Stream API Enhancements

Streams concept has been introduced in Java 1.8 version.

The main objective of Streams concept is to process elements of Collection with Functional Programming (Lambda Expressions).

What is the difference 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 are different concepts.

What is the difference between Collections and Streams?

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.

How to Create Stream Object?

We can create 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(); // c is any Collection object**

Note: **Stream is an interface present in java.util.stream package.**

How to process Objects of Collection By using Stream:

Once we got the stream, by using that we can process objects of that collection.
For this we can use either filter() method or map() method.

Processing Objects by using filter() method:

We can use filter() method to filter elements from the collection based on some boolean condition.

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

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

Demo Program:

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

Output:

```
Before Filtering:[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
After Filtering:[0, 2, 4, 6, 8, 10]
```

Processing Objects by using map() method:

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);
```

The argument can be lambda expression also

Demo Program:

```
1)  import java.util.*;
2)  import java.util.stream.*;
3)  public class Test
4)  {
5)     public static void main(String[] args)
6)     {
7)        ArrayList<Integer> l1 = new ArrayList<Integer>();
8)        for(int i =0; i <=10; i++)
9)        {
10)          l1.add(i);
11)       }
12)       System.out.println("Before using map() method:"+l1);
13)       List<Integer> l2=l1.stream().map(i->i*i).collect(Collectors.toList());
14)       System.out.println("After using map() method:"+l2);
15)    }
16) }
```

Output:

```
Before using map() method:[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
After using map() method:[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Processing Objects by using flatMap() method:

Both map and flatMap can be applied to a Stream<T> and they both return a Stream<R>. The difference is that the map operation produces one output value for each input value, whereas the flatMap operation produces an arbitrary number (zero or more) values for each input value.

Typical use is for the mapper function of flatMap to return Stream.empty() if it wants to send zero values, or something like Stream.of(x, y, z) if it wants to return several values.

Demo Program:

```
1)  import java.util.*;
2)  import java.util.stream.*;
3)  public class Test
4)  {
5)     public static void main(String[] args)
6)     {
7)        ArrayList<Integer> l1 = new ArrayList<Integer>();
8)        for(int i =0; i <=10; i++)
9)        {
10)          l1.add(i);
11)       }
12)       System.out.println("Before using map() method:"+l1);
13)       List<Integer> l2=l1.stream().flatMap(
14)                  i->{ if (i%2 !=0) return Stream.empty();
15)                     else return Stream.of(i);
16)                   }).collect(Collectors.toList());
17)       System.out.println("After using flatMap() method:"+l2);
18)
19)       List<Integer> l3=l1.stream().flatMap(
20)                  i->{ if (i%2 !=0) return Stream.empty();
21)                     else return Stream.of(i,i*i);
22)                   }).collect(Collectors.toList());
23)       System.out.println("After using flatMap() method:"+l3);
24)    }
25) }
```

Output:

```
D:\durga_classes>java Test
Before using map() method:[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
After using flatMap() method:[0, 2, 4, 6, 8, 10]
After using flatMap() method:[0, 0, 2, 4, 4, 16, 6, 36, 8, 64, 10, 100]
```

Q. What is the difference between map() and flatMap() methods?

Java 9 Enhancements for Stream API:

In Java 9 as the part of Stream API, the following new methods introduced.

1. takeWhile()
2. dropWhile()

3. Stream.iterate()
4. Stream.ofNullable()

Note: takeWhile() and dropWhile() methods are default methods and iterate() and ofNullable() are static methods of Stream interface.

1. takeWhile():

It is the default method present in Stream interface.

```
default Stream  takeWhile(Predicate p)
```

It returns the stream of elements that matches the given predicate.
It is similar to filter() method.

Difference between takeWhile() and filter():

filter() method will process every element present in the stream and consider the element if predicate is true.

But, in the case of takeWhile() method, there is no guarantee that it will process every element of the Stream. It will take elements from the Stream as long as predicate returns true. If predicate returns false, at that point onwards remaining elements won't be processed, i.e rest of the Stream is discarded.

Eg: Take elements until we will get even numbers. Once we got odd number then stop and ignore rest of the stream.

Demo Program:

```
1)  import java.util.*;
2)  import java.util.stream.*;
3)  public class Test
4)  {
5)     public static void main(String[] args)
6)     {
7)        ArrayList<Integer> l1 = new ArrayList<Integer>();
8)        l1.add(2);
9)        l1.add(4);
```

```
10)       l1.add(1);
11)       l1.add(3);
12)       l1.add(6);
13)       l1.add(5);
14)       l1.add(8);
15)       System.out.println("Initial List:"+l1);
16)       List<Integer> l2=l1.stream().filter(i->i%2==0).collect(Collectors.toList());
17)       System.out.println("After Filtering:"+l2);
18)       List<Integer> l3=l1.stream().takeWhile(i->i%2==0).collect(Collectors.toList());
19)       System.out.println("After takeWhile:"+l3);
20)    }
21) }
```

Output:

```
Initial List:[2, 4, 1, 3, 6, 5, 8]
After Filtering:[2, 4, 6, 8]
After takeWhile:[2, 4]
```

Eg 2:

```
Stream.of("A","AA","BBB","CCC","CC","C").takeWhile(s-
>s.length()<=2).forEach(System.out::println);
```

Output:

```
A
AA
```

2. dropWhile()

It is the default method present in Stream interface.

```
default Stream  dropWhile(Predicate p)
```

It is the opposite of takeWhile() method.
It drops elements instead of taking them as long as predicate returns true. Once predicate returns false then rest of the Stream will be returned.

Demo Program:

```
1)  import java.util.*;
2)  import java.util.stream.*;
3)  public class Test
4)  {
5)     public static void main(String[] args)
6)     {
7)        ArrayList<Integer> l1 = new ArrayList<Integer>();
8)        l1.add(2);
9)        l1.add(4);
```

```
10)       l1.add(1);
11)       l1.add(3);
12)       l1.add(6);
13)       l1.add(5);
14)       l1.add(8);
15)       System.out.println("Initial List:"+l1);
16)       List<Integer> l2=l1.stream().dropWhile(i->i%2==0).collect(Collectors.toList());
17)       System.out.println("After dropWhile:"+l2);
18)    }
19) }
```

Output:

```
Initial List:[2, 4, 1, 3, 6, 5, 8]
After dropWhile:[1, 3, 6, 5, 8]
```

Eg 2:

```
Stream.of("A","AA","BBB","CCC","CC","C").dropWhile(s-
>s.length()<=2).forEach(System.out::println);
```

Output:

```
BBB
CCC
CC
C
```

3. Stream.iterate():

It is the satic method present in Stream interface.

Form-1: iterate() method with 2 Arguments

This method introduced in Java 8.

```
public static Stream iterate (T initial,UnaryOperator<T> f)
```

It takes an initial value and a function that provides next value.

Eg: Stream.iterate(1,x->x+1).forEach(System.out::println);

Output:

```
1
2
3
...infinite times
```

How to limit the number of iterations:

For this we can use limit() method.

Eg: Stream.iterate(1,x->x+1).limit(5).forEach(System.out::println);

Output:

```
1
2
3
4
5
```

Form-2: iterate() method with 3 arguments

The problem with 2 argument iterate() method is there may be a chance of infinite loop. To avoid, we should use limit method.

To prevent infinite loops, in Java 9, another version of iterate() method introduced, which is nothing but 3-arg iterate() method.

This method is something like for loop

```
for(int i =0;i<10;i++){}
```

```
public static Stream iterate(T initial,Predicate conditionCheck,UnaryOperator<T> f)
```

This method takes an initial value,
- **A terminate Predicate**
- **A function that provides next value.**

Eg: Stream.iterate(1,x->x<5,x->x+1).forEach(System.out::println);

Output:

```
1
2
3
4
```

4. ofNullable():

public static Stream<T> ofNullable(T t)

This method will check whether the provided element is null or not. If it is not null, then this method returns the Stream of that element. If it is null then this method returns empty stream.

This method is helpful to deal with null values in the stream
The main advantage of this method is to we can avoid *NullPointerException* and null checks everywhere.

Usually we can use this method in flatMap() to handle null values.

Eg 1:

```
List l=Stream.ofNullable(100).collect(Collectors.toList());
System.out.println(l);
```

Output:[100]

Eg 2:

```
List l=Stream.ofNullable(null).collect(Collectors.toList());
System.out.println(l);
```

Output: []

Demo Program:

```
1)  import java.util.*;
2)  import java.util.stream.*;
3)  import java.util.*;
4)  import java.util.stream.*;
5)  public class Test
6)  {
7)     public static void main(String[] args)
8)     {
9)        List<String> l=new ArrayList<String>();
10)       l.add("A");
11)       l.add("B");
12)       l.add(null);
13)       l.add("C");
14)       l.add("D");
15)       l.add(null);
16)       System.out.println(l);
17)
18)       List<String> l2= l.stream().filter(o->o!=null).collect(Collectors.toList());
19)       System.out.println(l2);
20)
21)       List<String> l3= l.stream()
```

```
22)                         .flatMap(o->Stream.ofNullable(o)).collect(Collectors.toList());
23)         System.out.println(l3);
24)     }
25) }
```

Output:

```
[A, B, null, C, D, null]
[A, B, C, D]
[A, B, C, D]
```

Demo Program:

```
1)  import java.util.*;
2)  import java.util.stream.*;
3)  public class Test
4)  {
5)      public static void main(String[] args)
6)      {
7)          Map<String,String> m=new HashMap<>();
8)          m.put("A","Apple");
9)          m.put("B","Banana");
10)         m.put("C",null);
11)         m.put("D","Dog");
12)         m.put("E",null);
13)         List<String> l=m.entrySet().stream().map(e->e.getKey()).collect(Collectors.toList());
14)         System.out.println(l);
15)
16)         List<String> l2=m.entrySet().stream()
17)                         .flatMap(e->Stream.ofNullable(e.getValue())).collect(Collectors.toList());
18)         System.out.println(l2);
19)
20)     }
21) }
```

Output:

```
[A, B, C, D, E]
[Apple, Banana, Dog]
```