Introduction to streams.

A Collection is a group of objects as a single entity. If we want to process objects from collection (bulk operations) then we can go for streams.

Java.io.streams are useful for reading and writing data to files (file operations).

Java.util.streams and related collection operations

Streams reduces length of code

Streams came in 1.8 version.

Stream interface is present in java.util.stream

**import** java.util.stream.Stream;

stream method is present in Collection interface.

==

Stream s =collection.stream();

Here collection can be anything like ArrayList, Vector….

==

Once we create Stream interface, we can process collection objects in 2 steps.

1. Configuration
2. Processing

Configuration

1. Filter mechanism (filer elements based on some Boolean condition)
2. Map mechanism (if you want to created a mapped object then use map)

FILTER and MAP

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.stream.Collectors;

**public** **class** ExampleOneFilterMap {

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

List<Integer> intList = *getIntList*();

List<Integer> evenNumberList = **null**;

List<Integer> doubleNumberList = **null**;

/\* before 1.8 : to get even numbers \*/

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

**for** (Integer i : intList) {

**if** (i % 2 == 0) {

evenNumberList.add(i);

}

}

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

**for** (Integer i : intList) {

doubleNumberList.add(i \* 2);

}

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

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

/\* after 1.8 : to get even numbers \*/

evenNumberList = intList.stream().filter(i -> i % 2 == 0).collect(Collectors.*toList*());

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

doubleNumberList = intList.stream().map(i->i\*2).collect(Collectors.*toList*());

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

}

**public** **static** List<Integer> getIntList() {

List<Integer> intList = **new** ArrayList<Integer>();

intList.add(0);

intList.add(5);

intList.add(10);

intList.add(15);

intList.add(20);

intList.add(22);

intList.add(28);

intList.add(33);

**return** intList;

}

}

Filters: if we want to filter elements from the collection based on some Boolean condition, then we should go for filtering

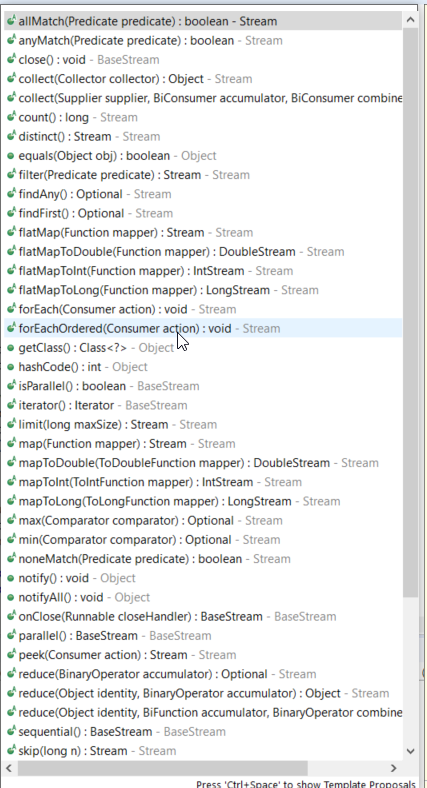
We can configure Filter by using filter () method of Stream interface

**public** **interface** Stream<T> **extends** BaseStream<T, Stream<T>> {

Stream<T> filter(Predicate<? **super** T> predicate);

==

evenNumberList = intList.stream().filter(i -> i % 2 == 0).collect(Collectors.*toList*());



Map

If you want to create a separate new object for every object present in collection based on some function then we should go for mapping mechanism

We can implement mapping my sing map() method of Stream interface

**public** **interface** Stream<T> **extends** BaseStream<T, Stream<T>> {

<R> Stream<R> map(Function<? **super** T, ? **extends** R> mapper);

==

doubleNumberList = intList.stream().map(i->i\*2).collect(Collectors.*toList*());

== Various methods of Stream==

1. Processing by collect method => collects the elements from the stream and adds to specified collection

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.stream.Collectors;

**public** **class** ExampleTwoFilterMap {

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

List<String> namesList = *getIntList*();

List<String> mamesFilteredList = namesList.stream().filter(name -> name != **null** && name.length() > 3)

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

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

List<String> upperCaseNamesList = namesList.stream().map(name -> name.toUpperCase())

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

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

**long** noOfElements = namesList.stream().filter(name -> name != **null** && name.length() > 3).count();

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

namesList

.stream()

.filter(s -> s.startsWith("S"))

.map(String::toUpperCase)

.sorted()

.forEach(System.***out***::println);

}

**public** **static** List<String> getIntList() {

List<String> namesList = **new** ArrayList<String>();

namesList.add("Steve");

namesList.add("Tim");

namesList.add("Lucy");

namesList.add("Pat");

namesList.add("Angela");

namesList.add("Tom");

**return** namesList;

}

}

Processing by sorted method

We can use sorted method to sort elements inside Stream

We can sort either on default natural sorting order or based on our won customized sorting order

1. sorted() =>For default natural sorting order
2. sorted(Comparator c) => For customized sorting order

default sort

namesList

.stream()

.filter(name -> name != **null** && name.length() > 3)

.map(String::toUpperCase)

.sorted()

.forEach(System.***out***::println);

Comparator sort

namesList

.stream()

.filter(name -> name != **null** && name.length() > 3)

.map(String::toUpperCase)

// .sorted((name1,name2)->name1.compareTo(name2)) asc

// .sorted((name1,name2)->name2.compareTo(name1)) desc

.sorted((name1,name2)->-name1.compareTo(name2)) // desc (see minus - sybmol)

.forEach(System.***out***::println);

Processing by min and max methods

There list should be sorted then only we can get min or max

1. min(comparator c) -> Returns minimum value according to specified comparator
2. max((comparator c) -> Returns maximum value according to specified comparator

System.***out***.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*min max \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

**int** minimum=*getIntList*()

.stream()

.min((val1,val2)->val1.compareTo(val2))

.get();

System.***out***.println("minimum "+minimum);

**int** maximum=*getIntList*()

.stream()

.max((val1,val2)->val1.compareTo(val2))

.get();

System.***out***.println("maximum "+maximum);

**public** **static** List<Integer> getIntList() {

List<Integer> intList = **new** ArrayList<Integer>();

intList.add(0);

intList.add(5);

intList.add(10);

intList.add(15);

intList.add(20);

intList.add(22);

intList.add(28);

intList.add(33);

**return** intList;

}

==== For each

namesList

.stream()

.filter(name -> name != **null** && name.length() > 3)

.map(String::toUpperCase)

.sorted()

.forEach(s1->System.***out***.println(s1));

Or below with method reference

namesList

.stream()

.filter(name -> name != **null** && name.length() > 3)

.map(String::toUpperCase)

.sorted()

.forEach(System.***out***::println);

==Processing by toArray() method==

System.***out***.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*toArray \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

Integer[] intArr =*getIntList*()

.stream()

.toArray(Integer[]::**new**);

System.***out***.println(intArr[2]);

We can also apply Stream for group of values & for arrays (not just for collections)

1. For group of values
2. For arrays

System.***out***.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*group of values or arrays\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

Stream<Integer> stream =Stream.*of*(1,2,3,4,5,6);

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

Integer abc[] = {10,11,2,56,99};

Stream<Integer> stream2 =Stream.*of*(abc);

stream2.forEach(System.***out***::println);