**Stream API**

**Streams:**

Streams are used to process the elements of the Collection Object.

-> Stream object we get by calling stream() on any Collection Object

Ex: Stream s=c.stream()

**filter() :**

filter(Predicate) method is used to filter the elements of an collection object based on the condition.

-> Predicate(Functional Interface) return type is boolean

**Ex 1: Filtering Collection Object Elements Based on Even Number Condition**

ArrayList<Integer> al = **new** ArrayList<Integer>();

**for** (**int** i = 1; i < 11; i++) {

al.add(i);

}

System.***out***.println("Before Filter,List is ::"+al);

List<Integer> filteredAl=al.stream().filter(ele->ele%2 ==0 ).collect(Collectors.*toList*());

System.***out***.println("Before Filter,List is ::"+filteredAl);

collect(Collectors.*toList*());

----> It is to get the result as List<T> Object

-----------------------------------------------------

**Ex2 : Filtering Employees of Collection Object Based on Employee Property Id**

ArrayList<Employee> emps = **new** ArrayList<Employee>();

**for** (**int** i = 1; i < 11; i++) {

emps.add(**new** Employee(i, "Pavan"));

}

System.***out***.println("Before Filter,List is ::" + emps);

List<Employee> filteredEmps = emps.stream().filter(ele -> ele.getId() % 2 == 0).collect(Collectors.*toList*());

System.***out***.println("Before Filter,List is ::" + filteredEmps);

**map(Function):**

map(Function) is used to manipulate the every element inside collection object;

-> Function(Functional Interface) return type is generic of collection object.

**Ex 1: Adding 10 to every element inside collection object**

ArrayList<Integer> alMap = **new** ArrayList<Integer>();

**for** (**int** i = 1; i < 11; i++) {

alMap.add(i);

}

System.***out***.println("Before map(),List is ::" + alMap);

List<Integer> mappedAl = alMap.stream().map(ele -> ele+10).collect(Collectors.*toList*());

System.***out***.println("After map(),List is ::" + mappedAl);

-----------------------------------------------------

**Ex2.Adding 100 to id property of every Employee element inside collection object**

ArrayList<Employee> empList = **new** ArrayList<Employee>();

**for** (**int** i = 1; i < 11; i++) {

empList.add(**new** Employee(i, "Pavan"));

}

System.***out***.println("Before map(),List is ::" + empList);

List<Employee> updatedempList = empList.stream().map(emp -> {

emp.setId(emp.getId()+100);

**return** emp;

}

).collect(Collectors.*toList*());

System.***out***.println("After map(),List is ::" + updatedempList);

**count():**

Count() is used to get the count of the elements inside collection object.

**Ex:**

Long count = countMap.stream().count();

**sorted():**

sorted() is used to sorting the elements of collection object.

->Natural Sorting order is ascending.

**Ex1:Natural Sorting Order**

ArrayList<Integer> list = **new** ArrayList<Integer>();

**for** (**int** i = 10; i > 0; i--) {

list.add(i);

}

System.***out***.println("Before sorted(),List is ::" + list);

List<Integer> sortedList= list.stream().sorted().collect(Collectors.*toList*());

System.***out***.println("After sorted(),List is ::" + sortedList);

**Natural Sorting Order With Comparable**

List<Integer> sortedList= list.stream().sorted((a,b)->a.compareTo(b)).collect(Collectors.*toList*());

-----------------------------------------------------

**Ex2:User defined Sorting Order**

**sorted(Comparator or Comparable):**

**->** we need to pass Comparator obj as parameter to sorted method with user defined sorting (descending) logic , Since Comparator is functional interface we can use lambda expression for implementation(compare).

ArrayList<Integer> alList = **new** ArrayList<Integer>();

**for** (**int** i = 1; i < 10; i++) {

alList.add(i);

}

System.***out***.println("Before sorted(Comparator),List is ::" + alList);

List<Integer> sortedCompList= alList.stream().sorted((a,b)->(a<b)?1:(a>b)?-1:0).collect(Collectors.*toList*());

System.***out***.println("After sorted(Comparator),List is ::" + sortedCompList);

**min(Comparator or Comparable):**

min() is used to find the first element of the collection object but to use min() first we need to sort the collection object.

**Ex:**

ArrayList<Employee> empsList = **new** ArrayList<Employee>();

**for** (**int** i = 1; i < 11; i++) {

empsList.add(**new** Employee(i,"Pavan"));

}

Employee minEmp= empsList.stream().min((a,b)->(a.getId()<b.getId())?1:(a.getId()>b.getId())?-1:0).get();

System.***out***.println("MIn Employee Is ::" + minEmp);

**max(Comparator or Comparable):**

max() is used to find the last element of the collection object but to use max() first we need to sort the collection object.

**Ex:**

ArrayList<Employee> empsList = **new** ArrayList<Employee>();

**for** (**int** i = 1; i < 11; i++) {

empsList.add(**new** Employee(i,"Pavan"));

}

Employee maxEmp= empsList.stream().max((a,b)->a.getId().compareTo(b.getId())).get();

System.***out***.println("Max Employee Is ::" + maxEmp);

**forEach(Consumer):**

forEach(Consumer) is used to process the each element of collection object.

**Ex:**

ArrayList<Employee> forEachList = **new** ArrayList<Employee>();

**for** (**int** i = 1; i < 11; i++) {

forEachList.add(**new** Employee(i,"Pavan"));

}

forEachList.stream().forEach(emp->emp.setName("PJ"));

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

**toArray():**

It is used to convert stream of objects into array;

**Ex:**

ArrayList<Employee> forEachList = **new** ArrayList<Employee>();

**for** (**int** i = 1; i < 11; i++) {

forEachList.add(**new** Employee(i,"Pavan"));

}

Employee[] empArray=forEachList.stream().toArray(Employee[]::**new**);

Employee[]::**new**

-->It is to return Employee[] object.

System.***out***.println(Arrays.*deepToString*(empArray));

**Stream.of():**

Wherever group of elements are there we can use stream object.

**Ex:**

Stream<Integer> s=Stream.*of*(1,11,111,1111);

System.***out***.println(Arrays.*deepToString*(s.toArray(Integer[]::**new** )));