**Difference between map() and flatMap() in Java 8 Stream**

when you use a [map()](http://java67.blogspot.com/2015/01/java-8-map-function-examples.html" \t "https://javarevisited.blogspot.com/2016/03/_blank), it applies a function on each element of stream and stores the value returned by the function into a new Stream. This way one stream is transformed into another e.g. a Stream of String is transformed into a Stream of Integer where each element is the length of the corresponding Stream.

The **Stream.map()** function performs map functional operation i.e. it take a Stream and transform it to another Stream. It applies a function on [each element of Stream](http://javarevisited.blogspot.com/2014/03/2-examples-of-streams-with-Java8-collections.html" \t "https://javarevisited.blogspot.com/2016/03/_blank) and store return value into new Stream. This way you can transform a Stream of String into a Stream of Integer where Integer could be [length of String](http://java67.blogspot.com/2014/09/right-way-to-check-if-string-is-empty.html" \t "https://javarevisited.blogspot.com/2016/03/_blank) if you supply the length() function. This is a very powerful function which is very helpful while dealing with collection in Java.  
Here is an example of Stream.map() in Java 8:

List listOfIntegers = Stream.of("1", "2", "3", "4")

.map(Integer::valueOf)

.collect(Collectors.toList());

In this example, we have a Stream of String values which represent numbers, by using map() function we have converted this Stream to Stream of Integers. How? by applying Integer.valueOf() on each element of Stream. That's how **"1"** converted to intger 1 and so on. Once transformation is done, we have collected the result into a List by [converting Stream to List using Collectors](http://javarevisited.blogspot.com/2015/03/5-ways-to-convert-java-8-stream-to-list.html" \t "https://javarevisited.blogspot.com/2016/03/_blank).  
  
The **Stream.flatMap()**function, as the name suggests, is the combination of a map and a flat operation. This means you first apply map function and then flattens the result.

To understand what flattening a stream consists in, consider a structure like [ [1,2,3],[4,5,6],[7,8,9] ] which has "two levels". It's basically a big List containing three more List.  Flattening this means transforming it in a "one level" structure e.g. [ 1,2,3,4,5,6,7,8,9 ] i.e. just one list.  
  
In short,  
Before flattening - Stream of List of Integer  
After flattening - Stream of Integer  
  
List evens = Arrays.asList(2, 4, 6);

List odds = Arrays.asList(3, 5, 7);

List primes = Arrays.asList(2, 3, 5, 7, 11);

List numbers = Stream.of(evens, odds, primes)

.flatMap(list -> list.stream())

.collect(Collectors.toList());

System.out.println("flattend list: " + numbers); Output: flattend list: [2, 4, 6, 3, 5, 7, 2, 3, 5, 7, 11]

## Stream.map() vs Stream.flatMap() in Java 8

In short, here are the key **difference between map() vs flatMap() in Java 8:**

* The function you pass to map() operation returns a single value.
* The function you pass to flatMap() operation returns a Stream of value.
* flatMap() is a combination of map and flat operation.
* map() is used for transformation only, but flatMap() is used for both transformation and flattening.

**Stream.peek()** method is very useful for debugging and understating the stream related code in Java.

1. The peek() method of Stream class is an intermediate method, hence you can call other stream methods after this.  
     
   2) It returns a new Stream, which is basically the stream it got.  
     
   3) It accepts an object of functional interface Consumer to perform non-interfering action e.g. printing values.  
     
   4) For parallel stream pipelines, the action may be called at whatever time and whatever thread the element is made available by the upstream operation.  
     
   You can use the peek() method for debugging. It allows you to see the elements as they flow past a certain point in the pipeline. By using this you can check whether your filter() method is working properly or not. You can see exactly which elements are got filtered by using peek() in Java 8.

The **Stream.peek()** method is mainly to support debugging, where you want to see the elements as they flow past a certain point in a pipeline.

The Stream.peek() method returns a stream consisting of the elements of this stream, additionally performing the provided action on each element as elements are consumed from the resulting stream.

For parallel stream pipelines, the action may be called at whatever time and in whatever thread the element is made available by the upstream operation. If the action modifies shared state, it is responsible for providing the required synchronization.

**public** **class** StreamPeekEx {

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

        Stream.of("bus", "car", "bycle", "flight", "train")

                .filter(e -> e.length() > 3)

                .peek(e -> System.out.println("Filtered value: " + e))

                .map(String::toUpperCase)

                .peek(e -> System.out.println("Mapped value: " + e))

                .collect(Collectors.toList());

    }

}

# How Java 8 Streams work

Streams support functional-style operations on streams of elements,

Stream operations are divided into intermediate and terminal operations, and are combined to form stream pipelines.

A stream pipeline consists of a source (such as a Collection, an array, a generator function, or an I/O channel); followed by zero or more intermediate operations such as Stream.filter or Stream.map; and a terminal operation such as Stream.forEach or Stream.reduce.

**Method References in Java 8 :**

Method reference is a shorthand notation of a lambda expression to call a method. For example:

If your lambda expression is like this:

**str -> System.out.println(str)**

then you can replace it with a method reference like this:

**System.out::println**

The **::** operator is used in method reference to separate the class or object from the method name

**Method references** are a special form of the [lambda expression](https://www.gauravbytes.com/2017/01/java-8-method-references.html" \t "https://dzone.com/articles/_blank). Since your lambda expressions are doing nothing other than invoking existing behavior (methods), you can achieve the same result by referring to it by name.

* :: is used to refer to a method.
* Method type arguments are inferred by JRE at runtime from the context it is defined.

**Four types of method references :**

* Static method reference
* Instance method reference of a particular object
* Instance method reference of an arbitrary object of a particular type
* Constructor reference

1. Method reference to an instance method of an object – object::instanceMethod

**MyInterface ref = obj::myMethod;**

2. Method reference to a static method of a class – Class::staticMethod

**BiFunction<Integer, Integer, Integer> product = Multiplication::multiply;**

**int pr = product.apply(11, 5);**

**System.out.println("Product of given number is: "+pr);**

3. Method reference to an instance method of an arbitrary object of a particular type – Class::instanceMethod

**Arrays.sort(stringArray, String::compareToIgnoreCase);**

4. Method reference to a constructor – Class::new

**MyInterface ref = Hello::new;**

**ref.display("Hello World!");**