**Java 8**

**Anonymous Class & Lambda Expression**

Say you have an interface (abstract class bhi chalega) say “Par”, we implement it and create a new class say “ChildC” where we override all members from “Par”. In main function we create an instance of ChildC and access all members.

This is usually what we do.

By using Anonymous Class we can ignore creating implemented class and directly override the members of “Par” at the time of Instantiaition and use it the very same way as above. Eg,

package com.company;  
  
interface Par{  
 abstract void meth1();  
}  
// Traditional way  
//class ChildC implements Par{  
// public void meth1(){  
// System.out.println("I'm Meth1");  
// }  
//}  
public class AnonymousClass {  
 public static void main(String[] args) {  
 // Traditional way  
// ChildC obj = new ChildC();  
// obj.meth1();

// Anonymous Class   
 Par obj = new Par() {  
 @Override  
 void meth1() {  
 System.*out*.println("I'm Meth1");  
 }  
 }

obj.meth1();

// Lambda Expression

Par obj1 = ()->{System.*out*.println("I'm Meth1 using Lambda Expression");}; // for a single line we can ignore {}  
obj1.meth1();

// Suppose khuch parameters pass krna hota to

Par obj1 = (a)-> System.*out*.println("I'm Meth1 using Lambda Expression " + a); // No need to mention datatpe of argument Lambda khud se dekh leta h data type  
obj1.meth1(5);

}  
}

Basically by using Anonymous Class we can avoid creating Child class. (Though not recommended)

**Note: Lambda Expression is only meant for Functional Interface (pg 41) while Anonymous class is for both Abstract class and interface.**

Basically, both is used to make code compact and concise (thode me bahut info)

**Functional Interface**(interface with only 1 abstract method)

Functional interfaces can do lot more than you think.

1. **Consumer Interface:**

Represents an operation that accepts input argument/arguments and returns nothing. Unlike most other functional interfaces.

**Abstract method: accept(T paraName)**. Jaisa Consumer interface ka reference type vaisa hi data type input leta h.

**Eg 1,**

Consumer<String> consumer = (s) -> System.*out*.println(s.toUpperCase()); // Consumer reference type String h to accept ko bhi string pass karenge.   
consumer.accept("pAppu");

**Eg 2,**

Consider a class Student with setter,getter & toString() of Student details.

Consider StudentDataBase Class with getAllStudents method which returns List of student details. (List contains details in object form).

package com.learnjava8.functionalinterfaces;  
// All necessary imports  
public class ConsumerInterfaceEg {

public static void printStudentDetail(){  
 Consumer<Student> c2 = (stu) -> System.*out*.println(stu); // here mentioned type is Class to phir accept abstract method bhi object hi lega. Therefore, here stu is a object  
 List<Student> studentList = StudentDataBase.*getAllStudents*();  
 studentList.forEach(c2);

// Let’s see what happens inside forEach Loop upar upar se

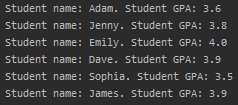
It accepts a Consumer interface or Lambda expression. If passed Consumer, forEach does this internally,

for(List studentList : li) c2.accept(li); // Obviously list end tk iterate karega.

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

**See what’s Inside Class Student and StudentDataBase:** [**https://github.com/dilipsundarraj1/java-8/tree/master/java-8/src/com/learnJava/data**](https://github.com/dilipsundarraj1/java-8/tree/master/java-8/src/com/learnJava/data)

**Consumer Chaining(**ek Consumer ko dusre consumer se jodhna)

In above printStudentDetail if we had done this Output below

Consumer<Student> c2 = (stu) -> System.*out*.print("Student name: " + stu.getName());  
Consumer<Student> c3 = (stu) -> System.*out*.println(". Student GPA: " + stu.getGpa());  
List<Student> studentList = StudentDataBase.*getAllStudents*();

studentList.forEach(c2.andThen(c3)); // Consumer Chaining

**Using Condition in Consumer**

**Note: Agr forEach me consumer pass kare to “accept” automically call ho jaata h but if we pass lamda expression we have to expilcitly call accept**.

In above printStudentDetail if we had done this

Consumer<Student> *c2* = (stu) -> System.*out*.print("Student name: " + stu.getName());  
Consumer<Student> *c3* = (stu) -> System.*out*.println(". Student GPA: " + stu.getGpa());

*studentList*.forEach(stu -> {  
 if(stu.getGpa() > 3.6) *c2*.andThen(*c3*).accept(stu);   
 });  
}

**Method Reference**

**It is used to simply the code written inside {} of Lambda Expression**.

**Syntax :**

**ClassName::methodName**

Ex,

// Using Lambda Expression  
Function<String,String> usingLambda = word -> word.toUpperCase();  
  
// Using Method Reference  
**Function<String,String> usingMethodReference = String::toUpperCase;**

// In Function Interface  
// Using Lambda Expression  
 Function<String,String> usingLambda = word -> word.toUpperCase();  
 System.*out*.println(usingLambda.apply("AadaPadaKonPada"));  
  
 // Using Method Reference  
 **Function<String,String> usingMethodReference = String::toUpperCase;**  
 System.*out*.println(usingMethodReference.apply("bantiWedsBabli"));  
  
 // In Consumer Interface  
 List<Student> studentList = StudentDataBase.*getAllStudents*();  
 // Using Lambda Expression  
 Consumer<Student> usingLambda1 = curStudent -> System.*out*.println(curStudent);  
 studentList.forEach(usingLambda1);  
  
 // Using Method Reference  
 Consumer<Student> usingMethodReference1 = System.*out*::println;  
 studentList.forEach(usingMethodReference1);  
  
 Consumer<Student> usingLambda2 = curStudent -> curStudent.printActivities();  
 studentList.forEach(usingLambda2);  
  
 Consumer<Student> usingMethodReference2 = Student::printActivities;  
 studentList.forEach(usingMethodReference2);

**Implementing Method Reference where it cannot be directly implemented**

public class MethodReferenceIndirectImplementation {  
 // Implementing Method Reference where it cannot be directly implemented  
 static Supplier<Student> *studentSupplier* = () -> new Student("bimlesh",3,4.4,"male", Arrays.*asList*("kirket", "tebal tanis","fud"));  
  
  
 // We cannot directly create method reference for Predicate Interface. But we can create boolean method and use it.  
 // Using Lambda Expression  
 static Predicate<Student> *usingLamda* = student -> student.getGpa() > 3;  
  
 // Using Method Reference Expression  
 static boolean gpaGreater3 (Student s){  
 return s.getGpa() > 3;  
 }  
 static Predicate<Student> *usingMethodReference* = MethodReferenceIndirectImplementation::*gpaGreater3*;  
  
 public static void main(String[] args) {  
 System.*out*.println(*usingLamda*.test(*studentSupplier*.get()));  
 System.*out*.println(*usingMethodReference*.test(*studentSupplier*.get()));  
 }  
}

**Note: It’s not necessary that every code must be compatible with Method Reference. Bss ye aur short code likhne ke liye bana h.**

**Constructor Reference**

**It is used to create a new object of a class but only using Functional Interface. Bss itna hi.**

**Syntax:**

**ClassName::new**

Eg,

Supplier<Student> newStudent = Student::new;  
// Note upar koi input parameter nhi h. Therefore, we must have a Default Constructor in that mentioned class. (here in Student Class).

System.*out*.println(newStudent.get());  
  
Function<String,Student> newStu = Student::new; // Created a new object which takes  
// Note upar ek input parameter h. Therefore, we must have a parameterized constructor with one parameter in that mentioned class (here in Student Class).

System.*out*.println(newStu.apply("champak"));



Output ->

**Limitation using Lamda Expression**

1. It’s not allowed to use the same local variable name as a lambda parameter or insidethe lambda body. Eg,

public static void main(String[] args) {  
 int i = 5;  
 Consumer<Integer> cs = i -> System.*out*.println(i); // Error -> java: variable i is already defined in method main  
 cs.accept(34);  
}

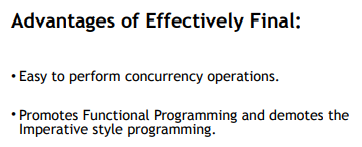
1. It’s not allowed to reassign a new value to a local variable in the lambda expression. Eg,

public static void main(String[] args) {  
 int i = 5;  
 Consumer<Integer> cs = j -> {  
 i =10; // Error dega  
 System.*out*.println(j);  
 };

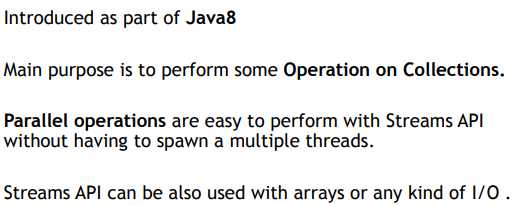
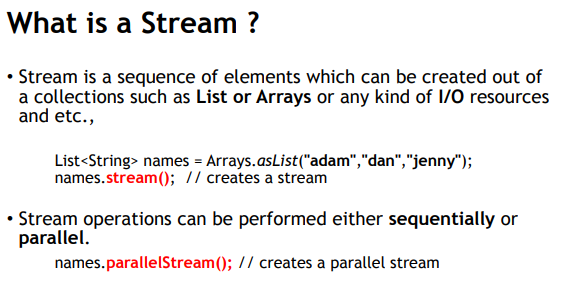
.

But we can reassign Reference Variable in Lambda Body.

public class LamdaRestriction {  
 static int *z* = 6;  
 public static void main(String[] args) {  
 Consumer<Integer> c = x -> {  
 *z*++; // chalega  
 System.*out*.println(i);  
 };  
 }  
}

****

**Effectively Final:** Even though the local variables are not declared final, they are still not allowed to be modified in the Lamda Body. Bss issi ko bolte Effectively Final

**Streams API**

**Terminology in Stream API**

**Stream Pipeline:** All the steps right from creating stream till Terminal Operation.

**Terminal Operation:** A terminal operation in Java 8 is a method that is applied to a stream as the final step. Terminal operations return a result or produce a side effect.

**Note:**  Termina Operation are the last operation in a stream pipeline and cannot be followed by any other operation. The result of a terminal operation can be a single value, a collection, or some other type of object.

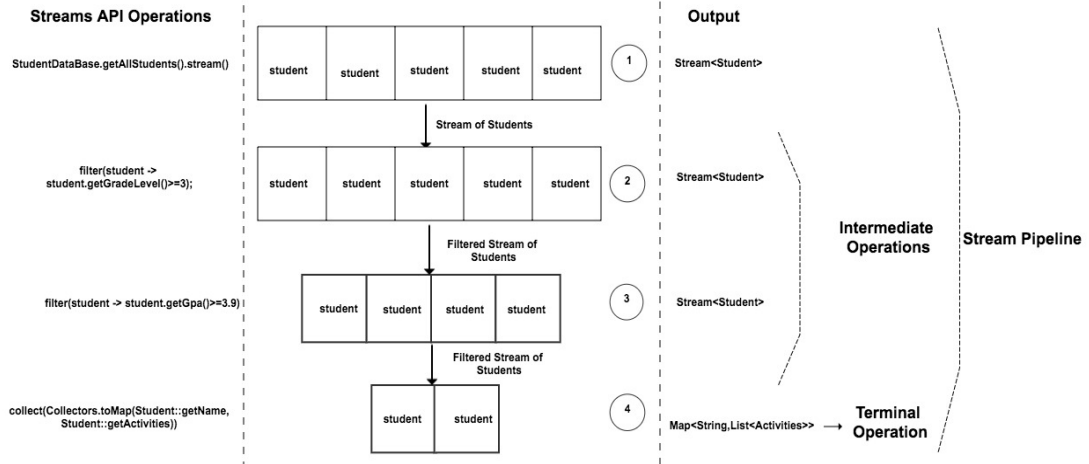
**Intermediate Operations:** All Operations performed after creating Stream and before Terminal Operation (like collect).

**Working of Stream API**

Predicate<Student> gradeGreaterEq3 = student -> student.getGradeLevel() >= 3;  
Predicate<Student> gpaGreaterEq3Dec9 = student -> student.getGpa() >= 3.9;

**Map<String,List<String>> studentNameAndActivities = StudentDataBase.*getAllStudents*().stream()**  // creates stream **.filter(gradeGreaterEq3)**  // creates stream (filter accepts predicate) **.filter(gpaGreaterEq3Dec9)**  // creates stream **.collect(Collectors.*toMap*(Student::getName,Student::getActivities));**  // creates Map  
  
System.*out*.println(studentNameAndActivities);

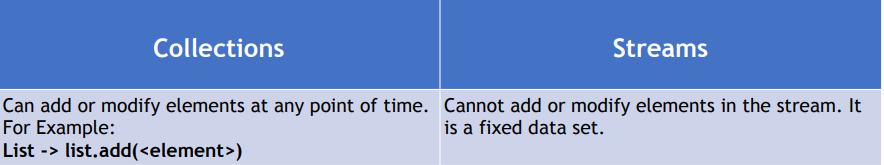
Let’s see the working using above code

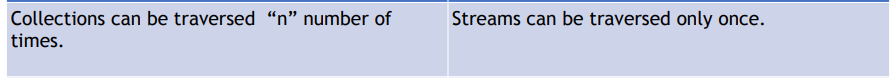


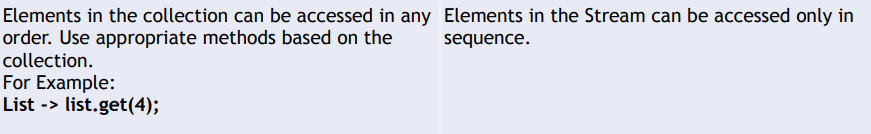
**Note:** Terminal Operation(TO) is the one which starts the complete process, meaning this is the one which starts the pipeline. If the TO method wasn't there, then there is no way this pipeline would have started and it wouldn't have created the output. All of the stream operation above won't happen at all.

**Inshort agar me collect comment krdu to khuch hoga hi nhi (stream bhi create nhi hoga).**

**Note:** Stream API uses declarative way of programming where we only focus on getting the desired output using inbuilt methods and classes without caring what those methods and classes are doing behind the scene.

**Difference b/w Collection and Stream**

****Stream me khuch changes nhi kr sakte. Stream ke help se khuch krna ho to vo kr sakte h

****

List<String> studentName = Arrays.*asList*("Titu","Bitu","Pittu","Gittu");  
Stream<String> nameStream = studentName.stream(); // Created Stream

nameStream.forEach(System.*out*::println);  
nameStream.forEach(System.*out*::println); // will throw error-> stream has already been operated upon or closed

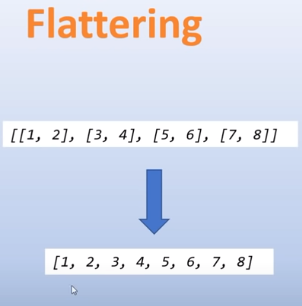
**Stream API Operations**

1. **map()**: converts one type to another.

Eg,

Set<String> studentName = StudentDataBase.*getAllStudents*().stream() // Created stream of Student  
 .map(Student::getName) // Converted Stream of Student to Stream of String  
 .collect(*toSet*()); // created Set. toList() likh deta to list bana deta.

System.*out*.println(studentName);



**Note:** map accepts Function Interface

1. **flatMap()**: **converts one type to another (same as map) + flattening.** (process of transforming and combining nested structures into a single, flat structure)

In general, flatMap is useful when you have a structure like a list of lists, arrays of arrays, or similar nested structures, and you want to flatten it into a single stream of elements.

// With Map  
 List<List<String>> studentActivities = StudentDataBase.*getAllStudents*().stream() // Stream of object  
 .map(Student::getActivities) // Stream of list<String>  
 .collect(*toList*());  
 System.*out*.println(studentActivities);

****Output :

// With flatMap  
 List<String> stuAct = StudentDataBase.*getAllStudents*().stream().  
 map(Student::getActivities).  
 flatMap(List::stream). // map se jo list nikala h uska bhi stream bana ( ye method reference ki tarah kiya h)  
// flatMap(stuActLi -> stuActLi.stream()). (lambda expression me is tarah)   
 collect(*toList*());  
  
 System.*out*.println(stuAct);

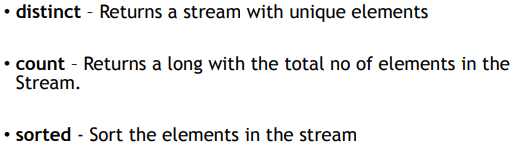
flatMap ka code is tarah bhi likh sakte h

List<String> stuAct1 = StudentDataBase.*getAllStudents*().stream().  
 flatMap(student -> student.getActivities().stream()). // Function interface ne object liya and stream diya  
 collect(*toList*());

**Note:**

collect(Collectors.*toList*());

// upar niche koi bhi chalega  
collect(*toList*());



List<String> stuAct = StudentDataBase.*getAllStudents*().stream().  
 flatMap(stuActi -> stuActi.getActivities().stream()).  
 distinct(). // Stream<String> -> with only distinct values

sorted(). // Stream<String> ko sort kr dega

collect(Collectors.*toList*());

output ->

long disActCount = StudentDataBase.*getAllStudents*().stream().  
 flatMap(stu -> stu.getActivities().stream()).  
 distinct(). // Stream<String> -> with only distinct values  
 count(); // ye dega count (type long rahega)

**Note: Sorted operation 2 type ka para accept krta**

**1) no para (upar ki tarah) 2) accepts Comparator as parameter**

List<Student> studentList = StudentDataBase.*getAllStudents*().stream().  
 sorted(Comparator.*comparing*(Student::getName)). // Note: comparing method accepts function interface  
// sorted(Comparator.comparing(stu -> stu.getName()))

// sorted(Comparator.comparing(Student::getName).reversed()). // descending order me sort krne ke liye

collect(Collectors.*toList*());  
   
studentList.forEach(System.*out*::println);



**filter()**: filter the elements in the stream.

List<Student> stList = StudentDataBase.*getAllStudents*().stream()  
 .filter(st -> st.getGender().equals("female")) // accepts Predicate interface  
 .collect(Collectors.*toList*());  
  
stList.forEach(System.*out*::print);

**reduce()**: This is a terminal operation used to reduce the contents of a stream to a single value.

It accepts both 1 and 2 parameter

For 2 para ->

List<Integer> intList = Arrays.*asList*(1,3,5,7);  
  
int multi = intList.stream()  
 .reduce(1,(a,b) -> a\*b);  
 // Here first 'a' is set to default value (also known as Identity) 1 and multiplied with 'b' = 1 (b = 1 from stream) -> result = 1  
 // 'a' is set to previous output 'b' = 3 (b = 3 from stream)-> result = 3  
 // 'a' = 3 'b' = 5 (from stream) -> result = 15  
 // 'a' = 15 'b' = 7 (from stream) -> result = 105  
  
System.*out*.println(multi);

For 1 para –> only BinaryOperator but than we have store the single value in a Optioanl<T> (Optional was introduced as part of Java 8 to handle the null values that it is going to return) aage padhenge iske baare me.

// For 1 para  
Optional<Integer> multi1 = intList.stream()  
 .reduce((a,b) -> a\*b);  
  
System.*out*.println(multi1.get()); // .get() -> returns the output of an optional

NOTE: Unlike 2 para , the version without an identity assumes that the stream is non-empty, and it uses the first element of the stream as the initial accumulated value. It's important to note that using reduce without an identity might result in an empty Optional if the stream is empty. Better Check whether a result is present before accessing. For that,

if(multi1.isPresent()) System.*out*.println(multi1.get()); // isPresent() -> returns true if result is present.

**limit(n)**: n elements hi process krna stream ke.

**skip(n)**:- n elements skip krke process shuru krna (1 based h, 0 based nhi).

These two function helps to create a sub-stream.

**anyMatch()**:- Returns true if any one of the element matches the predicate, otherwise false.

**allMatch():**- Returns true if all the element in the stream matches the predicate, otherwise false.

**noneMatch()**:- Just opposite to allMatch(). Returns true if none of the element in the stream matches the predicate, otherwise false.

All above 3 functions takes in a predicate as an input and returns a Boolean as an output.

boolean a = StudentDataBase.*getAllStudents*().stream()  
 .**anyMatch**(student -> student.getGpa() > 3);  
  
boolean b = StudentDataBase.*getAllStudents*().stream()  
 .**noneMatch**(student -> student.getGpa() > 3);  
  
boolean c = StudentDataBase.*getAllStudents*().stream()  
 .**allMatch**(student -> student.getGpa() > 3);

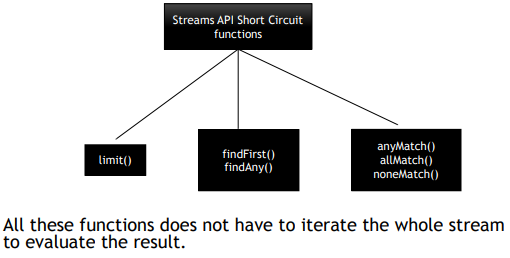
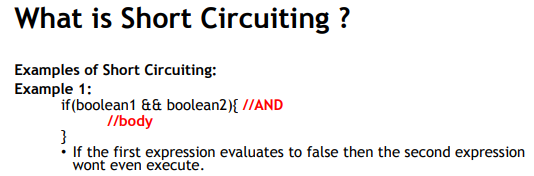
**findFirst()**:- Returns first element in the stream.

**findAny()**:- Returns the first encountered element in the stream.

**NOTE:** Both the functions returns the result of type Optional and are used to find an element in the stream.

Optional<Student> findAny = StudentDataBase.*getAllStudents*().stream()  
 .filter(student -> student.getGpa()>=3.8)  
 .**findAny();**  
  
Optional<Student> findFirst = StudentDataBase.*getAllStudents*().stream()  
 .filter(student -> student.getGpa()>=3.8)  
 .**findFirst();**

Concurrent coxtext me dono output same hi denge but jaha stream parallel banaya vaha output different jaata h very often (aage padhenge parallel stream me).

**Short Circuiting**

**Of():-** Creates a stream of certain values passed to this method.

Stream<String> nameStream = **Stream.*of*("Antu","Bntu","Cntu","Dntu"); // created a stream of string**nameStream.forEach(System.*out*::println);

// OR bina store kiye (pata hi h still chutiya hu me)  
// Stream.of("Antu","Bntu","Cntu","Dntu")  
// .forEach(System.out::println);

**iterate(), generate()**:- Used to create infinite Streams.

// iterate()  
Stream.*iterate*(1, a -> a\*2) // accepts initial value and Unary Operator (ye infinite stream banata jayega)  
 .limit(10) // limit se apn bolre bhai 10 se jyada mt chale jana.  
 .forEach(System.*out*::println);  
// generate  
Supplier<Integer> generateRan = new Random()::nextInt;  
Stream.*generate*(generateRan) // accepts supplier and infinte stream banata jayega agr limit set nhi kiya to  
 .limit(10)  
 .forEach(System.*out*::println);

**Numeric Stream**

It is recommended to use Numeric Stream over normal Stream when working with numeric data.

List<Integer> integerList = Arrays.*asList*(1,2,3,4,5,6);  
int sum = integerList.stream() // Stream<Integer>  
 .reduce(0,(x,y)->x+y); // unboxing to convert Integer to an int.

Ab yaha dekh apn List<Integer> ka stream bana rahe h and Integer is a wrapper class for primitive type int, to stream banate time baar baar unboking krna padta h (Integer to int), which is a extra overhead and can be avoided using Numeric Stream.

Same chiz with Numeric Stream

int sum = IntStream.*rangeClosed*(1,6) // rangeClosed takes beginning and ending index and returns stream thus stream banane ki jarurat nhi. Aage dekhenge isko bhi   
 .sum(); // sum method add kr dega elements ko

**Numeric Stream - Ranges**

**IntStream:**

**IntStream.range(1,50)** -> Returns an IntStream of 49 elements from **1 to 49.**

**IntStream.rangeClosed(1,50)** -> Returns an IntStream of 50 elements from **1 to 50.**

**LongStream:** ditto same as IntStream bss Int hata ke Long likh de

**DoubleStream**: - It does not support the range ()and rangeClosed().

IntStream intStream = IntStream.*range*(1,50);  
System.*out*.println(intStream.count()); // count returns no of elements in the stream.  
  
IntStream.*range*(1,50).forEach(curInt -> System.*out*.print(curInt + ", "));  
System.*out*.println();  
LongStream.*range*(1,50).forEach(curInt -> System.*out*.print(curInt + ", "));  
System.*out*.println();  
  
IntStream.*rangeClosed*(1,50).forEach(curInt -> System.*out*.print(curInt + ", "));  
System.*out*.println();  
LongStream.*rangeClosed*(1,50).forEach(curInt -> System.*out*.print(curInt + ", "));  
System.*out*.println();  
  
// We can use range and rangeClosed method for Double Stream too  
IntStream.*range*(1,49).asDoubleStream().forEach(curDouble -> System.*out*.print(curDouble + ", "));

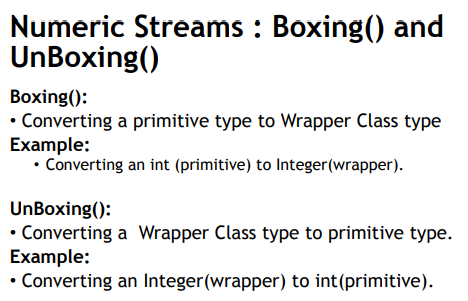
**Numeric Stream – Aggregate Function**

1) **sum()** 2) **max()** 3) **min()** 4) **average()**

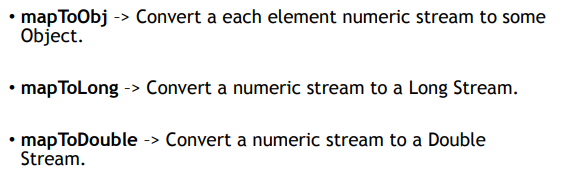
Jaisa naam vaisa kaam. **Note:**  **Aggregate Function returns Optional** (aage padhenge iske baare)

OptionalInt min = IntStream.*rangeClosed*(1,40).min(); // O/p -> 1  
OptionalInt max = IntStream.*rangeClosed*(1,40).max(); // max and min returns OptionalDouble // O/p -> 40  
OptionalDouble average = IntStream.*rangeClosed*(1,40).average(); // average returns OptionalDouble // O/p -> 20.5  
long countElInStream = IntStream.*rangeClosed*(1,40).count(); // count returns long // O/p -> 40  
  
System.*out*.println(min.isPresent() ? min.getAsInt() : "no stream");  
System.*out*.println(max.getAsInt() + " " + average.getAsDouble() + " " + countElInStream);

**Numeric Stream – Boxing and Unboxing**



List<Integer> integerList = IntStream.*rangeClosed*(1,25)  
 .boxed() //Stream<Integer> (Stream of wrapper class)  
 .collect(Collectors.*toList*());  
  
int sum = integerList.stream()  
 .mapToInt(Integer::intValue) // returns IntStream of int primitive type  
 .sum();

****

public static List<Integer> mapToObj(){  
 List<Integer> integerList = IntStream.*rangeClosed*(1,5)  
 .mapToObj((i)-> {  
 return new Integer(i);  
 })  
 .collect(*toList*());  
 return integerList;  
}  
  
public static double mapToDouble(){  
 return IntStream.*rangeClosed*(1,5)  
 .mapToDouble((i) -> i) // IntStream to DoubleStream  
 .sum();  
}  
  
public static long mapToLong(){  
 return IntStream.*rangeClosed*(1,5)  
 .mapToLong((i) -> i) // IntStream to LongStream  
 .sum();  
  
}  
public static void main(String[] args) {  
 System.*out*.println("mapToObj : " + *mapToObj*());  
 System.*out*.println("mapToDouble() : " + *mapToDouble*());  
 System.*out*.println("mapToLong() : " + *mapToLong*());  
}