**Index**

1. Program to print all number using enhanced for loop
2. Program to print all number using stream but with extra method
3. Program to print all number using **stream** and **method reference**
4. Program to print all even number using **stream** and **extra method**
5. Program to print all even number using **stream** and **lambda expression**
6. Some exercises

**Reduce function**

1. Program to add all numbers using for loop
2. Program to add all numbers using stream and extra method
3. Program to add all numbers using stream and lambda expression
4. Program to add all numbers using stream and method reference (predefined)
5. Some exercises

**Distinct and sorted**

1. Print distinct number, sortednumbers, distinct sorted numbers from stream
2. Sort the string from stream
3. Sort string in natural order,***customized*** order, ***reversed*** order

**Collect operation**

1. Collect list by squaring each number.
2. Some exercises
   1. Create a list with even number filtered from number list.(Ex\_09)
   2. Create a list with length of all course titles. (Ex\_10)
3. **Intermediate and terminal operations (theory)**
4. Functional Interfaces
5. Exercise of reduce functionimplementation (Ex11)
6. Behaviour parameterization
7. AlllMatch, noneMatch, anyMatch
8. Sorting operation

## Functional Interfaces

The interface which has only one abstract method is called as functional interface. It may have multiple static or default methods.

How can we implement the abstract method?

1. By taking a class and implement it.
2. By anonymous inner class
3. By lambda expression (advance to 2nd option)

Lets take an interface and will see the different way to implement it.

**@FunctionalInterface**

**publicinterface DisplayName {**

**public String displayNameInUpperCase(String name);**

**}**

1. **By taking a class and implement it.**

**package** com.in28Minutes.functinalInterface;

**publicclass** ImplementedClass **implements** DisplayName {

@Override

**public** String displayNameInUpperCase(String name) {

**return**name.toUpperCase();

}

**publicstaticvoid** main(String[] args) {

ImplementedClass obj = **new**ImplementedClass();

System.***out***.println(obj.displayNameInUpperCase("ankur"));

}

}

1. **By anonymous inner class**

**package** com.in28Minutes.functinalInterface;

**publicclass** AnonymousClassExample {

**publicstaticvoid** main(String[] args) {

DisplayName displayName=**new**DisplayName() {

@Override

**public** String displayNameInUpperCase(String name) {

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

**return**name.toUpperCase();

}

};

System.***out***.println(displayName.displayNameInUpperCase("nikhil"));

}

}

1. **By lambda expression (advance to 2nd option)**

**package** com.in28Minutes.functinalInterface;

**publicclass** AnonymousClassExample {

**publicstaticvoid** main(String[] args) {

DisplayName lambda=str->str.toUpperCase();

System.***out***.println(lambda.displayNameInUpperCase("dhanu"));

}

}

So now let move towards predefined functional interface and how they are implemented with step by step. Consider a following example.

**package** com.in28Minutes.functinalInterface;

**import** java.util.Arrays;

**import** java.util.List;

**import** java.util.function.Consumer;

**import** java.util.function.Function;

**import** java.util.function.Predicate;

**publicclass** FunctionalInterfaceDemo {

**publicstaticvoid** main(String[] args) {

List<Integer>numbers = Arrays.*asList*(1, 23, 56, 21, 25, 46, 89, 100);

**/////////////////////////////Predicate implematation**

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

@Override

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

**return** t % 2 == 0;

}

};

Predicate<Integer>newPredicate= num ->num % 2 == 0;

///////////////////////////Function

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

@Override

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

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

**Return** t\*t;

}

};

Function<Integer, Integer>newSquareFunction= t->t\*t;

/////////////////////////////Consumer

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

@Override

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

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

}

};

Consumer<Integer>newSysoutConsumer=t->System.***out***.println(t); // System.out::println

//-----------------------------

**//version without lambda** numbers.stream().filter(newPredicate).map(newSquareFunction).forEach(newSysoutConsumer);

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

**// final version with lambda expression**

numbers.stream().filter(num ->num % 2 == 0).map(num->num\*num).forEach(System.***out***::println);

}

}

## Behaviour parameterization

Passing logic of method as an argument while calling, called as behaviour parameterization.

Ex: Write a program to do following operations on it and return new list.

1. Square each number
2. Make cube of each number

**Normal way**

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

//normal way

List<Integer>squaredNumberNormalWay=

numbers.stream().***map(num->num\*num).***collect(Collectors.*toList*());

List<Integer>cubeNumberNormalWay=

numbers.stream().***map(num->num\*num\*num)***.collect(Collectors.*toList*());

}

Here, only the highlighted code is repeating again and again. So its better to send that part(logic) to a common method. So we will be sending a logic to common method as per requirement. That mean we are sending a logic or behaviour of a method as an argument.

See below example which will show normal way and behaviourParameterization way

package com.in28Minutes.examples;

import java.util.Arrays;

importjava.util.Comparator;

import java.util.List;

import java.util.function.Function;

importjava.util.stream.Collectors;

public class Ex15\_BehaviourParameterization {

public static void main(String[] args) {

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

**// normal way**

List<Integer>squaredNumberNormalWay = numbers.stream().map(num ->num \* num).collect(Collectors.toList());

List<Integer>cubeNumberNormalWay = numbers.stream().map(num ->num \* num \* num).collect(Collectors.toList());

**// behaviouralparameterization way**

Function<Integer, Integer>addNumber = num -> num + num;

List<Integer>addedNumber = mapAndFilterList(numbers, **addNumber**);

System.out.println(addedNumber);

List<Integer>squaredNumber = mapAndFilterList(numbers, **num -> num \* num**);

System.out.println(squaredNumber);

List<Integer>cubedNumber = mapAndFilterList(numbers, num ->**num \* num \* num**);

System.out.println(cubedNumber);

}

private static List<Integer>***mapAndFilterList***(List<Integer> numbers, Function<Integer, Integer> mapper) {

returnnumbers.stream().map(mapper).collect(Collectors.toList());

}

}

**Functional Interfaces**

**Predicate** used to check the boolean condition.

**Function** is used to do some logic and return any specific value.

**Consumer** takes an input and process It but it won’t return back.

## Supplier and Unary Operator

If you don’t want to give any input but you have to return something back that time we use supplier.

**Supplier** is *functional interface*which will not take anything but return something back.

Ex:

Supplier<Integer>supplier=()->2;

System.***out***.println(supplier.get());

Supplier<Integer>supplier=()->{

Random random=**new**Random();

**return**random.nextInt(500);

};

System.***out***.println(supplier.get());

## Unary Operator

It’s a functional interface which accept a values and return back the value.

UnaryOperator<Integer>operator=x->x\*3;

System.***out***.println(operator.apply(25));

## BiPredicate

We can combine multiple predicated if we have to check multiple conditions.

BiPredicate have following methods

1. Test( for testing any specific condition and returns boolean)
2. And (for combining two predicated)
3. Or(for doing or operation on multiple predicates)
4. Negate

**publicclass** Ex17\_BiPredicateDemo {

**publicstaticvoid** main(String[] args) {

// Simple predicate for checking equality

BiPredicate<Integer, String>biPredicate = (n, s) -> {

**if** (n == Integer.*parseInt*(s))

**returntrue**;

**returnfalse**;

};

System.***out***.println(biPredicate.test(2, "2"));

// Predicate for checking greater than

BiPredicate<Integer, String>biPredicate1 = (n, s) -> {

**if** (n>Integer.*parseInt*(s))

**returntrue**;

**returnfalse**;

};

// ANDing the two predicates

BiPredicate<Integer, String>biPredicate2 = biPredicate.and(biPredicate1);

System.***out***.println(biPredicate2.test(2, "3"));

// ORing the two predicates

biPredicate2 = biPredicate.or(biPredicate1);

System.***out***.println(biPredicate2.test(3, "2"));

// Negating the predicate

biPredicate2 = biPredicate.negate();

System.***out***.println(biPredicate2.test(3, "2"));

}

}

# AlllMatch, noneMatch, anyMatch

**AlllMatch**: used to check if all elements of stream satisfies with given predicate or not.

**noneMatch** : used to check if no elements of stream satisfy given predicate. If all elements are not satisfying that condition it will return true otherwise false.

**anyMatch:** it is used to check if any one of element of stream satisfies the predicate.

package com.in28Minutes.examples;

importjava.util.Arrays;

importjava.util.List;

importjava.util.function.BiPredicate;

importjava.util.function.Predicate;

public class Ex18\_AllMatchNoneMatchAnyMatch {

public static void main(String[] args) {

List<Course> courses = Arrays.asList(new Course("Spring", "Framework", 98, 20000),

new Course("Spring Boot", "Framework", 95, 18000),

new Course("API", "Microservices", 97, 22000),

new Course("Microservices", "Microservices", 96, 25000),

new Course("FullStack", "FullStack", 91, 14000), new Course("AWS", "Cloud", 92, 21000),

new Course("Azure", "Cloud", 99, 21000), new Course("Docker", "Cloud", 98, 20000),

new Course("Kubernetes", "Cloud", 91, 20000));

Predicate<?super Course> reviewScoreGreaterThan95Predicate = course ->course.getReviewScore() > 95;

Predicate<?super Course> reviewScoreGreaterThan90Predicate = course ->course.getReviewScore() > 90;

Predicate<?super Course> reviewScoreLessThan80Predicate = course ->course.getReviewScore() < 80;

// check if all are having > 90 review score

System.out.println(courses.stream().allMatch(reviewScoreGreaterThan90Predicate)); // true

// check if all are not having < 90 review score

System.out.println(courses.stream().noneMatch(course ->course.getReviewScore() < 90)); // true

// check if all are not having < 100 review score

System.out.println(courses.stream().noneMatch(course ->course.getReviewScore() < 100)); // false

// check if any one is having 99 review score

System.out.println(courses.stream().anyMatch(course ->course.getReviewScore() == 99));// true

}

}

## Sorting operation

1. Natural sort based on specific class property (ascending order)
2. Customized sort using comparator
3. Sorting based on mix criteria

For sorting stream element, those should be of type *comparable.*

**Sorted(comparator)** is used for this.

*How to write comparator?*

Comparator interface contains static method ***comparing*** which accepts Function.

Syntax: sorted(Comparator.compairing(***method reference***))

Above syntact sorts ascendingly. For reverse order , we just need to use reversed() method like below

sorted(Comparator.compairing(***method reference***).***reversed()***)

Ex: Ex19\_SortingOperation

Task : sort the list of objects based on property length.

//sort based on length of string -remained task

List<String>names = courses.stream().map(course->course.getName()).collect(Collectors.*toList*());

names.stream().sorted(Comparator.*comparing*(str->str.length())).forEach(System.***out***::println);

## Skip, limit, takeWhile, dropWhile

Skip: if we have to skip first specific count

Limit:if we want any limited elements

skipWhile:

dropWhile:

Ex20\_SkipLimitTakeWhileDropWhile

## Min, max, findFirst, findLast

Find an min, max, first or last element while comparing stream elements.