()->{

System.out.println("Thread2 is running...");

};

Thread t2=new Thread(r2);

t2.start();

}

}

## Java Lambda Expression Example: Filter Collection Data

import java.util.ArrayList;

import java.util.List;

import java.util.stream.Stream;

class Product{

int id;

String name;

float price;

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

super();

this.id = id;

this.name = name;

this.price = price;

}

}

public class LambdaExpressionExample11{

public static void main(String[] args) {

List<Product> list=new ArrayList<Product>();

list.add(new Product(1,"Samsung A5",17000f));

list.add(new Product(3,"Iphone 6S",65000f));

list.add(new Product(2,"Sony Xperia",25000f));

list.add(new Product(4,"Nokia Lumia",15000f));

list.add(new Product(5,"Redmi4 ",26000f));

list.add(new Product(6,"Lenevo Vibe",19000f));

// using lambda to filter data

Stream<Product> filtered\_data = list.stream().filter(p -> p.price > 20000);

// using lambda to iterate through collection

filtered\_data.forEach(

product -> System.out.println(product.name+": "+product.price)

);

}

}

## Stream API:

Stream API which is another new feature of java 8. All the classes and interfaces of this API is in the java.util.stream package. By using streams we can perform various aggregate operations on the data returned from collections, arrays, Input/Output operations. Before we see how stream API can be used in Java, let’s see an example to understand the use of streams.

## Java Stream Example

To understand how stream works, lets take an example without using stream and then we will see the same example with streams.

## Finding certain strings without using Stream

import java.util.ArrayList;

import java.util.List;

public class Example{

public static void main(String[] args) {

List<String> names = new ArrayList<String>();

names.add("Ajeet");

names.add("Negan");

names.add("Aditya");

names.add("Steve");

int count = 0;

for (String str : names) {

if (str.length() < 6)

count++;

}

System.out.println("There are "+count+" strings with length less than 6");

}

}

Output:

There are 3 strings with length less than 6

## Same example using Stream:

import java.util.ArrayList;

import java.util.List;

public class Example{

public static void main(String[] args) {

List<String> names = new ArrayList<String>();

names.add("Ajeet");

names.add("Negan");

names.add("Aditya");

names.add("Steve");

//Using Stream and Lambda expression

long count = names.stream().filter(str->str.length()<6).count();

System.out.println("There are "+count+" strings with length less than 6");

}

}

Output:

There are 3 strings with length less than 6

## What is the difference between these codes?

The output of both the examples are same, however there is a major difference between these examples if you consider the performance of the code.

In the first example, we are iterating the whole list to find the strings with length less than 6. There is no parallelism in this code.

In the second example, the stream() method returns a stream of all the names, the filter() method returns another stream of names with length less than 6, the count() method reduces this stream to the result. All these operations are happening parallelly which means we are able to parallelize the code with the help of streams. Parallel execution of operations using stream is faster than sequential execution without using streams.

## How to work with Stream in Java:

As we have seen in the above example, the working of stream can be explained in three stages:

1. Create a stream

2. Perform intermediate operations on the initial stream to transform it into another stream and so on on further intermediate operations. In the above example, the filter() operation is intermediate operation, there can be more than one intermediate operations.

3. Perform terminal operation on the final stream to get the result. In the above example, the count() operation is terminal operation.

## Java Stream Features:

1. Stream does not store the elements. it simply performs the aggregate operations(such as filter() and count() that we have seen in the above example) to get the desired stream of data.

2. The aggregate operations that we perform on the collection, array or any other data source do not change the data of the source, they simply return a new stream. For example the code we have seen above is filtering the strings with length less than 6 using the stream operations but it didn’t change the elements of the list.

3. All the stream operations are lazy in nature which means they are not executed until they are needed. For example, if we want to display only the first 2 elements of a list using stream, the stream operation would stop at the end of second iteration after displaying the second element of list.

## Let’s see few examples of Java Stream:

## Java Stream Example 1: Iterating and displaying selected integers

import java.util.stream.\*;

public class Example {

public static void main(String[] args){

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

.filter(number->number%3==0)

.limit(6)

.forEach(System.out::println);

}

}

Output:

3

6

9

12

15

18

## Java Stream Example 2: Concatenating two streams

import java.util.Arrays;

import java.util.List;

import java.util.stream.Stream;

public class Example {

public static void main(String[] args) {

//list 1

List<String> alphabets = Arrays.asList("A","B","C");

//list 2

List<String> names = Arrays.asList("Sansa","Jon","Arya");

//creating two streams from the two lists and concatenating them into one

Stream<String> opstream = Stream.concat(alphabets.stream(), names.stream());

//displaying the elements of the concatenated stream

opstream.forEach(str->System.out.print(str+" "));

}

}

Output:

A B C Sansa Jon Arya

## Java 8 Stream Filter with examples

The filter() is an intermediate operation that reads the data from a stream and returns a new stream after transforming the data based on the given condition. Lets take a simple example first and then we will see the examples of stream filter with other methods of the stream.

## A Simple Example of Java Stream Filter()

In this example we are creating a stream from the list of names using stream() method and then we are creating another stream of long names using stream filter(). As I mentioned above, the stream filter transforms the data of one stream into another stream.

import java.util.Arrays;

import java.util.List;

import java.util.stream.Stream;

public class Example {

public static void main(String[] args) {

List<String> names = Arrays.asList("Melisandre","Sansa","Jon","Daenerys","Joffery");

//Creating the stream of all names

Stream<String> allNames = names.stream();

//Creating another stream by filtering long names using filter()

Stream<String> longNames = allNames.filter(str -> str.length() > 6);

//displaying the long names

longNames.forEach(str->System.out.print(str+" "));

}

}

Output:

Melisandre Daenerys Joffery

Lets take few more examples of Java stream filter.

## Example 1: Stream filter() and collect()

We can create a stream and apply a filter in a one line as shown in the example below. The collect() method here collects the final stream and converts it into a list

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class Example {

public static void main(String[] args) {

List<String> names = Arrays.asList("Melisandre","Sansa","Jon","Daenerys","Joffery");

List<String> longnames = names.stream() // converting the list to stream

.filter(str -> str.length() > 6) // filter the stream to create a new stream

.collect(Collectors.toList()); // collect the final stream and convert it to a List

longnames.forEach(System.out::println);

}

}

Output:

Melisandre

Daenerys

Joffery

## Example 2: Stream filter() with multiple conditions

In the above examples we have seen that there is only one condition in the filter() method. We can have more than one conditions in the filter() method joined using the logical operators in java. In the following example, we have two conditions in the filter method joined using and (&&) logical operator.

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class Example {

public static void main(String[] args) {

List<String> names = Arrays.asList("Melisandre","Sansa","Jon","Daenerys","Joffery");

List<String> longnames = names.stream()

.filter(str -> str.length() > 6 && str.length() < 8) //Multiple conditions

.collect(Collectors.toList());

longnames.forEach(System.out::println);

}

}

Output:

Joffery

## Example 3: Stream filter() and map() method in Java

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class Example {

public static void main(String[] args) {

List<Integer> num = Arrays.asList(1,2,3,4,5,6);

List<Integer> squares = num.stream()

.map(n -> n \* n)

.collect(Collectors.toList());

System.out.println(squares);

}

}

Output:

[1, 4, 9, 16, 25, 36]

## Java 8 – Filter a Map by keys and values

we will see how to use Stream filter() method to filter a Map by keys and Values.

## Java 8 – Filter Map by Keys

import java.util.Map;

import java.util.HashMap;

import java.util.stream.Collectors;

public class Example {

public static void main(String[] args) {

Map<Integer, String> hmap = new HashMap<Integer, String>();

hmap.put(11, "Apple");

hmap.put(22, "Orange");

hmap.put(33, "Kiwi");

hmap.put(44, "Banana");

Map<Integer, String> result = hmap.entrySet()

.stream()

.filter(map -> map.getKey().intValue() <= 22)

.collect(Collectors.toMap(map -> map.getKey(), map -> map.getValue()));

System.out.println("Result: " + result);

}

}

Output:

Result: {22=Orange, 11=Apple}

## Java 8 – Filter Map by Values

import java.util.Map;

import java.util.HashMap;

import java.util.stream.Collectors;

public class Example {

public static void main(String[] args) {

Map<Integer, String> hmap = new HashMap<Integer, String>();

hmap.put(11, "Apple");

hmap.put(22, "Orange");

hmap.put(33, "Kiwi");

hmap.put(44, "Banana");

Map<Integer, String> result = hmap.entrySet()

.stream()

.filter(map -> "Orange".equals(map.getValue()))

.collect(Collectors.toMap(map -> map.getKey(), map -> map.getValue()));

System.out.println("Result: " + result);

}

}

Output:

Result: {22=Orange}

## Java 8 – Filter Map by both Keys and Values

In this example we are filtering a Map by keys and values both. When we filter a Map like this we are joining both the conditions by AND (&&) logical operator. You can also place both the conditions in the single filter() method and join them using any logical operator such as OR (||), AND(&&) or NOT(!).

import java.util.Map;

import java.util.HashMap;

import java.util.stream.Collectors;

public class Example {

public static void main(String[] args) {

Map<Integer, String> hmap = new HashMap<Integer, String>();

hmap.put(1, "ABC");

hmap.put(2, "XCB");

hmap.put(3, "ABB");

hmap.put(4, "ZIO");

Map<Integer, String> result = hmap.entrySet()

.stream()

.filter(p -> p.getKey().intValue() <= 2) //filter by key

.filter(map -> map.getValue().startsWith("A")) //filter by value

.collect(Collectors.toMap(map -> map.getKey(), map -> map.getValue()));

System.out.println("Result: " + result);

}

}

Output:

Result: {1=ABC}