

## 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:

**package** com.raos;

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.function.Function;

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

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

**class** suku{

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

{

ArrayList<Integer> a1=**new** ArrayList();

a1.add(10);

a1.add(20);

a1.add(30);

a1.add(40);

System.***out***.println("List:"+a1);

Function<Integer,Stream> f=(a)->{

**if**(a>30)

{

**return** Stream.*empty*();

}

**else** {

**return** Stream.*of*(a);

}

};

Function<Integer,Stream> f1=(a)->{

**if**(a>30)

{

**return** Stream.*of*(a);

}

**else** {

**return** Stream.*of*(a,a+2);

}

};

Stream s=a1.stream();

Stream v=a1.stream();

s=s.flatMap(f);

v=v.flatMap(f1);

List<Integer> l1=(List<Integer>)s.collect(Collectors.*toList*());

System.***out***.println("flatMap returns 0 or more ele:"+l1);

List<Integer> l2=(List<Integer>)v.collect(Collectors.*toList*());

System.***out***.println("flatMap returns 1 or more ele:"+l2);

}

}

Output:

List:[10, 20, 30, 40]

flatMap returns 0 or more ele:[10, 20, 30]

flatMap returns 1 or more ele:[10, 12, 20, 22, 30, 32, 40]

# 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.

## takeWhile():

It is the default method present in Stream public default Stream takeWhile(Predicate P);

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:

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.function.Function;

**import** java.util.function.Predicate;

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

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

**class** suku{

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

{

ArrayList<Integer> a1=**new** ArrayList();

a1.add(10);

a1.add(20);

a1.add(35);

a1.add(45);

a1.add(60);

System.***out***.println("List:"+a1);

Predicate<Integer>f=(a)->{

**if**(a%2==0)

{

**return** **true**;

}

**else** {

**return** **false**;

}

};

Function<Integer,Stream> f1=(a)->{

**if**(a%2==0)

{

**return** Stream.*of*(a);

}

**else**

{

**return** Stream.*empty*();

}

};

Stream s=a1.stream();

Stream v=a1.stream();

s=s.filter(f);

v=v.takeWhile(f);

List<Integer> l1=(List<Integer>)s.collect(Collectors.*toList*());

System.***out***.println("flatMap returns 0 or more ele:"+l1);

List<Integer> l2=(List<Integer>)v.collect(Collectors.*toList*());

System.***out***.println("flatMap returns 1 or more ele:"+l2);

}

}

Output:-

List:[10, 20, 35, 45, 60]

flatMap returns 0 or more ele:[10, 20, 60]

flatMap returns 1 or more ele:[10, 20]

1. 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:

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.function.Function;

**import** java.util.function.Predicate;

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

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

**class** suku{

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

{

ArrayList<Integer> a1=**new** ArrayList();

a1.add(10);

a1.add(20);

a1.add(35);

a1.add(45);

a1.add(60);

System.***out***.println("List:"+a1);

Predicate<Integer>f=(a)->{

**if**(a%2==0)

{

**return** **true**;

}

**else** {

**return** **false**;

}

};

Function<Integer,Stream> f1=(a)->{

**if**(a%2==0)

{

**return** Stream.*of*(a);

}

**else**

{

**return** Stream.*empty*();

}

};

Stream s=a1.stream();

Stream v=a1.stream();

s=s.filter(f);

v=v.dropWhile(f);

List<Integer> l1=(List<Integer>)s.collect(Collectors.*toList*());

System.***out***.println("flatMap returns 0 or more ele:"+l1);

List<Integer> l2=(List<Integer>)v.collect(Collectors.*toList*());

System.***out***.println("flatMap returns 1 or more ele:"+l2);

}

}

Output:

----------

List:[10, 20, 35, 45, 60]

flatMap returns 0 or more ele:[10, 20, 60]

flatMap returns 1 or more ele:[35, 45, 60]

## Stream.iterate():

## 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.

Example:-

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.function.Function;

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

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

**class** suku{

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

{

ArrayList<Integer> a1=**new** ArrayList();

a1.add(10);

a1.add(20);

a1.add(35);

a1.add(**null**);

a1.add(60);

System.***out***.println("List:"+a1);

Stream s=a1.stream();

Function<Integer,Stream>f1=(a)->{

**return** Stream.*ofNullable*(a);

};

s=s.flatMap(f1);

List<Integer> l1=(List<Integer>)s.collect(Collectors.*toList*());

System.***out***.println(l1);

}

}

Output:

-List:[10, 20, 35, null, 60]

[10, 20, 35, 60]

### 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]