Java 1.8 Features

**Date 02/01/2023**

# Introduction to Java 1.8 - 14th Mar 2014(Reason for Java 1.8 version)

1. To simplify programming.
2. To get the functional/procedural programming benefits using Lambda expression.
3. To enable parallel processing.

# Lambda Expression

Lambda expression is first used by LISP programming language. And slowly other programming languages like c, c++, objective c, c#.Net, scala, ruby started using it. Later Java and Python also started using the Lambda expression.

# Benefits of Lambda Expression

1. To enable functional programming in Java.
2. To write more readable, maintainable, and concise code.
3. To use APIs very easily and effectively.
4. To enable parallel processing.

# What is Lambda Expression

It is an anonymous function. The functions without name, return type and modifier is called as anonymous function.

# Lambda Expression Writing

Example 1,

public void m1 () {

SOP (“Hello”);

}

~~public void m1~~ () {

SOP (“Hello”);

}

() -> {SOP(“Hello”);}

**() -> SOP(“Hello”);** **//** When we have only one statement inside curly braces, then the braces are optional.

Example 2,

public void m1 (int a, int b) {

SOP (a+b);

}

**(a, b) -> SOP (a+b);** **//** In few scenarios, Compiler can guess the variable type, in that case, we can remove variable type as well.

Example 3,

public int squareIt (int n) {

return n\*n;

}

(int n) -> {return n\*n};

(int n) -> return n\*n;

(int n) -> n\*n;

(n) -> n\*n;

**n -> n\*n;**

Example 4,

public int m1 (String s) {

SOP (s.length());

}

(String s) -> {return s.length();}

(String s) -> return s.length();

(s) -> s.length();

**s -> s.length();**

# Functional Interface

An interface which contains single abstract methods is called as Functional Interface. Functional Interface is used to invoke the Lambda expression.

**Note:** From 1.8 version onwards, a functional interface can have **default and static methods in addition to SINGLE abstract method**. To explicitly mention an interface as Functional Interface an annotation is introduced in 1.8v as **@FunctionalInterface**. The **@FunctionalInterface** is optional.

Example,

interface Interf {

public void m1();

default void m2() {

}

static void m3() {

}

}

**@FunctionalInterface**

interface A {

public void m1();

}

**@FunctionalInterface**

interface B extends A {**// Valid**

}

**@FunctionalInterface**

interface C extends A {**// Valid**

public void m1();

}

**@FunctionalInterface**

interface D extends A {**// Invalid**

public void m2();

}

interface E extends A {**// Valid**

public void m2() {

}

}

# Lambda Expression Case Studies

**Example 1,**

**interface** Interf {

public void m1();

}

class **Demo** implements **Interf** {

public void m1 () {

SOP(“hello”);

}

}

class **Test** {

public static void main (String[] args) {

Demo d = new Demo ();

**//**Interf d = new Demo ();

d.m1();

}

}

The above program can be rewritten in java1.8 style as below,

**interface** Interf {

public void m1();

}

~~class Demo implements Interf {~~

~~public void m1 () {~~

~~SOP(“hello”);~~

~~}~~

~~}~~

**class** Test {

public static void main (String[] args) {

**Interf d = ()->SOP(“hello by Lambda expression”);**

d.m1();

}

}

**Example 2,**

**interface** Interf {

public void add(int a, int b);

}

class **Demo** implements **Interf** {

public void add(int a, int b) {

SOP (“Sum is:” +(a+b));

}

}

class **Test** {

public static void main (String[] args) {

Interf d = new Demo ();

d.add(10,20);

}

}

The above program can be rewritten in java1.8 style as below,

**interface** Interf {

public void add(int a, int b);

}

~~class~~ **~~Demo~~** ~~implements~~ **~~Interf~~** ~~{~~

~~public void add(int a, int b) {~~

~~SOP (“Sum is:” +(a+b));~~

~~}~~

~~}~~

class **Test** {

public static void main (String[] args) {

Interf d = (a, b) -> SOP (“Sum is:” +(a+b));

d.add(10,20);

}

}

**Note:** Wherever Functional Interface is available, there **only** we can use Lambda expression and in other places we can’t use Lambda expression. i.e., Lambda expression is associated with the Functional Interface.

**Example 3,**

**interface** Interf {

public void squareIt(int n);

}

class **Test** {

public static void main (String[] args) {

Interf d = n->n\*n;

SOP (d.squareIt(5));

}

}

## Multi-Threading

class **MyRunnable** implements **Runnable** {

public void run() {

for (int=0; i<10; i++) {

SOP (“Child Thread”);

}

}

class **Test** {

public static void main (String[] args) {

Runnable r = new MyRunnable();

Thread t = new Thread(r);

t.start();

for (int=0; i<10; i++) {

SOP (“Main Thread”);

}

}

}

In Java1.8 style as below,

class **Test** {

public static void main (String[] args) {

**Runnable r = ()-> {**

**for (int=0; i<10; i++) {**

**SOP (“Child Thread”);**

**}**

**};**

Thread t = new Thread(r);

t.start();

for (int=0; i<10; i++) {

SOP (“Main Thread”);

}

}

}

## ArrayList Comparator

import java.util.\*;

import java.util.stream.\*;

class **Test** {

public static void main (String[] args) {

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

l.add(20);

l.add(10);

l.add(25);

l.add(5);

SOP(l);

**Comparator<Integer> c = (I1, I2)->(I1<I2)?:-1:(I1>I2)?1:0;**

Collections.sort(l,c);

SOP(l);

l.stream().forEach(System.out::println);

List<Integer> l2 = l.stream().filter(i->i%2==0).collect(Collectors.toList());

SOP(l2);

}

}

**Output**

[5,10,20,25]

5

10

20

25

## Employee Example

class **Employee** {

int eno;

String name;

Employee (int eno, String name) {

this.eno = eno;

this.name = name;

}

Public String toString() {

return eno+”:”+name;

}

}

class **Test** {

public static void main (String[] args) {

ArrayList<Employee> l = new ArrayList<Employee>();

l.add(new Employee(“John”, 8979);

l.add(new Employee(“Blaze”, 1979);

l.add(new Employee(“Alex”, 2979);}

SOP (l);

Collections.sort(I, (e1,e2)->e1.name.compareTo(e2.name));

SOP (l);

}

}

**Output**

[2979:” Alex”, 1979:” Blaze”, 8979:” John”]

# Anonymous Inner Class vs Lambda Expression

If anonymous inner class implements an Interface and if that contains single abstract method, then only, we can replace that anonymous inner class with lambda expressions.

1. Anonymous inner class can extend a normal class.
2. Anonymous inner class can extend an abstract class.
3. Anonymous inner class can implement an interface which contains any number of abstract methods.

Anonymous Inner Class! = Lambda Expression

Anonymous Inner Class > Lambda Expression

# Default Methods (Virtual Extension Method | Defender Method)

**From Java 1.8v onwards**, we can take default methods inside an Interface.

**Until Java 1.7v**, every method present inside interface is always public and abstract.

void m1();

public void m1();

abstract void m1();

public abstract void m1();

**From Java 1.8v,**

Default methods and Static methods also available inside interface.

**From Java 1.9v,**

Private methods also available inside interface.

**Objective**

If we want add a method without affecting implementation classes of the Interface then we should go for **Default Methods**.

Example,

Interface **I** {

public void m1();

**public void m2(); // Adding new method will affect the implementation class. To avoid this, we should got for Default method to make necessary changes in the Interface.**

**default** void m2() {

SOP (“Default Method”);

}

}

class A **implements** I {

public void m1() {

}

}

class B **implements** I {

public void m1() {

}

}

We can override the functionality of default method present in the Interface if we don’t want the Interface’s default method.

Interface **I** {

**default** void m1() {

SOP (“Default Method”);

}

}

class A **implements** I {

public void m1() {

SOP (“Overriding of Default Method”);

}

public static void main (String[] args) {

A a = new A ();

a.m1();

}

}

Interface **I** {

**default** void m1() {

SOP (“Default Method”);

}

***default int hasCode() {***

***return 10;***

***}***

}

We can’t implement the object class method as default method in Interface. Also, we can’t override the object class method because by default the object class methods available to the implementation class. If we try to override then we will get the Compile time error.

Example,

interface **Left** {

default void m1() {

SOP (“Left m1 method”);

}

}

interface **Right** {

default void m1() {

SOP (“Right m1 method”);

}

}

class **Test** implements Left, Right {

public void m1 () {

SOP (“Own m1 method”);

// Left.super.m1();

// Right.super.m1();

}

public static void main (String[] args) {

Test t = new Test ();

t.m1();

}

}

# Class vs Interface

Class is costly when compared to an Interface as the Interface doesn’t contain constructors, static blocks, instance blocks. Hence, Interface is considered as light weight component.

When we have requirement to have only static methods, then we can go for Interface as static method no way related to object. Hence, we can go for Interface as it was not costly when compared to Class.

Example,

interface **Interf** {

public static void m1 () {

SOP (Interface m1 method);

}

}

class **Test** implements **Interf** {

public static void main (String[] args) {

Interf.m1();

}

}

Interface static methods by default won’t available to the implementation class. So, if we need to invoke the static methods of interface, then we should call by using the interface name.

From Java1.8v onwards, we can take static methods inside Interface.

interface **Interf** {

public static void main (String[] args) {

SOP(“Hello”);

}

}

To define general static utility methods, we can go for interface and not class.

# Predefined Functional Interfaces

1. Predicate
2. Function
3. Consumer
4. Supplier

# Two argument Predefined Functional Interfaces

1. BiPredicate
2. BiFunction
3. BiConsumer

# Primitive Functional Interfaces

1. IntPredicate
2. IntFunction
3. IntConsumer

# Predicate

Predicate Interface is used for conditional checks.

interface **predicate**<T> {

public boolean test (T t);

}

public boolean **test** (Integer I) {

if(I%2==0) {

return true;

}

else {

return false;

}

}

I -> I%2==0;

# Predicate Scenarios

Examples,

import java.util.function.\*;

class **Test** {

public static void main (String[] args) {

**Predicate<Integer> P = I->I%2==0;**

SOP(P.test(10));

SOP(P.test(15));

}

}

import java.util.function.\*;

class **Test** {

public static void main (String[] args) {

**Predicate<Employee> P = e-> e.salary > 2000 && e.deptId = 20;**

SOP(P.test(e));

}

}

import java.util.function.\*;

class **Test** {

public static void main (String[] args) {

String[] s = {“John”, “Alex”, “Blake”, “Jenni”};

**Predicate<String> P = s1 -> s1.length() > 5;**

for (String s1: s) {

if(P.test(s1)) {

SOP (s1);

}

}

}

}

Example,

import java.util.function.\*;

import java.util.\*;

class **Employee** {

String ename;

double esalary;

Employee (String ename, double esalary) {

this.ename = ename;

this.esalary = esalary;

}

public String toString() {

return ename+”:”+esalary;

}

}

class **Test** {

public static void main (String[] args) {

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

al.add(new Employee(“Alex”, 2000);

al.add(new Employee(“Blake”, 4000);

al.add(new Employee(“Brown”, 6000);

SOP (al);

**Predicate <Employee> p = e->e.esalary > 3000;**

for (Employee e: al) {

if(p.test(e)) {

SOP(e.ename+”:”+e.esalary);

}

}

}

}

# Predicate Joining

import java.util.function.\*;

class **Test** {

public static void main (String[] args) {

int[] x = {5, 10, 15, 20, 25, 30};

**Predicate<Integer> p1 = I -> I%2==0;**

**Predicate<Integer> p2 = I -> I > 10;**

for (int x1 : x) {

//**if(p1.negate().test(x1)) { // Checks for number not even**

//**if(p1.or(p2).test(x1)) { // Checks for any one valid condition**

if(p1.and(p2).test(x1)) {

SOP (x1);

}

}

}

}

# Function

If we get one input, performing operations on the input and produce results. If the result is of any type and not only boolean, then in such case we should go for Function.

I -> I\*I;

S -> S.length()

S -> S+S;

**interface Function<T, R> {**

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

**}**

# Function Scenarios

Examples,

import java.util.function.\*;

class **Test** {

public static void main (String[] args) {

**Function<Integer, Integer> f = I –> I\*I;**

SOP (f.apply(4)); // **16**

}

}

import java.util.function.\*;

class **Test** {

public static void main (String[] args) {

**Function<String, Integer> f = S -> S.length();**

SOP (f.apply(“Alex”)); // **4**

}

}

import java.util.function.\*;

class **Student** {

String name;

int marks;

Student (String name, int marks) {

this.name = name;

this.marks = marks;

}

}

class **StudentGrade** {

public static void main (String[] args) {

**Function<Student, String> f -> {**

**String grade=””;**

**int marks = s.marks;**

**if(marks > 85) grade=”A”;**

**else if(marks > 75) grade=”B”;**

**else (marks > 65) grade=”C”;**

**return grade;**

**};**

**Predicate <Student> p = s-> s.marks > 80;**

Student[] s = { new Student(“Alex”, 90), new Student(“Blake”, 80), new Student(“John”, 70)};

for (Student s1: s) {

if(p.test(s1)) {

SOP (“Student Name : “+s1.name);

SOP (“Student Marks : “+s1.marks);

SOP (“Student Grade : “+f.apply(s1));

SOP ();

}

}

}

# Function Chaining

f1.andThen(f2).apply(i);

In the above case, the function f1 will be performed first and then followed by function f2.

f1.compose(f2).apply(i);

In the above case, the function f2 will be performed first and then followed by function f1.

Example,

import java.util.function.\*;

class **Test** {

public static void main (String[] args) {

Function<Integer, Integer> f1 = i-> 2\*I;

Function<Integer, Integer> f2 = i-> i\*i\*I;

SOP (f1.andThen(f2).apply(2));

SOP (f1.compose(f2).apply(2));

}

}

**Output**

64

16

# Consumer

Consumer is used to accept the value passed to it.

**interface Consumer <T> {**

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

**}**

# Consumer Scenario

Example,

import java.util.function.\*;

class **Student** {

String name;

int marks;

Student (String name, int marks) {

this.name = name;

this.marks = marks;

}

}

class **StudentGrade** {

public static void main (String[] args) {

**Function<Student, String> f -> {**

**String grade=””;**

**int marks = s.marks;**

**if(marks > 85) grade=”A”;**

**else if(marks > 75) grade=”B”;**

**else (marks > 65) grade=”C”;**

**return grade;**

**};**

**Predicate <Student> p = s-> s.marks > 80;**

Student[] s = { new Student(“Alex”, 90), new Student(“Blake”, 80), new Student(“John”, 70)};

**Consumer<Student> c = s2 -> {**

**SOP (“Student Name : “+s2.name);**

**SOP (“Student Marks : “+s2.marks);**

**SOP (“Student Grade : “+f.apply(s2));**

**SOP ();**

**}**

for (Student s1: s) {

if(p.test(s1)) {

c.accept(s1);

}

}

}

}

# Consumer Chaining

import java.util.function.\*;

class **Movie** {

String name;

Movie (String name) {

this.name = name;

}

}

class **Test** {

public static void main (String[] args) {

{

Consumer<Movie> c1 = m -> SOP (m.name+ “ released”);

Consumer<Movie> c2 = m -> SOP (m.name+ “ big hit”);

Consumer<Movie> c3 = m -> SOP (m.name+ “ saved to database”);

**Consumer<Movie> cr = c1.andThen(c2).andThen(c3);**

Movie m = new Movie (“Thunivu”);

cr.accept(m);

}

}

# Supplier

Just supply the required objects and it won’t take any input. Then we should go for Supplier.

**interface Supplier<R> {**

**public R get ();**

**}**

# Supplier Scenarios

Examples

import java.util.function.\*;

import java.util.Date;

class **Test** {

public static void main (String[] args) {

**Supplier<Date> s = ()-> new Date ();**

SOP (s.get());

}

}

import java.util.function.\*;

class **Test** {

public static void main (String[] args) {

Supplier <String> s = ()->{

String otp=””;

for(int i=0; i<6;i++){

otp = otp+(int)(Math.random()\*10);

}

return otp;

};

SOP (s.get());

}

}

# BiPredicate

BiPredicate interface is used for conditional check and it accept more than one input.

**interface BiPredicate <T1, T2> {**

**public boolean test (T1 t1, T2 t2);**

**}**

Example,

import java.util.function.\*;

class **Test** {

public static void main (String[] args) {

**BiPredicate<Integer, Integer> p = (a,b) -> (a+b)%2==0;**

SOP (p.test(10,20));

SOP (p.test(15,20));

}

}

Normal Predicate can take only one argument and perform some conditional check. Sometimes, our programming requirement is to have two arguments and perform conditional check. In such cases, we can go for **BiPredicate**.

# BiFunction

If we get two inputs, performing operations on the inputs and produce results. If the result is of any type and not only boolean, then in such case we should go for BiFunction.

**interface BiFunction <T, U, R> {**

**public R apply (T t, U u);**

**}**

Example,

import java.util.function.\*;

import java.util.\*;

class **Employee** {

int eno;

String ename;

Employee(int eno, String ename) {

this.eno = eno;

this.ename = ename;

}

}

class **Test** {

public static void main (String[] args) {

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

**BiFunction<Integer, String, Employee> f = (eno, ename) -> new Employee(eno, ename);**

al.add(f.apply(121,”Alex”));

al.add(f.apply(122,”Blake”));

for (Employee s : al) {

SOP (eno);

SOP (ename);

}

}

}

# BiConsumer

BiConsumer is used to accept the two arguments passed to it.

Example,

import java.util.function.\*;

import java.util.\*;

class **Employee** {

String ename;

double salary;

Employee(String ename, double salary) {

this.ename = ename;

this.salary = salary;

}

}

class **Test** {

public static void main (String[] args) {

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

populate (al);

Public static void populate (ArrayList <Employee> al) {

al.add(new Employee(“Alex”, 2500);

al.add(new Employee(“Blake”, 3500);

}

**Biconsumer<Employee, Double> c = (e, d) -> e.salary + d;**

for (Employee e : al) {

c.accept(e, 500.0);

}

for (Employee e : al) {

SOP (e.name);

SOP (e.salary);

SOP ();

}

}

}

# Primitive Predicate Types

Predicate Interface is used for conditional checks. But it has performance issue as it takes the type as Object type and when we need to use int value, it accepts the int value as argument in test () method and internally converts (autoboxing /autounboxing) it into Integer type, then perform the operations.

**Int -> Integer -> Int -> Integer -> Int**

To overcome the performance issue, we have below Primitive Predicates.

IntPredicate

DoublePredicate

LongPredicate

# Primitive Function Types

IntFunction

DoubleFunction

LongFunction

DoubleToIntFunction

**int applyAsInt (Double d);**

DoubleToLongFunction

**long applyAsLong (Double d);**

IntToDoubleFunction

IntToLongFunction

LongToIntFunction

LongToDoubleFunction

ToIntFunction

ToLongFunction

ToDoubleFunction

ToIntBiFunction

ToLongBiFunction

ToDoubleBiFunction

# Primitive Consumer Types

IntConsumer

LongConsumer

DoubleConsumer

ObjDoubleConsumer<T>

ObjIntConsumer<T>

ObjLongConsumer<T>

# Primitive Supplier Types

BooleanSupplier

IntSupplier

LongSupplier

DoubleSupplier

# Unary Operator

If the input and output is of same type, then we should go for Unary operator. It is a child of Function.

Example,

class **Test** {

public static void main (String[] args) {

UnaryOperator<Integer> u = i->i\*I;

SOP (u.apply(5));

}

}

# Primitive Unary Operator Types

IntUnaryOperator

**applyAsInt();**

LongUnaryOperator

**applyAsLong();**

DoubleUnaryOperator

**applyAsDouble();**

# Binary Operator

Binary operator takes 2 arguments. It is the child of BiFunction<T, T, T>.

interface BinaryOperator<T>

{

public T apply (T t);

}

Example,

class **Test** {

public static void main (String[] args) {

BinaryOperator<String> b = (s1, s2) -> s1+s2;

SOP (b.apply(“java”, “program”);

}

}

# Primitive Binary Operator Types

IntBinaryOperator

LongBinaryOperator

DoubleBinaryOperator

# Method & Constructor Reference

## Method Reference

For Lambda Expressions, we have alternate syntax such as Method and Constructor reference.

The advantage of using the Method reference is code reusability. The restriction wrt Method Reference is, the argument type must be matched with original method which is being bypassed. But, the return type can be different and need not be same.

**Syntax**

**ClassName::MethodName;**

**ObjectReference::MethodName;**

Examples,

class **Test** {

public static void m1() {

for (int i=0; i<10; i++) {

SOP (“Child Thread -1”);

}

public static void main (String[] args) {

Runnable r = Test::m1;

Thread t = new Thread(r);

t.start();

for (int i=0; i<10; i++) {

SOP (“Main Thread -1”);

}

}

}

interface **Interf** {

public void add (int a, int b);

}

class **Test** {

public static void sum(int x, int y) {

SOP (“The Sum : ”+(x+y));

}

public static void main (String[] args) {

Interf i = Test::sum;

i.add(10, 20);

}

}

## Constructor Reference

If the Functional Interface method returns an object, in that case we should go for Constructor Reference.

Examples,

class **Sample** {

Sample () {

SOP (“Sample Class Constructor”);

}

}

interface **Interf** {

public Sample get();

}

class **Test** {

public static void main (String[] args) {

Interf i = Sample::new;

Sample s = i.get();

}

}

class **Sample** {

Sample (String s) {

SOP (“Sample Class Constructor : ”+s);

}

}

interface **Interf** {

public Sample get(String s);

}

class **Test** {

public static void main (String[] args) {

Interf i = Sample::new;

Sample s = i.get(“Java”);

}

}

class **Student** {

String name;

int rollno;

int marks;

Student (String name, int rollno, int marks) {

this.name = name;

this.rollno = rollno;

this.marks = marks;

}

}

interface **Interf** {

public Student get(String name, int rollno, int marks);

}

class **Test** {

public static void main (String[] args) {

// Interf i = (name, rollno, marks) -> new Student (name, rollno, marks);

**Interf i = Student::new;**

Student s1 = i.get(“Alex”, 121, 90);

}

}

# When to go for Method Reference and Lambda Exp?

If the implementation of the method is already available, then we can happily go for Method Reference else we can go for Lambda Expression.

# Streams

The Streams is used to process the elements from the Collection. It is best helpful to do bulk operations on collection. The **stream ()** method present inside the Collection Interface.

**Stream s = c.stream();**

The stream interface is present inside the **java.util.stream** package.

From Collection, if we want to filter data based on boolean condition then we can go for **filter ()** method. If we want to do manipulation on elements presents in the Collection, then we can go for **map ()** method.

\* filter takes predicate as input and map takes function as input.

**The main advantage of Stream is concise coding.**

The important methods present in Stream are as below,

collect ()

count ()

sorted ()

sorted (Comparator c)

min (Comparator c)

max (Comparator c)

forEach ()

toArray ()

Stream.of() -> To apply stream for other entities (entities similar like collection or have group of values).

## filter ()

Example,

**ArrayList<Integer> l1 = new ArrayList<Integer>();**

l1.add(5);

l1.add(15);

l1.add(10);

l1.add(0);

l1.add(20);

SOP (l1);

**Without Streams**

**ArrayList<Integer> l2 = new ArrayList<Integer>();**

**for (Integer i : l1) {**

**if ( i%2==0) {**

**l2.add(i);**

**}**

**}**

SOP (l2);

**With Streams**

**ArrayList<Integer> l2 = l1.streams().filter(i->i%2==0).collect(Collectors.toList());**

SOP (l2);

## map ()

Example,

**Without Streams**

**ArrayList<Integer> l2 = new ArrayList<Integer>();**

**for (Integer i : l1) {**

**l2.add(i\*2);**

**}**

**}**

SOP (l2);

**With Streams**

**ArrayList<Integer> l2 = l1.streams().map(i-> i\*2).collect(Collectors.toList());**

SOP (l2);

## collect ()

**With Streams**

**ArrayList<Integer> l2 = l1.streams().map(i-> i\*2).collect(Collectors.toList());**

SOP (l2);

## count ()

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

l2.add(70);

l2.add(80);

l2.add(90);

l2.add(30);

l2.add(33);

**int numOfFailedStudents = marks.stream().filter(m -> m<35).count();**

SOP (numOfFailedStudents);

## sorted ()

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

l2.add(70);

l2.add(80);

l2.add(90);

**List<Integer> sortedMarks = marks.stream().sorted().collect(Collectors.toLsit());**

SOP (sortedMarks);

**List<Integer> descendingSortedMarks = marks.stream().sorted((i1, i2)->(i1<i2)?+1:(i1>i2)?-1:0).collect(Collectors.toLsit());**

SOP (descendingSortedMarks);

// Internally CompareTo() method used for descending sorted order

**List<Integer> sortedMarks = marks.stream().sorted((i1, i2)-> -i1.compareTo(i2)).collect(Collectors.toLsit());**

SOP (sortedMarks);

**List<Integer> descendingSortedMarks = marks.stream().sorted((i1, i2)-> i2.compareTo(i1)).collect(Collectors.toLsit());**

SOP (descendingSortedMarks);

Example,

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

al.add(“A”);

al.add(“CCC”);

al.add(BB”);

al.add(“DDDD”);

al.add(“EEE”);

SOP (al);

//Natural Sorting

**List<String> sortedNames = al.stream().sorted((i1, i2)-> i1.compareTo(i2)).collect(Collectors.toLsit());**

SOP (sortedNames);

//Reverse of Natural Sorting

**List<String> sortedNames = al.stream().sorted((i1, i2)-> i2.compareTo(i1)).collect(Collectors.toLsit());**

SOP (sortedNames);

Comparator<String> c = (s1, s2) ->{

int l1 = s1.length();

int l2 = s2.length();

if (l1 <l2) return -1;

elseif (l1>l2) return +1;

else return s1.compareTo(s2);

};

//Increase length order sorting

**List<String> sortedNames = al.stream().sorted(c).collect(Collectors.toLsit());**

SOP (sortedNames);

## min () and max ()

In sorting, the min and max elements is calculated based on the position of elements in the sorted collection.

Example,

[**3**, 4, 5, 10, **15**] -> In ascending order, 3 is considered as min and 15 is considered as max.

[**15**, 10, 5, 4, **3**] -> In descending order, 15 is considered as min and 3 is considered as max.

class **Test** {

public static void main (String[] args) {

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

al.add(15);

al.add(0);

al.add(5);

al.add(20);

al.add(10);

int min1 = al.stream().min((i1, i2)-> i1.compareTo(i2)).get();

int max1 = al.stream().max((i1, i2)-> i1.compareTo(i2)).get();

SOP (min1); //**0**

SOP (max1); //**20**

int min2 = al.stream().min((i1, i2)-> -i1.compareTo(i2)).get();

int max2 = al.stream().max((i1, i2)-> -i1.compareTo(i2)).get();

SOP (min2); //**20**

SOP (max2); //**0**

}

}

## forEach ()

**forEach (Function);**

class **Test** {

public static void main (String[] args) {

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

al.add(15);

al.add(0);

al.add(5);

al.add(20);

al.add(10);

SOP (al);

**al.stream().forEach(System.out::println);**

Consumer<Integer> c = i->{

SOP (Square of i is i\*i);

};

**al.stream().forEach(c);**

}

}

## toArray ()

It is used to convert the stream of objects into an array.

class **Test** {

public static void main (String[] args) {

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

al.add(15);

al.add(0);

al.add(5);

al.add(20);

al.add(10);

SOP (al);

**Integer[] i1 = al.stream().toArray(Integer[]::new);**

for(Integer a : i1) {

SOP (a);

}

}

}

## Stream.of ()

Examples,

class **Test** {

public static void main (String[] args) {

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

al.add(15);

al.add(0);

al.add(5);

al.add(20);

al.add(10);

SOP (al);

**Integer[] i1 = al.stream().toArray(Integer[]::new);**

**Stream.of(i1).forEach(System.out::println);**

}

}

class **Test** {

public static void main (String[] args) {

**Stream<Integer> s = Stream.of(9,99,999,9999,99999);**

**s.forEach(System.out::println);**

Integer[] i = {10, 20, 30, 40, 50};

**Stream.of(i).forEach(System.out::println);**

}

}

## flatMap ()

flatMap () is used to perform the transformation and flattering.

flatMap () takes Stream<Stream<T>> as input and return Stream<R>

It is mapper function produces multiple value for each input value. Hence, it is also called One-To-Many mapping.

**Example for Data transformation**

Stream.of(“a”, “b”, “c”, “d”); => [A, B, C, D] (Transform data from lowercase to uppercase)

**Example for Data flattering**

[[1,2], [3,4], [5,6], [7,8]] => [1, 2, 3, 4, 5, 6, 7, 8] (Convert stream of stream into single stream)

|  |  |
| --- | --- |
| **Map** | **FlatMap** |
| It processes stream of values | It processes stream of stream of values |
| It does only mapping | It does mapping as well as flattering |
| It produces single value for each input | It produces multiple values for each input |
| It is a one-to-one mapping | It is a one-to-many mapping |
| Data transformation: Stream to Stream | Data transformation: From <Stream<Stream>> to <Stream> |
| It is used when mapper function produces a single value for each input value | It is used when mapper function produces multiple values for each input values |

## skip () & limit ()

Skip operation is useful when we want to discard some number elements from actual collection.

Limit operation is useful when we want to limit the number of elements from actual collection.

class **Test** {

public static void main (String[] args) {

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

Numbers.stream().skip(2).limit(7).forEach(System.out::println);

}

}

# Optional

Optional class is introduced in Java 1.8v. It is used to check the null check for the objects.

empty(), of(), and ofNullable() methods used to create the Optional object.

Optional<Object> emptyOptional = Optional.empty();

SOP (emptyOptional); => Optional.Empty

Let’s assume we have email id of Customer class has value as null.

Optional<String> emailOptional = Optional.of(customer.getEmail());

SOP (emailOptional);

In this case, it will throw NullPointerException. The of() method is used when the object is non null.

Optional<String> emailOptional = Optional.ofNullable(customer.getEmail());

SOP (emailOptional); => Optional.Empty

Let’s assume if the email has value abc then the output of above statement is,

SOP (emailOptional); => Optional[abc]

Checking the get() method available in the Optional class

if (emailOptional.isPresent()) {

SOP (emailOptional.get());

}

To have the default value for email if its null,

SOP (emailOptional.orElse(“[default@email.com”)](mailto:default@email.com));

To throw exception incase if the email is null,

SOP (emailOptional.orElseThrow(() -> new IllegalArgumentException (“Email not present”)));

To have the default value for email if its null,

SOP (emailOptional.map(String::toUpperCase).orElseGet(()->”[default@email.com”)](mailto:default@email.com));

# Map-Reduce

Map-Reduce is a functional programming model and it serves 2 purposes as below,

Map -> Transforming data

Reduce -> Aggregating data

(Combine elements of a stream and produces a single value)

Example,

Stream: [2, 4, 6, 7, 8, 9] sum of numbers presents in Stream?

map () - Transform Stream <Object> to Stream of int

reduce () – Combine stream of int and produce the sum result

**T reduce (identity, BinaryOperator<T> accumulator);**

Identity is the initial value of type T.

Accumulator is a function for combining two values.

Integer sumResult = Stream.of(2, 4, 6, 7, 9). reduce (0, (a,b) -> a+b);

0 -> identity initial value

(a,b)->a+b -> accumulator

Example,

class **Test** {

public static void main (String[] args) {

List<Integer> numbers = Arrays.asList(3, 7, 2, 5, 8);

int sum1 = numbers.stream().mapToInt(i->i).sum();

SOP (sum1);

Integer sum1 = numbers.stream().reduce(0, (a,b)->a+b);

SOP (sum1);

Optional<Integer> sum1 = numbers.stream().reduce(Integer::sum);

SOP (sum1);

Integer max1 = numbers.stream().reduce(0, (a,b) -> a > b?a:b);

SOP (max1);

Integer max1 = numbers.stream().reduce(Integer::max).get();

SOP (max1);

List<String> words = Arrays.asList(“corejava”, “spring”, “hibernate”);

String longestWord = words.stream().reduce((word1, word2) -> word1.length()>word2.length()? word1:

Word2).get();

SOP (longestWord);

}

}

# Parallel Stream

Parallel stream is a java8 feature. It meant for utilizing multiple cores of the processor.

Normally any java code has one stream of processing, where it is executed sequentially. Whereas by using parallel streams, we can divide the code into multiple streams that are executed in parallel on separate cores and the final result is the combination of the individual outcomes.

The order of execution is not under the control.

The Parallel Stream can be created using the parallel () and parallelStream () methods.

Example,

class **Test** {

public static void main (String[] args) {

long start=0;

long end=0;

start = System.currentTimeInMillis();

IntStream.range(1, 100).forEach(System.out::println);

end = System.currentTimeInMillis();

SOP (“Time taken by normal stream : “+(end-start));

start = System.currentTimeInMillis();

IntStream.range(1, 100).parallel().forEach(System.out::println);

end = System.currentTimeInMillis();

SOP (“Time taken by parallel stream : “+(end-start));

IntStream.range(1, 10).forEach( x -> {

SOP (“Thread :”+Thread.currentThread().getName()+”:”+x);

});

IntStream.range(1, 10)parallel().forEach( x -> {

SOP (“Thread :”+Thread.currentThread().getName()+”:”+x);

});

}

}

# Completable Feature

A new era of asynchronous programming. Using Async programming you can write the non-blocking code where concurrently you can run N no of tasks in separate thread without blocking the main thread.

When the task is complete it notifies to the main thread (whether the task was completed or failed).

# Why Completable Feature

There are different ways to implement the async programming in Java using any of the below mechanisms for example you can use Features, ExecutorService, Callback Interfaces, Thread Pools, etc. But these are having below disadvantages.

1. It can’t be manually completed.
2. Multiple features can’t be chained together
3. We can’t combine multiple features together
4. No proper exception handling mechanism

# CompletableFuture Creation

CompletableFuture<String> completableFuture = new CompletableFuture<>();

completableFuture.get();

completableFuture.complete(“task is taking long time, lets have the dummy data”);

# runAsync () vs supplyAsync ()

If we want to run some background tasks asynchronously and do not want to return anything from that task, then use CompletableFuture.runAsync() method. It takes a Runnable object and returns CompletableFuture<Void>.

1. CompletableFuture.runAsync(Runnble);
2. CompletableFuture.runAsync(Runnble, Executor);

If we want to run some background tasks asynchronously and want to return anything from that task, then use CompletableFuture.supplyAsync() method. It takes a Supplier<T> and returns CompletableFuture<T> where T is the type of the value obtained by calling the given supplier.

1. CompletableFuture.supplyAsync (Supplier<T>);
2. CompletableFuture.supplyAsync (Supplier<T>, Executor);

# thenApply(), thenAccept() & thenRun()

CompletableFeature have the methods like thenApply(Function), thenAccept(Consumer), and thenRun(Runnable). These methods are used to chain the CompletableFeature actions.

**Use Case: Employee Training Remainder**

1. Get all employees from DB
2. Filter out new joined employees
3. Check if training activity is pending for employee
4. Get employees email id
5. Send remainder notifications to employees