**JAVA**

**Git Repo :** **https://github.com/in28minutes/functional-programming-with-java**

Streams:

List<Integer> numbers = new ArrayList<>(Arrays.asList(12,9,6,2,122, 15));

List<String> courses = new ArrayList<>(Arrays.asList("Spring", "Spring Boot", "API" , "Microservices","AWS", "PCF","Azure", "Docker", "Kubernetes"));

* For Each
* Filter
* Map
* Collectors
* Reduce

# **ForEach:**

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

# **Filter:**

courses.stream().**filter(course -> course.contains("Spring"))**.forEach(System.out::println);

courses.stream().**filter(course -> course.length() >= 4).**forEach(System.out::println);

numbers.stream().**filter(number -> number % 2 != 0).**forEach(System.out::println);

# **Map**

courses.stream().**map(course -> course + " " + course.length()).**forEach(System.out::println);

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

# **Collectors**

List<Integer> evenNumbersOnly = numbers.stream().filter(x -> x%2==0).**collect(Collectors.toList());**

# **Reduce**

numbers.stream().parallel**().reduce(0, (x,y) -> x + y); (sum of numbers ,x is aggregator, y -numbers)**

numbers.stream().parallel**().reduce(0, Integer::sum);**

numbers.stream().parallel**().reduce(0, (x,y) -> x ); - return 0**

numbers.stream().parallel**().reduce(0, (x,y) -> y); - return 15 (last element)**

numbers.stream().parallel**().reduce(0, (x,y) -> x>y? x:y ); - return minvalue**

numbers.stream().parallel**().reduce(0, (x,y) -> x>y? y:x ); - return maxvalue**

**12+22+32+…….**

numbers.stream().**map(x-> x\*x ).reduce(0, Integer::sum );**

# **Distinct**

numbers.stream().distinct().forEach(System.out::println)

# **Sorted**

numbers.stream().sorted().forEach(System.out::println)

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

# **Functional Interfaces**

* Predicate
* Function
* Consumer

**Predicate:**

numbers.stream().filter(x-> x%2 == 0).forEach(System.out::println);

numbers.stream().filter(predicate).forEach(System.out::println);

**Function:**

Function<Integer, Integer> **mappingFunction** = x -> x\*x; (paramerter,Output )

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

numbers.stream().map(**mappingFunction**).collect(Collectors.toList())

**Consumer:**

Consumer<Integer> sysoutconsumer=System.out::println;

numbers.stream().map(**x -> x\*x**).forEach(System.out::println);

numbers.stream().map(**x -> x\*x**).forEach(sysoutconsumer);

**Predicate :**

Predicate<Integer> isEvenPredicate = x -> x%2==0;

Predicate<Integer> isEvenPredicate2 = new Predicate<Integer>() {

@Override

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

return x%2==0;

}

};

Behavioral Parameterization

private static void filterAndPrint(List<Integer> numbers, **Predicate<? super Integer> predicate**) {

numbers.stream()

.filter(**predicate**)

.forEach(System.out::println);

}

**Function:**

Function<Integer, Integer> squareFunction = x -> x \* x;

Function<Integer, Integer> squareFunction2 = new Function<Integer, Integer>() {

@Override

public Integer **apply**(Integer x) {

return x\*x;

}

};

**Consumer:**

Consumer<Integer> sysoutConsumer = System.out::println;

Consumer<Integer> sysoutConsumer2 = new Consumer<Integer>() {

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

System.out.println(x);

}

};

**Supplier:**

No Input,return Values

Supplier<Integer> randomIntegerSupplier = () -> {

Random random = new Random();

return random.nextInt(1000);

};

**BinaryOperator:**

Operator means Same type input and out put .

BinaryOperator<Integer> sumBinaryOperator = Integer::sum;

**UnaryOperator:**

UnaryOperator<Integer> unaryOperator = x-> x+3;

**BiPredicate :**

BiPredicate<Integer, String> biPredicate = (number,str) -> {

return number<10 && str.length()>5;

};

**BiFuncation:**

BiFunction<Integer, String, String> biFunction = (number,str) -> {

return number + " " + str;

};

**BiConsumer:**

BiConsumer<Integer, String> biConsumer = (s1,s2) -> {

System.out.println(s1);

System.out.println(s2);

};

# **Custom Class :**

class Course {

private String name;

private String category;

private int reviewScore;

private int noOfStudents;

**AllMatches (true/false)**

Predicate<Course> reviewScoreGThan95Predicate = course -> course.getReviewScore() > 95;

Syystem.out.println(courses.stream().allMatch(reviewScoreGThan95Predicate));

**NoneMatches(true/false)**

Predicate<Course> reviewScoreLThan90Predicate=course ->course.getReviewScore()< 90;

System.out.println(courses.stream().noneMatch(reviewScoreLThan90Predicate));

AnyMatches **(true/false)**

Predicate<Course> reviewScoreGThan95Predicate = course -> course.getReviewScore() > 95;

Syystem.out.println(courses.stream().anyMatch(reviewScoreGThan95Predicate));

**Sorting**

Comparator<Course> comparingByNoOfStud= Comparator.comparingInt(Course::getNoOfStudents);

Comparator<Course> comparingBydecrStud= Comparator.comparingInt(Course::getNoOfStudents),reversed();

Comparator<Course> comparingByNoOfStudentsAndNoOfReviews = Comparator.**comparingInt(Course::getNoOfStudents).thenComparingInt**(Course::getReviewScore).reversed();

Sys.out.println(courses.stream().sorted(comparingByNoOfStud).collect(Collectors.toList()));

**Limit**

courses.stream().sorted(comparingByNoOfStudentsAndNoOfReviews).**limit(5)**.collect(Collectors.toList()));

**Skip**

courses.stream().sorted(comparingByNoOfStudentsAndNoOfReviews).**skip(5)**.collect(Collectors.toList()));

**takeWhile**

courses.stream().takeWhile(course -> course.getReviewScore()>=95).collect(Collectors.toList()));

**dropWile**

courses.stream().dropWhile(course -> course.getReviewScore()>=95).collect(Collectors.toList()));

T