Streams

To process objects of the collection, in 1.8 version Streams concept introduced.

What is the differences between Java.util.streams and Java.io streams?

java.util streams meant for processing objects from the collection. Ie, it represents a stream of objects from the collection but Java.io streams meant for processing binary and character data with respect to file. i.e it represents stream of binary data or character data from the file .hence Java.io streams and Java.util streams both are different.

What is the difference between collection and stream?

- If we want to represent a group of individual objects as a single entity then We should go for collection.
- If we want to process a group of objects from the collection then we should go for streams.
- We can create a stream object to the collection by using stream() method of Collection interface. stream() method is a default method added to the Collection in 1.8 version.

```
default Stream stream()
```

Ex: Stream s = c.stream();

- Stream is an interface present in java.util.stream. Once we got the stream, by using that we can process objects of that collection.

- We can process the objects in the following 2 phases

1.Configuration
2.Processing

1) Configuration:

We can configure either by using filter mechanism or by using map mechanism.

Filtering:

We can configure a filter to filter elements from the collection based on some boolean condition by using filter()method of Stream interface.

```
public Stream filter(Predicate<T> t)
```

here (Predicate<T > t) can be a boolean valued function/lambda expression

Ex:

```
Stream s = c.stream();
Stream s1 = s.filter(i -> i%2==0);
```

Hence to filter elements of collection based on some Boolean condition we should go for filter() method.

Mapping:

If we want to create a separate new object, for every object present in the collection based on our requirement then we should go for map() method of Stream interface.

```
public Stream map (Function f);
```

→ It can be lambda expression also

Ex:

```
Stream s = c.stream();
Stream s1 = s.map(i-> i+10);
```

Once we performed configuration we can process objects by using several methods.

2) Processing

processing by collect() method
Processing by count()method
Processing by sorted()method
Processing by min() and max() methods
forEach() method
toArray() method
Stream.of()method

Processing by collect() method

This method collects the elements from the stream and adding to the specified to the collection indicated (specified) by argument.

Ex 1: To collect only even numbers from the array list

Approach-1: Without Streams

```
1)  import java.util.*;
2)  class Test {
3)      public static void main(String[] args) {
4)          ArrayList<Integer> l1 = new ArrayList<Integer>();
5)          for(int i=0; i<=10; i++) {
6)              l1.add(i);
7)          }
8)          System.out.println(l1);
9)          ArrayList<Integer> l2 = new ArrayList<Integer>();
10)         for(Integer i:l1) {
11)             if(i%2 == 0)
12)                 l2.add(i);
13)         }
14)         System.out.println(l2);
15)     }
16) }
```

Approach-2: With Streams

```
1)  import java.util.*;
2)  import java.util.stream.*;
3)  class Test {
4)      public static void main(String[] args) {
5)          ArrayList<Integer> l1 = new ArrayList<Integer>();
6)          for(inti=0; i<=10; i++) {
7)              l1.add(i);
8)          }
9)          System.out.println(l1);
10)         List<Integer> l2 = l1.stream().filter(i -> i%2==0).collect(Collectors.toList());
11)         System.out.println(l2);
12)     }
13) }
```


DURGASOFT, # 202, 2nd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 92 46 21 21 43 | www.durgasoft.com

Ex: Program for map() and collect() Method

```
1)  import java.util.*;
2)  import java.util.stream.*;
3)  class Test {
4)        public static void main(String[] args) {
5)                ArrayList<String> l = new ArrayList<String>();
6)                l.add("rvk"); l.add("rk"); l.add("rkv"); l.add("rvki"); l.add("rvkir");
7)                System.out.println(l);
8)                List<String> l2 = l.Stream().map(s ->s.toUpperCase()).collect(Collectors.toList());
9)                System.out.println(l2);
10)       }
11) }
```

II.Processing by count()method

This method returns number of elements present in the stream.

```
public long count()
```

Ex:

```
long count = l.stream().filter(s ->s.length()==5).count();
sop("The number of 5 length strings is:"+count);
```

III.Processing by sorted()method

If we sort the elements present inside stream then we should go for sorted() method.
the sorting can either default natural sorting order or customized sorting order specified by comparator.
sorted()- default natural sorting order
sorted(Comparator c)-customized sorting order.

Ex:

```
List<String> l3=l.stream().sorted().collect(Collectors.toList());
sop("according to default natural sorting order:"+l3);

List<String> l4=l.stream().sorted((s1,s2) -> -s1.compareTo(s2)).collect(Collectors.toList());
sop("according to customized sorting order:"+l4);
```

IV.Processing by min() and max() methods

min(Comparator c)
returns minimum value according to specified comparator.

max(Comparator c)
returns maximum value according to specified comparator

Ex:

```
String min=l.stream().min((s1,s2) -> s1.compareTo(s2)).get();
sop("minimum value is:"+min);

String max=l.stream().max((s1,s2) -> s1.compareTo(s2)).get();
sop("maximum value is:"+max);
```

V.forEach() method

This method will not return anything.
This method will take lambda expression as argument and apply that lambda expression for each element present in the stream.

Ex:

```
l.stream().forEach(s->sop(s));
l3.stream().forEach(System.out:: println);
```

Ex:

```
1)  import Java.util.*;
2)  import Java.util.stream.*;
3)  class Test1 {
4)          public static void main(String[] args) {
5)                  ArrayList<Integer> l1 = new ArrayaList<Integer>();
6)                  l1.add(0); l1.add(15); l1.add(10); l1.add(5); l1.add(30); l1.add(25); l1.add(20);
7)                  System.out.println(l1);
8)                  ArrayList<Integer> l2=l1.stream().map(i-> i+10).collect(Collectors.toList());
9)                  System.out.println(l2);
10)                 long count = l1.stream().filter(i->i%2==0).count();
11)                 System.out.println(count);
12)                 List<Integer> l3=l1.stream().sorted().collect(Collectors.toList());
13)                 System.out.println(l3);
14)                 Comparator<Integer> comp=(i1,i2)->i1.compareTo(i2);
15)                 List<Integer> l4=l1.stream().sorted(comp).collect(Collectors.toList());
16)                 System.out.println(l4);
17)                 Integer min=l1.stream().min(comp).get();
18)                 System.out.println(min);
19)                 Integer max=l1.stream().max(comp).get();
20)                 System.out.println(max);
```


```
21)                 l3.stream().forEach(i->sop(i));
22)                 l3.stream().forEach(System.out:: println);
23)
24)         }
25) }
```

VI.toArray() method

We can use toArray() method to copy elements present in the stream into specified array

```
Integer[] ir = l1.stream().toArray(Integer[] :: new);
for(Integer i: ir) {
        sop(i);
}
```

VII.Stream.of()method

We can also apply a stream for group of values and for arrays.

Ex:

```
Stream s=Stream.of(99,999,9999,99999);
s.forEach(System.out:: println);

Double[] d={10.0,10.1,10.2,10.3};
Stream s1=Stream.of(d);
s1.forEach(System.out :: println);
```