1.Once stream is used u cant use it again. Its same like iterator

Couple of short circuiting operations are anyMatch() and findFirst().

The Diff between Collections and stream is, stream wont store data , whereas collection stores data

**Intermediate operations** do not actually perform any filtering, but instead creates a new stream that, when traversed, contains the elements of the initial stream that match the given predicate.

## Reference links

<https://stackoverflow.com/questions/68692822/how-to-get-first-and-last-day-in-specific-week-in-month>

<https://kodejava.org/how-do-i-get-the-first-sunday-of-the-year-in-java/>

<https://www.baeldung.com/java-8-date-time-intro>

for legend

<https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html>

<https://www.baeldung.com/java-string-to-date#:~:text=Date%20object%3A,parse(dateInString)%3B>

## Tips while using streams

1. Instead of passing lambda expression pass the method reference

# Stream

Stream is nothing but a flow of data or sequence of data

Ex:- a factory assembly line

 **Intermediate operations:** These operations transform or filter the stream and return a new stream. Examples: map, filter, sorted, distinct, peek, limit, skip.

 **Terminal operations:** These operations produce a result or perform a side effect and terminate the stream. Examples: forEach, collect, reduce, count, min, max, anyMatch, allMatch, noneMatch, findFirst, findAny.



Once that crosses a person it wont come back, means stream once it is used , we cant reuse

## Intermediate Operations - Types

Intermediate operations are further divided as

* stateless operations
* stateful operations

**Stateless operations**, such as filter and map, retain no state from previously seen element when processing a new element. Each element can be processed independent of operations on other elements.

Stream mapToInt(), Stream mapToDouble()

**Stateful operations**, such as distinct and sorted, may incorporate state from previously seen elements when processing new elements.

## Terminal Operation

Terminal operation methods🡪 forEach(), collect(), match(), count() and reduce(), findFirst(), findAny()

Terminal operations may traverse the stream to produce a result or a side-effect.

After the terminal operation is performed, the stream pipeline is considered consumed, and can no longer be used.

Terminal operations are eager, completing their traversal of the data source and processing of the pipeline before returning

Few examples for terminal operation

Until a terminal operation is seen stream will not be executed

Create a Stream

Stream fromArray = Stream.of(1, 2, 3); // count = 2

Stream randoms = Stream.generate(Math::random);

StartsWith Demo

==============

String firstMatchedName = memberNames.stream() .filter((s) -> s.startsWith("L")) .findFirst().get();

# Methods in stream

Allmatch(Predicate p), anyMatch(Predicate p),noneMatch(Predicate p)

mapToInt

findFirst()

List<String> list= Arrays.*asList*("mani","santu","sai");  
Optional<String> s1 = list.stream().filter(s -> s.startsWith("s")).findFirst();  
System.*out*.println(s1.get()); // it will give santu

|  |  |
| --- | --- |
| **To create a stream**   1. Stream empty = Stream.empty(); // count = 0 2. Stream singleElement = Stream.of(1,2); // count = 1   Stream s = Stream.of("monkey", "gorilla", "bonobo");   1. new ArrayList().stream()// from list also we can create stream 2. Arrays.stream(new int[]{}) | **To generate continuous flow of elements**  Stream<T> generate(Supplier<T> s) // this supplier will be invoked continuously  Stream<T> iterate(T seedInitialElement, UnaryOperator<T> f)   1. Stream<String> randomVal =   Stream.generate(()->UUID.randomUUID().toString()).limit(10000); randomVal.forEach(e-> System.out.println(e));  Stream.generate(Math::random)  Generate method needs a supplier that’s it I passed a supplier interface  IntStream.range(1,100).forEach(e-> System.out.println(e));   1. *Stream*.*iterate*(2,*e*->*e*+2).limit(20).forEach(System.***out***::println); 2. Random r=new Random(); *Stream*.*generate*(()->r.nextInt()).limit(200).forEach(System.***out***::println); |
| Stream<T> filter(Predicate predicate) | The filter() method returns a Stream with elements that match a given expression  If condition is true only those elements will be considered |
| Stream distinct() | It will give unique elements (based on equals method)  Stream s = Stream.of("duck", "duck", "duck", "goose"); s.distinct().forEach(System.out::print); // duckgoose |
| min() and max()  // Find the employee with the minimum salary  Optional<Employee> lowestPaidEmployee = employees.stream()  .min(Comparator.comparing(Employee::getSalary)); | //sort using length of each string  Stream s = Stream.of("monkey", "ape", "bonobo");  Optional min = s.min((s1, s2) -> s1.length()—s2.length());  min.ifPresent(System.out::println); // ape  or // sort using alphabetical order  *Stream*<String> s =*Stream*.*of*("mohan","rama","moanideep","sai","santu"); Optional<String> min = s.min((*e1*, *e2*) -> *e1*.compareTo(*e2*)); System.***out***.println("min value is -->"+min.orElse("no value ")); |
| Count() | To count the ele in the stream  Stream.of("monkey", "gorilla", "bonobo").count();// 3 |
| limit() and skip()  limit(100)- means limits to 100 elements alone even if 10,000 ele are there  skip(20) – skips firsts 20 elements | Limit()  To turn an infinite stream to finite stream  Stream.generate(()->UUID.randomUUID().toString()).limit(10000); //if limit is not used here, it will cont generate the elements until u kill the program  *getIntStream*(100).skip(10).limit(50).forEach(System.***out***::println); |
| Related to searching  findAny() /findFirst()  Optional<T> findAny()  Optional<T> findFirst()  boolean anyMatch(Predicate predicate)  boolean allMatch(Predicate predicate)  boolean noneMatch(Predicate predicate) | findFirst() will return the ele among the filtered stream  whereas anyMatch(predicate p) will return only true or false  Both findAny() and findFirst() are same to get the elements we cant pass any arg to these methods  Optional findAny() // useful if we are using parallel stream  Optional findFirst() // returns the first element  *Stream*<String> s = *getAStream*().filter(*s2*->*s2*.length()>6); *Stream*<String> s1 = *getAStream*().filter(*s2*->*s2*.length()>6); System.***out***.printf("findany -->%s , findFirst --> %s",s.findAny(),s1.findFirst());// both returns same ele  anyMatch(p) // returns true if any ele satisfies the condition  allMatch(p) // return true only when all ele satisfies the condition |
| void forEach(Consumer action) | This is the only terminal operation method that returns void |
|  |  |
| Optional<*T*> reduce(*BinaryOperator*<*T*> *bop*);  Optional<*T*> reduce(defaultInitialValue,*BinaryOperator*<*T*> *bop*); | The reduce() method is used to reduce a stream into a single object.  BinaryOperator op = (a, b) -> a \* b; //binary operator is bifunction with same return type  Stream stream = Stream.of(3, 5, 6);  System.out.println(stream.reduce(1, op, op)); // 90 |
| collect()-  collect(), reduce() almost same but prefer collect()  collect is called mutable reduction operation(A mutable reduction is one in which the reduced value is a mutable result container, such as an ArrayList, and elements are incorporated by updating the state of the result rather than by replacing the result.) | Collect method signatures  R collect(Supplier supplier, BiConsumer accumulator, BiConsumer combiner)  R collect(Collector collector)  Stream stream = Stream.of("w", "o", "l", "f");  StringBuilder word = stream.collect(StringBuilder::new, StringBuilder::append, StringBuilder:append)//flow  Sometimes We can perform this reduction operation using reduce method also  Ex-2  Set set = stream.collect(Collectors.toSet()); |
| Map(Function f) - take one ele and return one ele  *IntStream* mapToInt(*ToIntFunction* *mapper*);  *DoubleStream* mapToDouble(*ToDoubleFunction* *mapper*);  *LongStream* mapToLong(*ToLongFunction mapper*); | <*R*> *Stream*<*R*> map(*Function*<? super *T*, ? extends *R*> *mapper*);  *IntStream* mapToInt(*ToIntFunction*<? super *T*> *mapper*); //Gives us IntStream to perform sum(),avg()  *LongStream* mapToLong(*ToLongFunction*<? super *T*> *mapper*);  *DoubleStream* mapToDouble(*ToDoubleFunction*<? super *T*> *mapper*); |
| flatMap()  *flatmap takes 1 ele returns only stream* | This is also Used to transform one element to another type,  This is used where each ele in the list represents multiple elements (list)  *List*<String> collect = emps  .map(Employee::getHobbies)*//map takes 1 ele and returns 1 ele* .flatMap(*e* -> *e*.stream()) *// flatmap takes 1 ele returns only stream* .collect(Collectors.*toList*());  Result  [eating, sleeping, reading, surfing, dancing, singing] |
| Sorted()  Sorted(Comparator c) | Comparator c = Comparator.comparing();  *List*<Employee> collect = StreamsPrac.*getEmpAsStream*()  .sorted(*Comparator*.*comparing*(*e*->*e*.getSalary())) *// .sorted(Comparator.comparing(Employee::getSalary)) // .sorted((e1,e2)->e1.getSalary()-e2.getSalary()) // .sorted(Comparator.comparingInt(Employee::getSalary))* .collect(Collectors.*toList*());  System.***out***.println(collect); |

### Reduce()

–reduce() is to reduce the stream into single element or if u want to reduce list into 1 element u can use this ex:- for addition, multiplication

reduce(BinaryOperator<T> accumulator)

[**reduce**](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#reduce-T-java.util.function.BinaryOperator-)([**T**](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html) identity, [**BinaryOperator**](https://docs.oracle.com/javase/8/docs/api/java/util/function/BinaryOperator.html)<[**T**](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html)> accumulator) //**T is the initial value & BinaryOperator- takes 2 inputs and return 1 output, all 3 are of same type**

**from 2nd iteration onwards, among 2 elements 1 ele will be ele in the array, 2nd ele will be ele in the result of previous operation**

now among list of employees we want to find the smallest one or we want to find the biggest one

|  |  |
| --- | --- |
| public static *Stream*<Employee> getEmpAsStream(){  Employee e1=new Employee(1,"Mani");  Employee e2=new Employee(2,"Sai");  Employee e3=new Employee(2,"Santu");  return *Stream*.*of*(e1,e2,e3); } | *Stream*<Employee> empAsStream = *getEmpAsStream*(); *// now our aim is to reduce this stream into single obj whose name is so much big* Optional<Employee> reduce = empAsStream.reduce((*s1*, *s2*) -> *s1*.getEmpName().length()> *s2*.getEmpName().length()?*s1*:*s2*); System.***out***.println("reduced emp is "+reduce.get()); |

Here we are cal the sum of all salaries to one , whereas this can be done using Instream.sum() also

*Stream*<Employee> empSal = *getEmpAsStream*();  
*BinaryOperator*<Integer> bop=(*e1*,*e2*)->*e1*.intValue()+*e2*.intValue();  
System.***out***.println("reduced salary is -->"+empSal.map(*e*->*e*.getSalary()).reduce(bop));

1. We can perform sum of all salaries using reduce
2. We can perform sum of multiplication of all salaries using reduce
3. U can get max number using reduce() -without sorting – if ur aim is to get max num why to sort 1 crore elements In the list and then get the last element

*Stream*<Employee> emps = StreamsPrac.*getEmpAsStream*();  
System.***out***.println("max value is "+emps.mapToInt(*e*->*e*.getSalary())  
 .reduce((*e1*,*e2*)->*e1*>*e2*?*e1*:*e2*).getAsInt());

1. U can get min number using reduce() but u can do this using Intstream().min()/ max() but here it will happen without sorting it will just get the ele by comparing

*Stream*<Employee> emps = StreamsPrac.*getEmpAsStream*();  
OptionalInt min = emps.mapToInt(*e* -> *e*.getSalary()).reduce((*e1*, *e2*) -> {  
 if (*e1* < *e2*) {  
 System.***out***.println("in if condition");  
 return *e1*;  
 } else {  
 System.***out***.println("in else condition");  
 return *e2*;  
 }  
});  
System.***out***.println("min ele with reduce() is "+min.getAsInt());

1. S

### NumericStreams

Predefined classes 🡪 IntStream,DoubleStream,LongStream

|  |  |
| --- | --- |
| System.***out***.println(*IntStream* .*rangeClosed*(1,100) *// returns a stream of values* .sum());*//performs sum operation* | *IntStream*.*rangeClosed*(1, 10).mapToLong() // to convert to longStream *IntStream*.*rangeClosed*(1, 10).mapToDouble() //to convert to Doublestream *IntStream*.*rangeClosed*(1, 10).mapToObj() //to convert to ibj |
| IntStream mapToInt(ToIntFunction<? super T> mapper);  List.stream().mapToInt() //will gives us IntStream |  |
| IntStream.range(1,50)// it will gives us 1to 49  IntStream.rangeClosed(1,50) it will gives values from 1 to50 |  |
| *IntStream*.*rangeClosed*(1,20).asDoubleStream()  //in double stream we don’t have range() or rangeClosed() we can convert from intstream to doublestream |  |
| Max()  Min()  Average() |  |
|  |  |
|  |  |
|  |  |

Boxing (int value to Integer) & unboxing (Integer to int value)

|  |  |
| --- | --- |
| For boxing | For unboxing |
| public static void intAutoBoxingStreamDemo(){  *List*<Integer> collect = *IntStream*.*rangeClosed*(1, 10).boxed()  .collect(Collectors.*toList*());  System.***out***.println("boxed output is "+collect.toString()); } | public static void intStreamAutoUnBoxingDemo(){  *List*<Integer> collect = *IntStream*.*rangeClosed*(1, 10).boxed().collect(Collectors.*toList*());  int sum = collect.stream().mapToInt(Integer::intValue).sum();*//Here we are doing unboxing from integer to int* System.***out***.println("unboxed value sum is "+sum); } |

### **Collectors.Collect()**

Collect is called mutable reduction operation to reduce entire collection into single value (A mutable reduction is one in which the reduced value is a mutable result container, such as an ArrayList,StringBuffer and elements are incorporated by updating the state of the result rather than by replacing the result.)

Signature

R collect(Supplier supplier, BiConsumer accumulator, BiConsumer combiner)

Ex:- R collect(like initial value obj ref, BiConsumer accumulator, BiConsumer combiner)

R collect(Collector collector)

Stream stream = Stream.of("w", "o", "l", "f");

StringBuilder word = stream.collect(StringBuilder::new, StringBuilder::append, StringBuilder:append)

The first parameter is a Supplier that creates the object that will store the results as we collect data. Remember that a Supplier doesn’t take any parameters and returns a value. In this case, it constructs a new StringBuilder.

**The second parameter is a BiConsumer, which takes two parameters and doesn’t return anything (as per muting the same object). It is responsible for adding one more element to the data collection. In this example, it appends the next String to the StringBuilder.**

The final parameter is another BiConsumer. It is responsible for taking two data collections and merging them. **This is useful when we are processing in parallel**. Two smaller collections are formed and then merged into one

1. Collectors.toSet(),Collectors.toList()

Stream stream = Stream.of("w", "o", "l", "f");

Set set = stream.collect(Collectors.toSet());

System.out.println(set); // [f, w, l, o]

Stream stream = Stream.of("w", "o", "l", "f");

TreeSet set = stream.collect(TreeSet::new, TreeSet::add, TreeSet::addAll); System.out.println(set); // [f, l, o, w]

The collector has three parts as before. The supplier creates an empty TreeSet. The accumulator adds a single String from the Stream to the TreeSet. The combiner adds all of the elements of one TreeSet to another in case the operations were done in parallel and need to be merged.

Stream stream = Stream.of("w", "o", "l", "f"); TreeSet set = stream.collect(Collectors.toCollection(TreeSet::new)); System.out.println(set); // [f, l, o, w] If we didn’t need the set to be sorted, we could make the code even shorter: Stream stream = Stream.of("w", "o", "l", "f"); Set set = stream.collect(Collectors.toSet()); System.out.println(set); // [f, w, l, o]

StringBuilder collect = *Stream*.*of*("f", "l", "o", "w")  
 .collect(StringBuilder::new, StringBuilder::append, StringBuilder::append);  
System.***out***.println(collect);*// flow*

1. Collectors.joining(", ")

*print*(*getNamesAsStream*().collect(Collectors.*joining*()));*// omsairam  
print*(*getNamesAsStream*().collect(Collectors.*joining*("-")));*// om-sai-ram  
print*(*getNamesAsStream*().collect(Collectors.*joining*("-","(",")"))); *//(om-sai-ram)*

1. Collectors.groupingBy(Employee::getDepartment)

Collectors.groupingBy(Employee::getDepartment, Collectors.summingInt(Employee::getSalary)

* groupingBy(Function f) or groupingBy(Classifier )
* groupingBy(Function f, Collector dc) or groupingBy(Classifier , downstream)
* groupingBy(Function f, Supplier s, Collector dc) groupingBy(Classifier , supplier ,downstream)

group by 1 param == after grouping it will look as below, key will be Gender and value will be list of employees

|  |  |
| --- | --- |
| Male | List of all employees |
| Female | List of all female employees |

*//Here we are grouping by gender - only 2 groups named female and male will be formed  
print*(StudentDataBase.*getAllStudentsAsStream*().collect(Collectors.*groupingBy*(*e* -> *e*.getGender())));  
*print*(StudentDataBase.*getAllStudentsAsStream*().collect(Collectors.*groupingBy*(Student::getGender)));

group by with with 2 params – when to choose this?

If u want to perform any other operation on above maps values column then we should choose this

ex:- on that right side list if u want the top employee or emp with highest num of notebooks… or if u want to perform any operation on right side list then chose this

public static void groupByDept() {  
 *//group by dept  
 Map*<*Enum*, *List*<CollegeStudent>> collect = CollegeStudent.*getCollegeStudents*().collect(Collectors.*groupingBy*(CollegeStudent::getDeptName));  
 System.***out***.println(collect);

*//groupby dept and print who has highest marks in each dept  
 Map*<*DeptName*, Optional<CollegeStudent>> highestMarksGroupByDept = CollegeStudent.*getCollegeStudents*().collect(Collectors.*groupingBy*(  
 CollegeStudent::getDeptName,  
 Collectors.*maxBy*(*Comparator*.*comparing*(CollegeStudent::getMarks))  
 ));  
 System.***out***.println("highestMarksGroupByDept-->"+highestMarksGroupByDept);  
 *//groupby dept and i want each again groupby gender and i want who is highest in male and who is highest in females  
 //gorupby dept and gender  
 Map*<*DeptName*, *Map*<String, Optional<CollegeStudent>>> groupByDeptGender = CollegeStudent.*getCollegeStudents*().collect(Collectors.*groupingBy*( *// Function, collector* CollegeStudent::getDeptName,*// Function* Collectors.*groupingBy*( *// collector* CollegeStudent::getGender, *//Function* Collectors.*maxBy*(*Comparator*.*comparing*(CollegeStudent::getMarks)) *//Collector* )  
  
 ));  
 System.***out***.println("groupByDeptGender"+groupByDeptGender);  
}

{CSE=[CollegeStudent(name=Rama, deptName=CSE, marks=25, gender=Female), CollegeStudent(name=SAI gadu, deptName=CSE, marks=65, gender=Male), CollegeStudent(name=rangayya, deptName=CSE, marks=85, gender=Male)], EEE=[CollegeStudent(name=Santu, deptName=EEE, marks=95, gender=Female), CollegeStudent(name=charan, deptName=EEE, marks=85, gender=Male), CollegeStudent(name=Loopa, deptName=EEE, marks=45, gender=Female)], ECE=[CollegeStudent(name=Mani, deptName=ECE, marks=95, gender=Male), CollegeStudent(name=KhadarVali, deptName=ECE, marks=90, gender=Male), CollegeStudent(name=DrSarala, deptName=ECE, marks=99, gender=Female)]}

highestMarksGroupByDept-->{CSE=Optional[CollegeStudent(name=rangayya, deptName=CSE, marks=85, gender=Male)], EEE=Optional[CollegeStudent(name=Santu, deptName=EEE, marks=95, gender=Female)], ECE=Optional[CollegeStudent(name=DrSarala, deptName=ECE, marks=99, gender=Female)]}

groupByDeptGender{CSE={Male=Optional[CollegeStudent(name=rangayya, deptName=CSE, marks=85, gender=Male)], Female=Optional[CollegeStudent(name=Rama, deptName=CSE, marks=25, gender=Female)]}, EEE={Male=Optional[CollegeStudent(name=charan, deptName=EEE, marks=85, gender=Male)], Female=Optional[CollegeStudent(name=Santu, deptName=EEE, marks=95, gender=Female)]}, ECE={Female=Optional[CollegeStudent(name=DrSarala, deptName=ECE, marks=99, gender=Female)], Male=Optional[CollegeStudent(name=Mani, deptName=ECE, marks=95, gender=Male)]}}

*print*("male highest notebooks and female highest notebooks-->"+  
 StudentDataBase.*getAllStudentsAsStream*().collect(  
 Collectors.*groupingBy*(Student::getGender,Collectors.*maxBy*(*Comparator*.*comparing*(Student::getNoteBooks))))  
 );

*print*("Top gpa in each grade"+StudentDataBase.*getAllStudentsAsStream*().collect(

Collectors.*groupingBy*(Student::getGradeLevel,Collectors.*maxBy*(*Comparator*.*comparing*(Student::getGpa))  
 )));

//Top gpa in each grade{

2=Optional[Student{name='Jenny', gradeLevel=2, gpa=3.8, }],

3=Optional[Student{name='Emily', gradeLevel=3, gpa=4.0],

4=Optional[Student{name='James', gradeLevel=4, gpa=3.9]}

*//Instead of collecting into optional, if u want direct employee object then use collectingAndThen  
Map*<Integer, Student> collect1 = StudentDataBase.*getAllStudentsAsStream*().collect(Collectors.*groupingBy*(  
 Student::getGradeLevel, Collectors.*collectingAndThen*(Collectors.*maxBy*(*Comparator*.*comparing*(Student::getGpa)), Optional::get)  
));  
System.***out***.println("collectors.groupby collecting and then"+collect1);

*print*("grouping gender"+StudentDataBase.*getAllStudentsAsStream*().collect(Collectors.*groupingBy*(Student::getGender,Collectors.*toList*())));  
*print*("grouping gender"+StudentDataBase.*getAllStudentsAsStream*().collect(Collectors.*groupingBy*(Student::getGender)));  
*//The below will group the records by gender and provides value as count of notebooks gender{female=33, male=47}  
print*("grouping gender"+StudentDataBase.*getAllStudentsAsStream*().collect(Collectors.*groupingBy*(Student::getGender, Collectors.*summingInt*(*e*->*e*.getNoteBooks()))));  
*//The below will group the records by gender and provides value as sum of their gpa gender{female=11.3, male=11.5}  
print*("grouping gender"+StudentDataBase.*getAllStudentsAsStream*().collect(Collectors.*groupingBy*(Student::getGender,Collectors.*summingDouble*(Student::getGpa))));

1. Collectors.partitioningBy(s -> s.getGrade() >= PASS\_THRESHOLD)

*Collector*<*T*, ?, *Map*<Boolean, *List*<*T*>>> partitioningBy(*Predicate*<? super *T*> *predicate*)

*Collector*<*T*, ?, *Map*<Boolean, *D*>> partitioningBy(*Predicate*<? super *T*> *predicate*, *Collector*<? super *T*, *A*, *D*> *downstream*)

This is also like grouping by , where partitioning by will create only 2 groups, condition failed or success

|  |  |
| --- | --- |
| Condition failed == false | List<Employees> |
| Condition success == true | List<Employees> |

After partitioning still if u want to perform some operations on values column in map then choose 2 partitioning by with 2 params

Ex: - partitioning by male and who has highest gpa in male and highest gpa in female

public static void partitioningWith1Param() {  
 *Predicate*<Student> predicate=(*p*)->"male".equalsIgnoreCase(*p*.getGender());  
 *Map*<Boolean, *List*<Student>> collect = StudentDataBase.*getAllStudentsAsStream*().collect(Collectors.*partitioningBy*(predicate));  
 System.***out***.println(collect);  
}  
public static void partitioningWith2Param(){  
 *Predicate*<Student> predicate=(*p*)->"male".equalsIgnoreCase(*p*.getGender());  
 *Map*<Boolean, Optional<Student>> collect = StudentDataBase.*getAllStudentsAsStream*().collect(Collectors.*partitioningBy*(  
 predicate, Collectors.*maxBy*(*Comparator*.*comparing*(Student::getGpa))  
 ));  
 System.***out***.println(collect);  
}

Below are the collectors methods which has already good alternatives but it will be used while grouping by

1. Collectors.mappingBy(Function,collector)

*Instead of collectors.mapping/// we can use Stream.map() method*

*print*(*getEmpAsStream*().map(Employee::getEmpName).collect(Collectors.*toList*())); *//[Mani, Sai, Santu]  
print*(*getEmpAsStream*().collect(Collectors.*mapping*(Employee::getEmpName,Collectors.*toList*()))); *//[Mani, Sai, Santu]*

1. Collectors.maxBy(Comparator c) or Collectors.minBy(Comparator c) 🡪 finds largest or smallest elements

Just to find smallest and largest u can use direct Stream.min() & Stream.max() methods these Collectors.minBy() is not even recommended by intellij

public static void maxByDemo() {  
 *//Both below returns same whereas Stream.max(Comparator c) is best  
 print*("min value is " + *getEmpAsStream*().collect(Collectors.*minBy*(*Comparator*.*comparing*(Employee::getSalary))));  
 *print*("min value is " + *getEmpAsStream*().min(*Comparator*.*comparing*(Employee::getSalary)));  
}  
  
public static void minByDemo() {  
 *//Both below returns same whereas Stream.min(Comparator c) is best  
 print*("min value is " + *getEmpAsStream*().collect(Collectors.*maxBy*(*Comparator*.*comparing*(Employee::getSalary))));  
 *print*("min value is " + *getEmpAsStream*().max(*Comparator*.*comparing*(Employee::getSalary)));  
}

1. Collectors.summingInt() & Collectors.averagingInt()

Instead of these 2 collectors.summingInt() and averaginngInt() methods use IntStream.sum() & IntStream.average()

public static void summingInt() {  
 *//Both below returns same whereas min(Comparator c) is best  
 print*("sum value is " + *getEmpAsStream*().collect(Collectors.*summingInt*(Employee::getSalary))); *//sum value is 2800  
 print*("sum value is " + *getEmpAsStream*().mapToInt(Employee::getSalary).sum()); *//sum value is 2800* }  
public static void averagingInt() {  
 *//Both below returns same whereas min(Comparator c) is best  
 print*("avg value is " + *getEmpAsStream*().collect(Collectors.*averagingInt*(Employee::getSalary)).doubleValue()); *//700  
 print*("avg value is " + *getEmpAsStream*().mapToInt(Employee::getSalary).average()); *//700* }

### Sorting

Note:- always while sorting make sure to remove the nulls by doing filter operation, because if null is there then u will get null pointer exception

Or atleast use Comparator.nullsFirst().. like these methods

1. list.sort(Comparator c)
2. Collections.sort(Comparator c)
3. Stream.sorted(Comparator c)

Comparator c=Comparator.comparing(Function f)

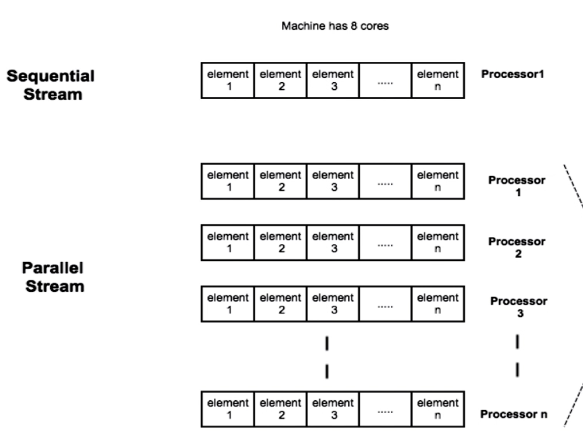
|  |  |
| --- | --- |
| Comparator.nullFirst(Comparator c) // so that null will appear first | Comparator.nullLast(Comparator c) // so that nulls will appear at last |

private static void sortingAfterFilteringNulls() {  
 *Comparator*<CollegeStudent> comparator = *Comparator*.*comparing*(CollegeStudent::getName);  
 *List*<CollegeStudent> collect = CollegeStudent.*getCollegeStudents*()  
 .filter(*e* -> Objects.*nonNull*(*e*))  
 .sorted(comparator).collect(Collectors.*toList*());  
 System.***out***.println(collect);  
}  
  
public static void sortingWithNullsFirst() {  
 *Comparator*<CollegeStudent> c1 = *Comparator*.*comparing*(CollegeStudent::getName);  
 *Comparator*<CollegeStudent> compWithNullsFirst = *Comparator*.*nullsFirst*(c1);  
 *List*<CollegeStudent> sortedWithNUllsFirst = CollegeStudent.*getCollegeStudents*().sorted(compWithNullsFirst).collect(Collectors.*toList*());  
 System.***out***.println("sorting with nulls first"+sortedWithNUllsFirst);  
}  
public static void sortingWithoutNulls() {  
 *//sorting on list  
 Comparator*<CollegeStudent> comparator = *Comparator*.*comparing*(CollegeStudent::getName);  
  
 *List*<CollegeStudent> collect1 = CollegeStudent.*getCollegeStudents*().collect(Collectors.*toList*());  
 collect1.sort(comparator);  
 System.***out***.println("sorted using list.sort-->" + collect1);  
}

Parallel streams

Parallel stream will divide the data into multiple chunks and it will assign each chunk to one processor,

if we have a octa core processor then entire data would be divided into 8 data sets and each data set will be given to 1 processor and all processors will work parallelly



In reality check the parallel stream performance before u choose parallel stream

Sometime when ur using parallel streams in that if multiple threads are trying to update a non-synchronized variable( like integer instead of AtomicInteger) then wrong output will come at that time either don’t use parallel streams or better use synchronized variables like AtomicInteger instead of integer

For example in below code, for addition examples if multiple threads updates same variables for sum if the variables in not synchronised it will get dirty updates

Either we should use synchronised variables like atomic integer or better to use single stream which internally will use 1 thread.

public class Adder {  
 int **sum**;  
 AtomicInteger **sumAtomic**;  
  
 public Adder() {  
 }  
  
 public Adder(AtomicInteger *sumAtomic*) {  
 this.**sumAtomic** = *sumAtomic*;  
 }  
  
 public void update(Integer *e*){  
 **sum**+=*e*.intValue();  
 }  
 public void updateAtomically(Integer *e*){  
 **sumAtomic**.addAndGet(*e*);  
 }  
  
 public static void main(String[] *args*) {  
 Adder adder=new Adder();  
 Adder adderAtomic=new Adder(new AtomicInteger());

*// this will give correct output as we are using 1 thread* *IntStream*.*rangeClosed*(1,1000).forEach(adder::update);

*//wrong o/p because here multiple threads are using updating same variable  
 IntStream*.*rangeClosed*(1,1000).parallel().forEach(adder::update);  
  
 *//Here correct o/p because multiple threads are updating a synchronized variable  
 IntStream*.*rangeClosed*(1,1000).parallel().forEach(adderAtomic::updateAtomically);  
 System.***out***.println("sum is "+adder.**sum**);  
 System.***out***.println("sumed using atomic integer is "+adderAtomic.**sumAtomic**.intValue());  
 }  
}

Optional

Methods

|  |  |
| --- | --- |
| // 2 ways to create optional object  Optional.ofNullable(null) // this wont throw null pointer excep  Optional.of(null) // this will throw null pointer excep, so be cautious while using this method | U can enclose any method call in this & this wont throw any exception |
| Optional.empty() | To create an empty optional object with null value |
| Boolean flag= Optional.isPresent() | To check if value is there or not |
| void Optional.ifPresent(Consumer c)  It is like🡪 if element Is present then, take this element & do this | Optional.*ofNullable*(2).ifPresent((*e*)-> System.***out***.println(*e*+" is consumed"));  //2 is consumed |
| Optional.get() | To get the value |
| Object Optional.*empty*().orElse(Object) | The diff between orElse() & orElseGet() is supplier but both will be executed when optional value is missing  System.***out***.println(Optional.*empty*().orElse("default")); *//default* |
| Object Optional.*empty*().orElseGet(Supplier ) | System.***out***.println(Optional.*empty*().orElseGet(()-> "default value ")); *//default value* |
| Optional.*empty*().orElseThrow( Supplier ) | System.***out***.println(Optional.*empty*().orElseThrow(()-> new RuntimeException("no value hence exception is throw")));  // this will throw exception |
| Boolean flag=Optional.empty().isPresent() // to check if value is present or not |  |
|  |  |
| Optional map(Function f) | Optional.*ofNullable*("2")  .map(*e*->"orayya").ifPresent((*c*)-> System.***out***.println("value is there ")); |

Default & static methods

Default method is a concrete method allowed to define in interface which will be default available to all child classes who are implementing this interface

So that how many no of methods we declare in interface, all those 10,000 childs who are implementing this interface doesn’t need to implement this method as

It is already concrete and default method, so to provide the backward compatibility only they introduced this default methods so that even if new methods added in parent also child will not get any compilation error

Default methods are those concrete methods which will be inherited and CAN be overridden whereas

Static methods are those concrete methods which will be inherited and CANNOT be overridden whereas

Note: if ur class is implementing 2 interfaces which have same method then both methods will be inherited into child then error will come as if client calls child method which super interface method should be called? So to solve this we have to override that method in child class

|  |  |  |  |
| --- | --- | --- | --- |
| interface *A*{  public default String m1(){  return "P1";  } } | interface *B*{  public default String m1(){  return "P1";  } } | //This leads to error as per above case  class C implements *A*,*B* {  } | class C implements *A*,*B* {  @Override  public String m1(){  return "P1";  } }  since method is overridden, then if child class method is called only that overridden method will be called |

# Date and Time API in java8

If u want to create any java object related to DateTimeAPI use of() method

Duration hours\_3 = Duration.*ofHours*(3); *sop*("duration is "+hours\_3.toMinutes());

Create date:- LocalDate.of()

LocalDate.now()

New series of date and time APIs are created in the new java.time package.

Following are some of the important classes introduced in java.time package –

LocalDate is immutable same like string

LocalTime wont store the offset (+6 or +5:30) whereas ZonedDateTime will store all the offset and zone related information

|  |  |
| --- | --- |
| To calculate the difference between 2 LocalDates use | //This approach is useful to cal the diff between 2 dates, 2 locatime,2 localdatetimes api  long days2 = *ChronoUnit*.***DAYS***.between( wifeBday,myBday); -870 *sop*("Days b/n 2 dates is"+days2); |
| To calculate the difference between 2 LocalDates, 2 LocalDateTime, 2 Instances use  This is not compatible for LocalDate API  If u pass LocalDate u will get exceptions | //Diff b/n 2 instances  Instant startInstance = Instant.*now*();  Instant endInstance = Instant.*now*();  System.***out***.println("Seconds elapsed -->"+Duration.*between*(startInstance,endInstance).toMillis());  //Diff b/n 2 localTimes  LocalTime start = LocalTime.*now*();  LocalTime end = LocalTime.*now*();  System.***out***.println("Seconds elapsed -->"+Duration.*between*(start,end).toMillis());  // diff b/n 2 localDateTimes  LocalDateTime myBdayWithTime=LocalDateTime.*of*(1994,06,12,9,36,00); LocalDateTime myWifeBdayWithTime=LocalDateTime.*of*(1996,10,29,21,36,00);  Duration between = Duration.*between*(myWifeBdayWithTime, myBdayWithTime); *sop*("duration between 2 bdays"+ between.toHours());  Above can be done using Chronounit.hours.between … |
|  |  |

|  |  |
| --- | --- |
| TemporalField(interface) ,TemporalUnit  |  |  ChronoField,ChronoUnit | TemporalAmount(interface)  |  |  ChronoPeriod |
| Create a local date  LocalDate.now(),LocalTime.now();  LocalDate.of()  LocalTime.of(hour,min,sec) | LocalTime now = LocalTime.*of*(15, 50, 01); |
| Get details from the date or from time | sop(today.**get**(ChronoField.**DAY\_OF\_MONTH**));//  *sop*("hour is "+now.get(*ChronoField*.***HOUR\_OF\_DAY***)); *sop*("minute of hour is -->"+now.getMinute()); *sop*("minute of day is -->"+now.get(*ChronoField*.***MINUTE\_OF\_DAY***) ); |
| Modify or add or decrease the date | *sop*("add 2 day from today"+today.plus(2, *ChronoUnit*.***DAYS***));  *sop*("Add 2 days from today -->"+today.plusDays(2)); *sop*("minus 2 days from today-->"+today.minusDays(2));  ChronoUnit.DAYS.between(date1, date2); |
| Change the current date/time with new date/new time | *//Date manipulations*  *sop*("today with 12 month-->"+today.withMonth(12)); *sop*("today with 11th month-->"+today.with(*ChronoField*.***MONTH\_OF\_YEAR***,11)); *sop*("today with 22nd day of month-->"+today.with(*ChronoField*.***DAY\_OF\_MONTH***,22));  *sop*(today.**with**(TemporalAdjusters.*lastDayOfYear*()));  //Time manipulations  *sop*("current time with eve 8pm "+now.with(*ChronoField*.***HOUR\_OF\_DAY***,20)); *sop*("Current time with last minute"+now.with(*ChronoField*.***SECOND\_OF\_DAY***,0));*//00:00:00.775* |
| Comparing 2 dates to check if it is equal or before or after | LocalDate today = LocalDate.*now*();  LocalDate tomm = today.plus(1, *ChronoUnit*.***DAYS***);  *sop*("is Equal"+today.isEqual(tomm)); *sop*("is today before tomm -->"+today.isBefore(tomm)); *sop*("is today after tomm -->"+today.isAfter(tomm)); |
| Calculating the days/Period between 2 dates   1. Period id a worst api instead prefer ChronoUnit.\*.between() methods to calculate the diff between dates,   If u want the diff in hours,minutes,seconds,days,months what ever u can use this api | Period is nothing but some certain time, it doesn’t hold the date  Period class is compatible only with LocalDate  Duration class is compatible with LocalTime,LocalDateTime  LocalDate myBday = LocalDate.*of*(1994,06,12); LocalDate wifeBday = LocalDate.*of*(1996, 10, 29);  Best option is use chronoUnit it will give u the exact days by considering the year 1) long days2 = *ChronoUnit*.***DAYS***.between( wifeBday,myBday); -870 *sop*("Days b/n 2 dates is"+days2);  //This period is worst it will just calc the diff b/n the dates it wont consider the year and month   1. int days = Period.*between*( wifeBday,myBday).getDays(); *sop*("Days b/n 2 dates is"+days);*// 17*   LocalDateTime myBdayWithTime=LocalDateTime.*of*(1994,06,12,9,36,00); LocalDateTime myWifeBdayWithTime=LocalDateTime.*of*(1996,10,29,21,36,00);  long hours = *ChronoUnit*.***HOURS***.between(myBdayWithTime, myWifeBdayWithTime); *//20892*  *sop*("seconds between 2 bdays is "+*ChronoUnit*.***SECONDS***.between(myBdayWithTime,myWifeBdayWithTime)); |
| Calculating the difference between 2 times  LocalTime or 2 LocalDateTimes, 2 Instances  So use Duration API for this  This is not compatible for LocalDate API  If u pass LocalDate u will get exceptions | LocalDateTime myBdayWithTime=LocalDateTime.*of*(1994,06,12,9,36,00); LocalDateTime myWifeBdayWithTime=LocalDateTime.*of*(1996,10,29,21,36,00); // this operation can also be achieved by ChronoUnit.HOURS.between 1)Duration between = Duration.*between*(myWifeBdayWithTime, myBdayWithTime); *sop*("duration between 2 bdays"+ between.toHours()); //-20892  //To create any object in DateTimeAPI use of() method  Duration hours\_3 = Duration.*ofHours*(3); *sop*("duration is "+hours\_3.toMinutes());  Calculating the seconds or time elapsed using LocalTime, same can be done using instance  LocalTime start = LocalTime.*now*();  LocalTime end = LocalTime.*now*(); System.***out***.println("Seconds elapsed -->"+Duration.*between*(start,end).toMillis()  //Unsupported operation exception  Duration.*between*(LocalDate.*now*(),LocalDate.*now*().plusDays(2)); |
| Using Instance class | Instant startInstance = Instant.*now*();  Instant endInstance = Instant.*now*(); System.***out***.println("Seconds elapsed -->"+Duration.*between*(startInstance,endInstance).toMillis()); |
| LocalDate🡪LocalDateTime | LocalDateTime ldt = LocalDate.atTime(LocalTime t)  LocalDateTime ldt = LocalDate.atTime(OffSetTime t)  LocalDateTime ldt = LocalDate.atTime(int hour, int min, int sec)  LocalDateTime localDateTime = LocalDate.*now*().atTime(LocalTime.*now*()); |
| LocalDateTime🡪 LocalDate | *sop*("localDateTime --> localDate"+localDateTime.**toLocalDate**()); |
| LocalTime🡪LocalDateTime | LocalDateTime ldt = LocalTime.*now*().atDate(LocalDate.*of*(1996, 10, 29)); |
| LocalDateTime🡪 LocalTime | *sop*("ldt ->localTime "+ldt.toLocalTime()); |
| 1. Getting U.S.A time using **ZoneDateTime** | *//getting us time using ZonedDateTime*  *ZoneId* usZone=*ZoneId*.*of*(*ZoneId*.***SHORT\_IDS***.get("CST")); ZonedDateTime nowInChicago=ZonedDateTime.*now*(usZone); System.***out***.println("CST time now is "+nowInChicago);  *ZoneId* ESTZone=*ZoneId*.*of*(*ZoneId*.***SHORT\_IDS***.get("EST")); *sop*("time in EST is "+ZonedDateTime.*now*(ESTZone));  answer  *CST time now is 2024-04-08T02:54:37.927-05:00[America/Chicago]*  *time in EST is 2024-04-08T02:54:38.072-05:00* |
| 1. Getting u.s.a time using **localDateTime**   LocalDateTime.now(ZoneId z)  //The problem with localDateTime is we don’t have zone info at all | *//Getting u.s time using localDateTime*  *sop*("india time now -->"+LocalDateTime.*now*(*ZoneId*.*of*("Asia/Kolkata"))); *sop*("America/Los\_Angeles time now -->"+LocalDateTime.*now*(*ZoneId*.*of*("America/Los\_Angeles")));  *response*  *india time now -->2024-04-08T13:24:38.073*  *America/Los\_Angeles time now -->2024-04-08T00:54:38.074* |
| 1. Get u.s.a or paris time using **Instant**   The only diff is u will not get the timezone info using localDateTime | *//Getting paris time using instant* ZonedDateTime getParisTimeUsingInstant = Instant.*now*().atZone(*ZoneId*.*of*("Europe/Paris")); System.***out***.println("getParisTimeUsingInstant"+getParisTimeUsingInstant);  // getParisTimeUsingInstant2024-04-08T10:20:44.797+02:00[Europe/Paris] |
| Convert LocalDateTime to ZonedDateTime by just adding the zone info we are not changing the time | LocalDateTime currentTime=LocalDateTime.*now*(); *//This will just add the zone info to the provided time* ZonedDateTime americaLosZonedDateTime = currentTime.atZone(*ZoneId*.*of*("America/Los\_Angeles")); System.***out***.println("americaLosZonedDateTime"+americaLosZonedDateTime);  LocalDateTime australiaTime= LocalDateTime.*now*(*ZoneId*.*of*("Australia/Sydney")); *sop*("now time in Australia/Sydney is "+australiaTime);  //now time in Australia/Sydney is 2024-04-08**T18:35:27.060** ZonedDateTime addedAfricaTimeZone = australiaTime.atZone(*ZoneId*.*of*("Africa/Cairo")); *sop*("just added the zone to australia time"+addedAfricaTimeZone);  just added the zone to australia time2024-04-08**T18:35:27.060+02:00[Africa/Cairo]**  **//Here if u see for Australia time we just added Africa zone info** |

OffsetDateTime just adds the offset, like -6:00 or -5:00 it wont add the zone info like “Asia/Calcutta”

If u want both then use zonedDateTime

**Local** − Simplified date-time API with no complexity of timezone handling.

**Zoned** − Specialized date-time API to deal with various timezones

<https://docs.oracle.com/javase/8/docs/api/java/time/package-summary.html>

As such, a clock can be used instead of [System.currentTimeMillis()](https://docs.oracle.com/javase/8/docs/api/java/lang/System.html#currentTimeMillis--) and [TimeZone.getDefault()](https://docs.oracle.com/javase/8/docs/api/java/util/TimeZone.html#getDefault--).

long millis = Clock.*systemUTC*().millis();

LocalDate l=LocalDate.*of*(2011, Month.*FEBRUARY*,29);

LocalTime lt=LocalTime.*of*(12,60,60,200)

## How to get any country time as below

LocalTime lt=LocalTime.now(ZoneId.of("GMT"));

### Old date time formatter

===========================

import java.text.SimpleDateFormat;

private static final DateFormat sdf = new SimpleDateFormat("yyyy/MM/dd HH:mm:ss");

public static void main(String[] args) {

Date date = new Date();

System.out.println(sdf.format(date));

Calendar cal = Calendar.getInstance();

System.out.println(sdf.format(cal.getTime()));

## Date Formatting

Parse()- from string to LocalDate conversion –parse(dateAsString, DateTimeFormatter) //Here se in parse tells string to date conversion

Format() from LocalDate to String conversion

While parsing from string we have to tell the exact format of string whether it is ISO local time or ISO zoned time? Or is it just time? Or just date?

Whatever u have to tell the exact format of string what u are passing

### **//from LocalDateTime to string conversion –format()**

LocalDate date = LocalDate.now();

String text = date.format(formatter);

LocalDate parsedDate = LocalDate.parse(text, formatter);

|  |  |
| --- | --- |
| [ISO\_LOCAL\_DATE](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#ISO_LOCAL_DATE) | '2011-12-03' |
| [ISO\_LOCAL\_DATE\_TIME](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#ISO_LOCAL_DATE_TIME) | '2011-12-03T10:15:30' |
| [ISO\_ZONED\_DATE\_TIME](https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html#ISO_ZONED_DATE_TIME) | '2011-12-03T10:15:30+01:00[Europe/Paris]' |
|  |  |

LocalDateTime now = LocalDateTime.now();

private static final DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss");

System.out.println(dtf.format(now));

LocalDate localDate = LocalDate.now();

System.out.println(DateTimeFormatter.ofPattern("yyy/MM/dd").format(localDate));

### **//String to LocalDate conversion –parse()**

String now = "2017-06-13 12:30";

DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm");

LocalDateTime formatDateTime = LocalDateTime.parse(now, formatter);

System.out.println("Before : " + now);

System.out.println("After : " + formatDateTime);

System.out.println("After : " + formatDateTime.format(formatter));

}

}

LocalDate parsedDate = LocalDate.*parse*("2024-04-08", DateTimeFormatter.***ISO\_DATE***);  
*sop*("parsedDate is "+parsedDate);  
String formattedDate = parsedDate.format(DateTimeFormatter.*ofPattern*("dd==MM==yyyy"));  
*sop*("formatted as String -->"+formattedDate);

parsedDate is 2024-04-08

formatted as String -->08==04==2024

private static void parseAndFormatLocalDateAndTime() {  
 ZonedDateTime canadaZDT=ZonedDateTime.*now*(*ZoneId*.*of*("Canada/Atlantic"));  
 *sop*(canadaZDT);  
 String canadaTimeAsString = canadaZDT.format(DateTimeFormatter.***ISO\_ZONED\_DATE\_TIME***);  
 *sop*("formatted from date to str -->"+canadaTimeAsString);  
 *//Here we have to tell the format of the string which u are passing* LocalDateTime parsedFromString = LocalDateTime.*parse*(canadaTimeAsString, DateTimeFormatter.***ISO\_ZONED\_DATE\_TIME***);  
 *sop*("parsedFromString to Date--> "+parsedFromString);  
}  
  
private static void parseAndFormatLocalTime() {  
 LocalTime canadaCurrentTime=LocalTime.*now*(*ZoneId*.*of*("Asia/Calcutta"));  
 *sop*(canadaCurrentTime);  
 DateTimeFormatter formatter=DateTimeFormatter.*ofPattern*("HH@@mm@@ss");  
 String timeAsString = canadaCurrentTime.format(formatter);  
 *sop*(timeAsString);  
 LocalTime parsedDateFromString = LocalTime.*parse*(timeAsString, formatter);  
 *sop*("parsedDateFromString"+parsedDateFromString);  
}  
  
private static void parseAndFormatLocalDate() {  
 LocalDate parsedDate = LocalDate.*parse*("2024-04-08", DateTimeFormatter.***ISO\_DATE***);  
 *sop*("parsedDate is "+parsedDate);  
 String formattedDate = parsedDate.format(DateTimeFormatter.*ofPattern*("dd==MM==yyyy"));  
 *sop*("formatted as String -->"+formattedDate);  
}

### **Get Instant from Clock**

Instant i=Clock.*systemUTC*().instant();

long millis = Clock.systemUTC().millis();

### **From instant to localDateTime**

Instant instant = Instant.now();

System.out.println("Instant : " + instant);

LocalDateTime ldt = LocalDateTime.ofInstant(instant, ZoneOffset.UTC);

LocalDateTime localDateTime = instant.atZone(defaultZoneId).toLocalDateTime();

ZonedDateTime zonedDateTime = instant.atZone(defaultZoneId);

### **From Local Date time to Instant**

Instant instant = date.toInstant();

### **Convert Date to LocalDate**

import java.time.\*;

import java.util.Date;

public class DateToJavaTime {

public static void main(String[] args) {

Date date = new Date();

//Convert to LocalDate

LocalDate localDate = date.toInstant().atZone(ZoneId.systemDefault()).toLocalDate();

System.out.println(localDate.toString());

//Convert to LocalDateTime

LocalDateTime localDateTime = date.toInstant().atZone(ZoneId.systemDefault()).toLocalDateTime();

System.out.println(localDateTime);

}

### **Temporal Adjusters**

LocalDate with1 = localDate.with(TemporalAdjusters.lastDayOfMonth());

LocalDate with2 = localDate.with(TemporalAdjusters.next(DayOfWeek.MONDAY));

### **ZonedDateTime**

withZoneSameInstant() method

The **java.time.ZonedDateTime.withZoneSameInstant**(ZoneId zone) method returns a copy of the date-time with a different time-zone, retaining the instant.

Following is the declaration for **java.time.ZonedDateTime.withZoneSameInstant**(ZoneId zone) method.

**public ZonedDateTime withZoneSameInstant**(ZoneId zone)

The following example program shows the usage of **java.time.ZonedDateTime.withZoneSameInstant**(ZoneId zone) method.

import java.time.ZoneId;

import java.time.ZonedDateTime;

public class ZoneMethod {

public static void main(String[] args) {

ZonedDateTime date = ZonedDateTime.parse("2020-08-28T19:10:38.492+05:30[Asia/Kolkata]");

ZonedDateTime result = date.withZoneSameInstant(ZoneId.of("Asia/Tokyo"));

System.out.println(result);

}

}

The output is

The same instant fot the given zone Asia/Tokyo is : 2020-08-28T22:40:38.492+09:00[Asia/Tokyo]

Just add zone to the current Date and time

ZoneId zoneId=ZoneId.of("Asia/Singapore");  
ZonedDateTime zdt=ZonedDateTime.of(LocalDateTime.now(),zoneId);  
System.out.println(zdt);

### **Add few days/months to localDateTime**

LocalDate ldt = LocalDate.parse(stringInputDate, dtf);  
ldt= ldt.plus(days, ChronoUnit.DAYS)  
 .plus(months, ChronoUnit.MONTHS)  
.plus(years,ChronoUnit.YEARS);

Copying the date

LocalDate date = LocalDate.of(year,month,1);

LocalDate firstOfWeek = LocalDate.from(date);

### Difference between 2 dates- 2 seonds

LocalDate date1 = LocalDate.now();

LocalDate date2 = date1.plus(1, ChronoUnit.MONTHS);

Period period = Period.between(date2, date1);

long a = Clock.systemUTC().millis();  
Thread.sleep(3000);  
long a1 = Clock.systemUTC().millis();  
System.out.println(a1-a);

## Code samples

### Get First Sunday of any year august

**import** **static** java.time.temporal.TemporalAdjusters.firstInMonth;

**public** **class** FirstSundayOfTheYear {

public static String friendShipDay(int inputYear) {  
 DateTimeFormatter dtf = DateTimeFormatter.*ofPattern*("dd-MM-yyyy");

LocalDate august = LocalDate.*of*(inputYear, Month.*AUGUST*, 1);  
 LocalDate with = august.with(TemporalAdjusters.*firstInMonth*(DayOfWeek.*SUNDAY*));  
  
 return with.format(dtf);  
}  
public static void main(String[] args) {  
 System.*out*.println(*friendShipDay*(2022));  
}

### **Now whats time in US, UK,Australia**

Given a certain in india may be like – 12 june-1994 – 9:36 AM and what that’s time

Adi India time, a second ki US lo time entha., and UK lo time entha

*//Build DATE and time with asia time zone.*DateTimeFormatter dtf=DateTimeFormatter.*ofPattern*("yyyy-MM-dd HH:mm:ss");  
LocalDateTime parse = LocalDateTime.*parse*("2020-08-27 10:40:50",dtf);  
ZonedDateTime zonedDT= parse.atZone(ZoneId.*of*("Asia/Calcutta"));  
  
*//aa samayam lo america lo time entha / ade samayanaki london lo time entha*

*// santu puttindi 29 october 1996 7:00 pm IST apudu US lo time entha*ZonedDateTime zonedDateTime = zonedDT.withZoneSameInstant(ZoneId.*of*("Canada/Central"));  
System.*out*.println(zonedDateTime);

API

## TemporalAdjuster

**All Known Implementing Classes:**

[DayOfWeek](https://docs.oracle.com/javase/8/docs/api/java/time/DayOfWeek.html)[Instant](https://docs.oracle.com/javase/8/docs/api/java/time/Instant.html),  [LocalDate](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html), [LocalDateTime](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDateTime.html), [LocalTime](https://docs.oracle.com/javase/8/docs/api/java/time/LocalTime.html) , [Month](https://docs.oracle.com/javase/8/docs/api/java/time/Month.html), [MonthDay](https://docs.oracle.com/javase/8/docs/api/java/time/MonthDay.html), [OffsetDateTime](https://docs.oracle.com/javase/8/docs/api/java/time/OffsetDateTime.html), [OffsetTime](https://docs.oracle.com/javase/8/docs/api/java/time/OffsetTime.html),  [Year](https://docs.oracle.com/javase/8/docs/api/java/time/Year.html), [YearMonth](https://docs.oracle.com/javase/8/docs/api/java/time/YearMonth.html), [ZoneOffset](https://docs.oracle.com/javase/8/docs/api/java/time/ZoneOffset.html)

# Repeatable annotations

public class App {

@Repeatable(value = Cars.class )

public @interface Manufacturer {

String value();

};

/cars will just have manufacturer array

@Retention( RetentionPolicy.RUNTIME )

public @interface Cars {

Manufacturer[] value() default{};

}

@Manufacturer( "Mercedes Benz")

@Manufacturer( "Toyota")

@Manufacturer( "BMW")

@Manufacturer( "Range Rover")

public interface Car {

}

semaphores