Java 8 :-

@FunctionalInterface :

A FunctionalInterface having only one abstract method, if @FunctionalInterface is must satisfied condition then we will apply Lamda expression(where we can use Lamda) for that interface otherwise Lamda Expressions not support.

Instead of creating object we can directly use lamda expressions. Lamda expressions works with only Functional interface

Jdk 1.7 :

Step :1

package com.app.inte;

@FunctionalInterface

public interface FunctionalInterfaceExample {

void applyLamdaExp();

}

Step :2

package com.app.impl;

import com.app.inte.FunctionalInterfaceExample;

public class FunctionalInterfaceExampleImpl {

public static void main(String[] args) {

FunctionalInterfaceExample e = new FunctionalInterfaceExample() {

@Override

public void applyLamdaExp() {

System.out.println("--Call method--");

}

};

e.applyLamdaExp();

}

}

Jdk 1.8 :

Step :1 – above same

Step :2

**package** com.app.impl;

**import** com.app.inte.FunctionalInterfaceExample;

**public** **class** FunctionalInterfaceExampleImpl {

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

FunctionalInterfaceExample e =()->System.***out***.println("--call method---");

e.applyLamdaExp();

}}

Example 2

Step :1

**package** com.app.impl;

**import** com.app.inte.FunctionalInterfaceExample;

**public** **class** FunctionalInterfaceExampleImpl **implements** FunctionalInterfaceExample {

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

FunctionalInterfaceExampleImpl i = **new** FunctionalInterfaceExampleImpl();

i.applyLamdaExp();

}

@Override

**public** **void** applyLamdaExp() {

System.***out***.println("--Call Method---");

}

}

Step :2

**package** com.app.impl;

**import** com.app.inte.FunctionalInterfaceExample;

**public** **class** FunctionalInterfaceExampleImpl {

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

FunctionalInterfaceExample i = ()->System.***out***.println("--call method--");

i.applyLamdaExp();

}

}

Streams : -

A stream is used to process collection of objects.

Examples :

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

data.stream().forEach(i->System.***out***.print(i));

(Or)

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

Streams Types :

1.sequential stream :-

execution of sequential streams works just like for-loop using a single core.

2. parallel Stream :-

execution of Parallel streams run multiple iterations simultaneously in different available cores.

In parallel stream, Fork and Join framework is used in the background to create multiple threads.

Parallel streams create ForkJoinPool instance via static ForkJoinPool.commonPool() method.

Fork->A Task splits subtask concurrently (asynchronously)

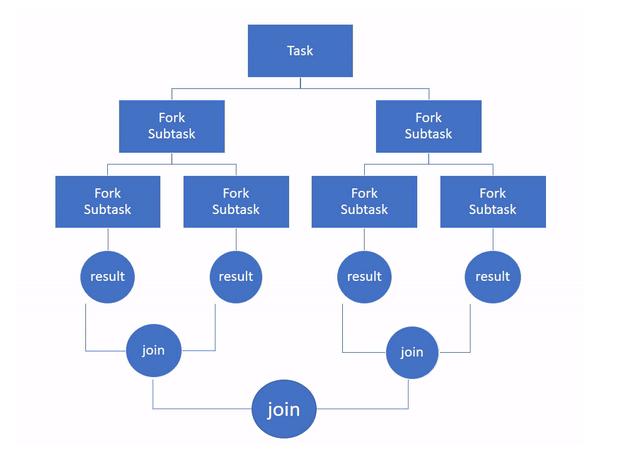
1. fork() method
2. compute() method

Join->A Task Join subtask

1. join() method

ForkJoinPool(JDK 1.7) ☹Class

Which is used to work perform for above operations



Work Stealing :-

aim of reducing contention(  
వివాదాస్పద) in multi-threaded applications. see below

### Divide and Conquer Approach

if problem is small, directly solve problem or split problem into independent parts. fork new subtasks to solve each part join all subtasks compose result from subresults .

Different Operations On Streams :-

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

**Intermediate Operations:**

1. **map :** Stream<R> map(Function<? **super** T, ? **extends** R> mapper);

**(or)** Stream<R> map(Function<T, R>);

**Creating interface object go for implementation class to create object**

**//Anonymous inner class code for below interface----imperative style**

Function<Integer,Integer>fun=**new** Function<Integer,Integer>() {

@Override

**public** Integer apply(Integer t) {

**return** t\*2;

}

};

Stream mapobj= colref.stream().map(fun);

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

one line :output

colref.stream().map((t)->t\*2).forEach(System.***out***::print);

**op:** **24681012**

1. **filter :-** Stream<T> filter(Predicate<? **super** T> predicate);

**requirement :- I want to add given numbers are divisible by 5**

**Jdk 1.7 :**

List<Integer> values = Arrays.*asList*(12, 20, 35, 46, 55, 75);

**int** result = 0;

**for** (**int** i : values) {

**if** (i % 5 == 0) {

result += i;

}

}

System.***out***.println(result);

**Jdk 1.8 :**

**Creating interface object go for implementation class to create object**

**//Anonymous inner class code for below interface----imperative style**

Predicate<Integer> p=**new** Predicate<Integer>() {

@Override

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

// **TODO** Auto-generated method stub

**return** t%5==0;

}

};

System.***out***.println(values.stream().filter(p).reduce(0, Integer::*sum*));

one line :output

System.***out***.println(values.stream().filter(i->i%5==0).reduce(0, Integer::*sum*))

**op:** **185**

1. **sorted :**

**Terminal Operations:**

1. **collect :**
2. **forEach : void** forEach(Consumer<? **super** T> action);

Consumer<String>con=**new** Consumer<String>() {

@Override

**public** **void** accept(String name) {

System.***out***.println(name);

}

};

names.stream().forEach(name->System.***out***.println(name));

one line :output

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

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

T reduce(T, BinaryOperator<T>);

BinaryOperator<T> **extends** BiFunction<T,T,T> :------

If you see for Functional interface method extends class (BiFunction<T,T,T> ):-

Method : R apply(T t, U u);

**Creating interface object go for implementation class to create object**

**//Anonymous code for below interface----imperative style**

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

BinaryOperator<Integer>bina=**new** BinaryOperator<Integer>() {

@Override

**public** Integer apply(Integer t,Integer u) {

**return** t+u;

}

};

Stream mapres=colref.stream().map(t->t\*2);

Integer redres=(Integer) mapres.reduce(0,bina);//bina->apply lamd

System.***out***.println(redres);

one line :output

System.***out***.println(colref.stream().map(t->t\*2).reduce(0,(t,u)->t+u));

(or)

System.***out***.println(colref.stream().map(t->t\*2).reduce(0,(t,u)->Integer.*sum*(t,u)));

(or)

System.***out***.println(colref.stream().map(t->t\*2).reduce(0,Integer::*sum*));

**op:** **42**

Arrays.stream() method : - It works only primitive types

**char** arr[] = { '1', '2', '3', '4', '5' }; //char

// to convert char array into Stream

Arrays.stream(arr); // compile time error

Stream.of() method :- it works any generic types

**char** arr[] = { '1', '2', '3', '4', '5' };

Stream<**char**[]> st = Stream.*of*(arr);

st.forEach(s->System.***out***.println(s)); o/p:success

**Examples :**

**int**[]arr= {1,2,3,4,5,6};

// To convert int array to Streams

IntStream sol=Arrays.*stream*(arr);

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

Example :

List<Integer>one=Arrays.*asList*(10,11,12,13);

Consumer<Integer>obj=**new** Consumer<Integer>() {

@Override

**public** **void** accept(Integer t) {

System.***out***.println(t);

}

};

one.forEach(obj);

(Or)

List<Integer>one=Arrays.*asList*(10,11,12,13);

Consumer<Integer>obj=(i)->System.***out***.println(i);

one.forEach(obj);

(Or)

List<Integer>one=Arrays.*asList*(10,11,12,13);

one.forEach((i)->System.***out***.println(i));

Examples :

Filter :-

Ex: List<String> names = Arrays.*asList*("dhana", "jaya", "kakarla");

Jdk 1.7: imperative style

**for** (String name : names) {

**if** (!name.equals("dhana")) {

System.***out***.println(name);

}

}

Jdk 1.8:

Predicate<String>pre=**new** Predicate<String>()

{

@Override

**public** **boolean** test(String name) {

**return** !name.equals("dhana");

}

};

names.stream().filter(pre).forEach(System.***out***::println);

one line op : names.stream().filter(name -> !name.equals("dhana")).forEach(System.***out***::println);

output :

jaya

kakarla

Example :

Step 1: Model class:

**package** com.app.model;

**public** **class** User {

**private** String name;

**private** Integer age=30;

**//def con**

**//one arg con**

**//param con**

**//set/get**

**// tostring**

**}**

**Step 2:**

**public** **class** FunctionalInterfaceExampleImpl {

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

List<String> names = Arrays.*asList*("dhana", "jaya", "kakarla");

Names

.stream()

.filter(name -> !name.equals("dhana"))

.map(name->**new** User(name))

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

//names.stream().filter(name -> !name.equals("dhana")).map(User::new).forEach(System.out::println);

}

}

Example :collect

List<String> names = Arrays.*asList*("dhana", "jaya", "kakarla");

List<User>listuser= names

.stream()

.filter(name -> !name.equals("dhana"))

.map(User::**new**)

.collect(Collectors.*toList*());

System.***out***.println(listuser);

**int** su=listuser

.stream()

.mapToInt(user->user.getAge())

.sum();

System.***out***.println(su);

Example:

List<User> names = Arrays.*asList*(

**new** User("Dhanamjaya",26,Arrays.*asList*(1,2,3)),

**new** User("Kakarla",28,Arrays.*asList*(5,6,7)),

**new** User("Jaya",30,Arrays.*asList*(8,9,10)));

// requirement : we want matching for who's phone number is 5

Optional<Integer>o=names

.stream()

.map(user->user.getPhoneNumbers().stream())

.flatMap(phoneNo->phoneNo.filter(no->no.equals(12)))

.findAny();

System.***out***.println(o);