**Return to my GitHub if want to see code examples**

**Shortcut for Intelliji:**

**Sout > to println**

**Ctl+b > to navigate and know more about function, method..**

**Psvm > main method**

**General: window + shift + s > snipping tool**

**Highlait text (whole world): ctl+shift+ arrow**

**Highlait text (letter by letter): shift + arrow**

Graphical user interface, text, application

Description automatically generated with medium confidence

Graphical user interface, text, application, email

Description automatically generated

#Case we want to sum nums from 0-100

Imparative: will define mutable variable sum =0; then for loop then increment sum;

In declarative approach, we use functional programming: - more readable, no need to worry how to perform summation behind the scene.

**int** sum = IntStream.*rangeClosed*(0, 100)

.parallel() // to be multi-thread safe?

.sum();

# Example 2

|  |
| --- |
| List<Integer> integerList =Arrays.asList(1,2,3,4,4,5,5,6,7,7,8,9,9); |
|  |  | |
| //Remove the duplicates from the list. | |
|  | |
| /\*\* | |
| \* Imperative Style | |
| \*/ | |
| List<Integer> uniqueList = new ArrayList<>(); | |
| for(Integer i :integerList) | |
| if(!uniqueList.contains(i)){ | |
| uniqueList.add(i); | |
| } | |
| System.out.println("unique List : " + uniqueList); | |
|  | |
| /\*\* | |
| \* Declarative Syle | |
| \*/ | |
|  | |
| List<Integer> uniqueList1 = integerList.stream() | |
| .distinct() | |
| .collect(toList()); | |
| System.out.println("uniqueList1 : " + uniqueList1); | |





// forEach take Consumer or BiConsumer implementation

//Consumer functional interface has

void accept(T t);

* One parameter and also has andThen default method

// BiConsumer have the same methods but accept method has two pararmeters

public class BiConsumerExample {  
 public static void printNamesAndActivities(){  
 BiConsumer<String, List<String>> biConsumer = (name, activities) -> System.*out*.println(name +  
 " : " + activities);  
 List<Student> studentList = StudentDataBase.*getAllStudents*();  
 studentList.forEach((student -> biConsumer.accept(student.getName(), student.getActivities())));  
 }  
  
 public static void main(String[] args) {  
 *printNamesAndActivities*();  
 }  
}

//Predicate functional interface has test abstract method return boolean alue and pass aparameter

// Also has other default methods: and , or, negate

// we can use predicate and biconsumer inside consumer – predicate to test, biconsumer to print. Consumer will take Student object to use the student input and call the methods of getNames and getActvites as paramateres of accept inside bicosumer

Predicate<Student> p1 = (s) -> s.getGradeLevel()>=3;  
Predicate<Student> p2 = (s) -> s.getGpa()>=3.9;  
  
BiConsumer<String, List<String>> biConsumer = (name, activities) ->  
 System.*out*.println(name + " : " + activities);  
  
Consumer<Student> consumer = (student ->  
{  
 if(p1.and(p2).test(student)){  
 biConsumer.accept(student.getName(), student.getActivities());  
 }  
});  
  
public void namesAndActivitiesFilter(List<Student> students){  
 students.forEach(consumer);  
}  
public static void main(String[] args) {  
 List<Student> students = StudentDataBase.*getAllStudents*();  
 new PredicateAndConsumerExample().namesAndActivitiesFilter(students);  
  
  
  
}

BiPredicate same as Predicate but test accept two inputs.

Function interfacehas apply abstract method

 R apply(T t, U u);

And dafualt methods are: andThen, Compose

// first parameter is inpt, sec is outut  
static Function<String, String> *function* = (name) -> name.toUpperCase();  
static Function <String,String> *function1* = (name) -> name.toUpperCase().concat(" default");  
public static void main(String[] args) {  
 System.*out*.println("apply result is " + *function*.apply("Fatimah") );  
 // FATIMAH default  
 System.*out*.println("and then result is " + *function*.andThen(*function1*).apply("Fatimah"));  
 // FATIMAH DEFAULT  
 System.*out*.println("Compose result is " + *function*.compose(*function1*).apply("Fatimah"));

Use Unary Operator over Function if the input and output are of the same type

Use Binary Operator over Function if the inputs(2) and output(1) are of the same type, has twstatic methods- minBy ad maxBy

maxBy take instance of Comparator class – Comarator takes two iputs, and we used Compare to method,maxBy return the max numbr of passed int – same thing w/ minBy

static Comparator<Integer> *comparator* = (a,b) -> a.compareTo(b);  
public static void main(String[] args) {  
 BinaryOperator<Integer> binaryOperator = (a,b) -> a\*b;  
 System.*out*.println(binaryOperator.apply(3,5));  
  
 BinaryOperator<Integer> max = BinaryOperator.*maxBy*(*comparator*);  
 System.*out*.println(max.apply(4,2));

Supplier is posie to Cosumer , it has T get() method.. no iniut anjs return an otput- where consumer void accept(T t)

**Metho Reference: simplify lamda expresion**

Syntax: ClassName::instance-metod name

ClassName::static-methodName

Instance::metodName

static Function<String, String> *toUpperCaseLambda* = (s -> s.toUpperCase());  
  
static Function<String, String> *toUpperCaseMethodReference* = String::toUpperCase;  
public static void main(String[] args) {  
 System.*out*.println(*toUpperCaseLambda*.apply("lambda"));  
 System.*out*.println(*toUpperCaseMethodReference*.apply("Mthod reference"));

Intellijii can cnert from lambda to method reference (alt+enter)

Constructor Reference used only w/ functional interfaces, class name::new

static Supplier<Student> *studentSupplier* = Student::new;

Lambda restrictins in using local variables:

Not allowed to use the same local variable name inside lambda parameter or body.

Not allowed to re-assign value to a local variable (effectively final)

No restriction in instance variables

**Stream API:**

Graphical user interface, text, application, chat or text message

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

// student names activitis in a map  
Map<String, List<String>> namesActivties =  
 StudentDataBase.*getAllStudents*().stream()  
 .collect(Collectors.*toMap*(Student::getName,Student::getActivities));  
System.*out*.println(namesActivties);

// filter

// student names activitis in a map with filter  
Map<String, List<String>> namesActivties =  
 StudentDataBase.*getAllStudents*().stream()  
 .filter((student -> student.getGradeLevel()>=3))  
 .collect(Collectors.*toMap*(Student::getName,Student::getActivities));  
System.*out*.println(namesActivties);

// using predicate inside filter method

Predicate<Student> studentPredicate = (student -> student.getGradeLevel()>=3);  
Predicate<Student> studentPredicate2 = (student -> student.getGpa()>=3.9);  
// student names activitis in a map with filter  
Map<String, List<String>> namesActivties =  
 StudentDataBase.*getAllStudents*().stream()  
 .filter(studentPredicate)  
 .filter(studentPredicate2)  
 .collect(Collectors.*toMap*(Student::getName,Student::getActivities));  
System.*out*.println(namesActivties);

The steam methods that is called betwen stream method and Collect metod ar called intermediate operatons.

Collect method (terminate operations) started th whole process of stream pipeline.(lazeily constructed)

Sream dosen’t allow you t add, remove, or modify elements where in collection you can do that

Elemets in stream can be acessed onlyin sequence, in olection it can b accessed by n order list.ge(2);

Streams ca be traversed only once, colecton n times

You have to poulate collecions w/ values if you want to maipulate it. (eagerl cnstructed)

Debug stream by cling .peak method which takes consumer parameter whch takes input and it’s void

**Stream API operations**

map: convert or transform one type to another

public static List<String> nameList (){  
 return StudentDataBase.*getAllStudents*().stream() //Stream<Student>  
 //Student as an input -> Student name  
 .map(Student::getName) //Stream<String> // map here converting the whole type  
 .map(String::toUpperCase) // Stream<String> // upperCase operation on each input  
 .collect(*toList*()); //List<String>  
}

flatMap: used when every and each element in the stream represent any type of collection

distinct: returns a stream with unique elements

count- returns a long with the total num of elements

sorted- sort the elements in the stream

We can customize sorted operation using Comparator

public static List<Student> sortedStudentNames(){  
 List<Student> students = StudentDataBase.*getAllStudents*().stream() //Stream(Student>  
 .sorted(Comparator.*comparing*(Student::getName)) //Stream(Student>  
 .collect(Collectors.*toList*()); //List<Student>  
 return students;  
}

filter: filter the element in the stream, input to the filter is a Predicate functional interface

reduce- a terminal operation (like collect) used to reduce the contents of a stream to a single value- takes two parameters as an input 1. Default or initial value 2. BinaryOperator<T>

when we use Reduce operation we have two options either to use Optional in the method signature to return empty instead of null, or pass an identity or default value in the reduce function to return it in case of not finding the required value, never use a default value to find max, min value in the stream

limit(n): limits the (n) number of elements in the stream to be processed.

Skip(n): skips the “n” number of elements from the stream.

anyMatch(), allMatch(), noneMatch().. all these elements takes in a predicate as an input and returns a Boolean as an output.

findAny.. return the first element (not Boolean as the previous) that match the specific criteria in filter for example, and stop once found it not executing the rest of elements in the stream

findFirst.. returns the first element in the stream- same as the findAny but the difference will appear in parallel mode

Streams API factory methods:

Of() .. creates a stream of certain values passed to this method.

public static<T> Stream<T> of(T... values) {  
 return Arrays.*stream*(values);  
}

Iterate() .. create infinite stream, we can use limit –

public static<T> Stream<T> iterate(final T seed, final UnaryOperator<T> f)

Generate().. infinite sream, we can use limit -

public static<T> Stream<T> generate(Supplier<? extends T> s)

**Numeric Streams**

The problem when we have list of Integers and want to return an int like sum of it in the stream, is that Integer is a wrapper class which means it will unboxing every element to convert from Integer to int. the solution is to use IntStream.rangeClosed

public static IntStream rangeClosed(int startInclusive, int endInclusive)

range.. exclusive last int

there is also LongStream.reange or .rangeClosed

regarding DoubleStream we don’t have ranges but we can use asDoubleStream

IntStream.*range*(1,50).asDoubleStream().forEach(i-> System.*out*.print(i+ ", "));

Aggregate functions: sum(),max(),min(),average()

Sum return regular type int, double, ..

With max and min it will return OptionalInt or OptionalLong based on your range

Average return OptionalDouble

Boxing(): converting from primitive type to wrapper class type

UnBoxing: opposite

Text

Description automatically generated

Text

Description automatically generated

mapToInt return IntStream which can perform IntStream functions like sum()

IntStream mapToInt(ToIntFunction<? super T> mapper)

// intVlue() return type of int

public int intValue()

mapToObj(): convert a each element in umeric stream to object,

mapToLong(), mapToDouble(): convert a numeric stream to long or double stream

**Terminal Operation**

**Part of collect – meaning will call .collect and inside it will call one of theses terminal operations which avoids additional intermediate operation.**

joining().. collectors performs the string concatenation on the elements in the stream

.collect(Collectors.*joining*())

Counting() collector returns the total nums of elements as result

Mapping():

.collect(*mapping*(Student::getName,*toList*())); // this avoids the additional map intermediate operation.

maxBy(), minBy().. used in conjunction with the Comparator

summingInt(), averagingInt().. average returns double, it has also long and double versions too.

GroupingBy().. the output is Map<K,V>

3 versions: groupingBy(classifier)

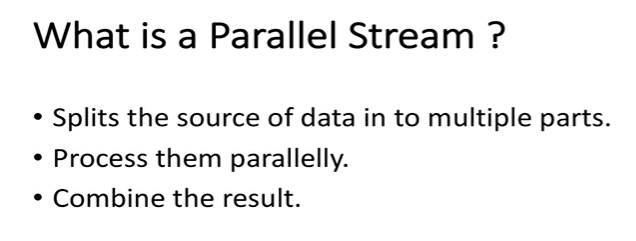
groupingBy(classifier,downstream) – downstream is another any type of collector like we learned like summingInt or groupingBy or toSet()..

groupingBy(classifier,supplier,downstream) – supplier override the default supplier HashMap::new to any other supplier like LinkedHashMap

partitioningBy().. accepts a predicate as an input, return type is Map<K,V> where key return type is Boolean.

Two versions: PartitioningBy(predicate) and (predicate,downstream)

**Parallel Stream**



Graphical user interface, text, application

Description automatically generated

When not to use parallel stream?

Example if you need to sum List of Integers and you pass the list of Integers as it is using boxed function (deals with wrapper classes) so in parallel when using reduce to return int value, it will take extra time to perform unboxing from Integer to int.

2. when having mutable variable. Like if we have Sum class that have variable: total, that have getter and setter, and performSum method that take an iput as parameter and sum it then assign it to total variable (total is getting updated and the result will shown by calling get total), when using parallel to call the getTotal method will give us different and wrong values each time because total variable is getting updated with each split and give wrong number.

**Optional**

The idea behind optional is to give your code a perspective on what operation you are performing instead of just doing a null check.

For Eg., Optional<String> -> This code explains that you are expecting an optional of type String. But in the case of null you dont have any type attached because any object can be null. (empty)

public static Optional<String> getStudentNameOptional(){  
  
 // wrap the actual student object inside optional object  
 //Optional<Student> student = Optional.ofNullable(StudentDataBase.studentSupplier.get());  
 Optional<Student> student = Optional.*ofNullable*(null); // Optional.empty  
  
 if(student.isPresent()){ // checking if optional object have a value of student object using isPresent  
 return student.map(Student::getName); // don't get confused this with streams map method. Both are different.  
 }  
 return Optional.*empty*(); // represents an optional object with no value  
}

public static void main(String[] args) {  
  
 if(*getStudentNameOptional*().isPresent()){  
 System.*out*.println(*getStudentNameOptional*().get().length()); // String which is student name  
 }else{  
 System.*out*.println("Name is returned as empty.");  
 }

}

}

ofNullable: When we call the method of Optional type, it will return the Optional object value, if we want to access the real value of the object inside optional object we need to use .get method, which in case the value was null will return empty object of Optional – but if we use .get will need to check if it is not null by using .isPresent method or it will throw a exception.

Of: will always expect you to have valid value for the input - no null

empty: // if you have an Optional method type, in exception you can return empty optional

// which is the same type as the method return type

orElse: accept actual string, orElseGet: accept a supplier, orElseThrow takes supplier which return an exception

we use flatMap if we need to access Optional object inside optional object

Text

Description automatically generated

**Date/Time APIs**

Text

Description automatically generated

For examples, refer to GitHub

With LocalDate, LocalTime we can get the current date/ ime, get specific value (day,hour, minut,..), modifying the values(plusYears, minus..). We can’t use date functions with time and vise versa.

Period: mainly used to calculate the difference between the two dates. Accept only localDate

Duration: Used to calculate the difference btw the time objects, Compatible with localTime, localDateTime

Instant: represents the time in machine readable format, represents the time in seconds from Jan 01, 1970 to current time as a huge number.

Instant ins = Instatnt.now();

Parse: converting a string to localDate/localTime/localDateTime

Format: opposite