**Q. What is the difference between Collection and Stream?**

**Ans**. The main difference between a Collection and Stream is that **Collection contains their elements**, but Stream doesn't. Stream work on a view where elements are actually stored by Collection or array, but unlike other views**, any change made on Stream doesn't reflect on the original Collection.**2 Examples of Streams with Collections in Java 8

Java 8 is bringing on new Streams API **java.util.stream package**, which allows we to **process elements of Java Collections in parallel**. Java is inheritably sequential and there are no direct means to introduce parallel processing at the library level, stream API is going to fill that gap. By using this, we can filter elements of collection on a given criterion e.g. if we have a list of orders, we can filter buy orders with sell orders, filter orders based upon there quantity and price, and so on.

**We can also perform two of the most popular functional programming functions e.g. map and reduce**. java.util.stream class provides function such **mapToInt**(), **mapToLong**(), and **map function** to apply an operation on all elements of Collections.  
I have chosen a List for these examples, but we can use any Collection e.g. Set, LinkedList, etc.  
By the way, the use of Stream is not limited to Collections only, we can even use an array, a generator function, or an I/O channel as the source. In most cases, a Stream pipeline in Java 8  consists of a source, followed by zero or more intermediate stream operations **e.g. filter() or map(); and a terminal operation such as forEach() or reduce().**  
For this example, I have a list of Orders, where each order contains bare minimum details like Side (buy or sell), price, quantity, and security.  
  
Once we initialized our list with some orders, we can perform interesting operations exposed by stream API. In the first example, we are using a filter() method to filter all sell orders.  
  
In the second example, we are using Stream APIs mapToDouble() method to calculate the total for both price and quantity, which would have to mean iterating over Collection and adding each element price in total. Because of Java 8 lambda expression and stream API, our code is reduced into pretty much one-liner.

import java.util.ArrayList;

import java.util.List;

import java.util.stream.Stream;

public class StreamDemo{

public static void main(String args[]) {

// Initialization of Collection

List<Order> orderBook = new ArrayList<>();

Order buyGoogle = new Order("GOOG.NS", 300, 900.30, Order.Side.BUY);

Order sellGoogle = new Order("GOOG.NS", 600, 890.30, Order.Side.SELL);

Order buyApple = new Order("APPL.NS", 400, 552, Order.Side.BUY);

Order sellApple = new Order("APPL.NS", 200, 550, Order.Side.SELL);

Order buyGS = new Order("GS.NS", 300, 130, Order.Side.BUY);

orderBook.add(buyGoogle);

orderBook.add(sellGoogle);

orderBook.add(buyApple);

orderBook.add(sellApple);

orderBook.add(buyGS);

// Java 8 Streams Example 1 : Filtering Collection elements

// Filtering buy and sell order using filter() method of java.util.Stream class

Stream<Order> stream = orderBook.stream();

Stream buyOrders = stream

.filter((Order o) -> o.side().equals(Order.Side.BUY));

System.out.println("No of Buy Order Placed :"

+ buyOrders.count()); // output - 3

Stream<Order> sellOrders

= orderBook

.stream()

.filter((Order o) -> o.side() == Order.Side.SELL); // output - 2

System.out.println("No of Sell Order Placed : " + sellOrders.count());

// Java 8 Streams Example 2 : Reduce or Fold operation

// Calculating total value of all orders

double value = orderBook.stream()

.mapToDouble((Order o) -> o.price())

.sum();

System.out.println("Total value of all orders : " + value);

long quantity = orderBook.stream()

.mapToLong((Order o) -> o.quantity())

.sum();

System.out.println("Total quantity of all orders : " + quantity);

}

}

class Order {

enum Side {

BUY, SELL;

}

private final String symbol;

private final int quantity;

private double price;

private final Side side;

public Order(String symbol, int quantity, double price, Side side) {

this.symbol = symbol;

this.quantity = quantity;

this.price = price;

this.side = side;

}

public double price() {

return price;

}

public void price(double price) {

this.price = price;

}

public String symbol() {

return symbol;

}

public int quantity() {

return quantity;

}

public Side side() {

return side;

}

}

Output:

No of Buy Order Placed :3

No of Sell Order Placed : 2

Total value of all orders : 3022.6

Total quantity of all orders : 1800

Important points about Java 8 Stream API

1) Stream API allows we to process Collection both sequentially and parallel. This is also useful for bulk data operation. We can create a sequential and parallel stream as follows :

List<Order> orders = getListOfOrders();

// sequential version

Stream<Order> stream = orders.stream();

//parallel version

Stream<Order> parallelStream = **orders.parallelStream();**

The collection interface is enhanced to provide stream support. It now has a stream() method which returns a sequential Stream with this collection as its source. Once we get the reference of stream, we can perform bulk data operations with this collection.  
  
**2**) One of the important things to note is that Stream does not modify the original source. **For every operation, a new Stream is created** and the original collection remains unmodified. Similarly, we cannot reuse Stream either. **Reusing a closed stream will throw IllegalStateException as shown below :**

Exception in thread "main" java.lang.IllegalStateException:

stream has already been operated upon or closed

at java.util.stream.AbstractPipeline.(AbstractPipeline.java:203)

at java.util.stream.ReferencePipeline.(ReferencePipeline.java:94)

at java.util.stream.ReferencePipeline$StatelessOp.(ReferencePipeline.java:618)

at java.util.stream.ReferencePipeline$2.(ReferencePipeline.java:163)

at java.util.stream.ReferencePipeline.filter(ReferencePipeline.java:162)

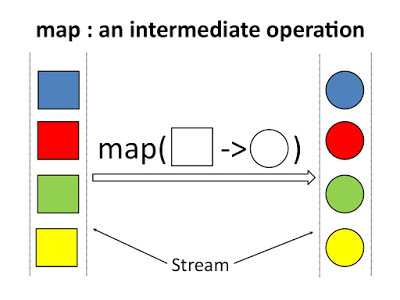
at Test.main(Test.java:33)

**3)** Stream operations are mainly divided into two categories : intermediate and terminal operations.

**Intermediate operations** => such as filter() or map() returns a new Stream, while

**Terminal operations=>**  such as Stream.forEach() produce a result or side effect. After the terminal operation, **the stream pipeline is considered consumed, and can no longer be used.**  
**4) Intermediate operations are also of two types stateless and stateful.** As the name suggests, stateless operations don't retain any state from the previously processed element, filter() and map() are two examples of stateless intermediate operation. On the other hand distinct() and sorted() are examples of stateful operations, which can have state from previously processed elements, while processing new elements.

**Q. What does the map() function do? why we use it?**

**Ans.** The map() function perform map functional operation in Java. This means it can transform one type of object to others by applying a function.  
For example, if we have a List of String and we want to convert that to a List of Integer, we can use map() to do so.  
Just supply a function to convert String to Integer e.g., parseInt() to map() and it will apply that to all elements of List and give we a List of Integer. In other words, the map can convert one object to another.  
The map is a well known functional programming concept that is incorporated into Java 8. Map is a function defined in java.util.stream.Streams class, which is used to transform each element of the stream by applying a function to each element. Prior to Java 8, there is no function to do this. We had to iterate through List using a for loop or foreach loop and transform each element. In Java 8, we get the stream, which allows we to apply many functional programming operators like the map, reduce, and filter.  
  
By using the map() function, we can apply any function to every element of Collection. It can be any **predefined function or a user-defined function.** We not only can use the lambda expression but also method references.  
Some examples of Map in Java 8 is to convert a list of integers and then the square of each number. The map function is also an intermediate operation and it returns a stream of the transformed element.  
  
Stream API also provides methods like **mapToDouble**(), **mapToInt**(), and **mapToLong**() which returns DoubleStream, IntStream and LongStream, which are specialized stream for double, int and long data types.  
  


How to use map() function in Java 8

Now let's see an example to convert each element of a List to upper case using the map function. Once again, the map applies a mapping function on each element of Stream and stores result in another Stream.

package test;

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

import static java.util.stream.Collectors.toList;

/\*\*

\* Java 8 example to convert each element of List into upper case. We can use

\* Map function of Java 8 to transform each element of List or any collection.

\* @author Javin

\*/

public class Java8MapExample {

public static void main(String args[]) {

List<String> cities = Arrays.asList("London", "HongKong",

"Paris", "NewYork");

System.out.println("Original list : " + cities); // Original list : [London, HongKong, Paris, NewYork]

System.out.println("list transformed using Java 8 :"

+ transform(cities)); //:[LONDON, HONGKONG, PARIS, NEWYORK]

System.out.println("list transformed using loop before Java 8 : "

+ beforeJava8(cities)); //:[LONDON, HONGKONG, PARIS, NEWYORK]

// We can even on the fly tranform Collection in Java using

// Map function

// let's transform a List of integers to square each element

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);

List<Integer> squares = numbers.stream()

.map( i -> i\*i)

.collect(Collectors.toList());

System.out.println("original list of numbers : " + numbers);// [1, 2, 3, 4, 5, 6, 7, 8, 9]

System.out.println("transformed list of integers using Map in Java 8 : "

+ squares);// [1, 4, 9, 16, 25, 36, 49,64, 81]

}

/\*\*

\* This is how we convert all elements of list into upper case

\* using loop before Java 8

\* @param listOfString

\* @return List with each element converted into upper case

\*/

public static List<String> beforeJava8(List<String> listOfString) {

List<String> coll = new ArrayList<>();

for (String str : listOfString) {

coll.add(str.toUpperCase());

}

return coll;

}

/\*\*

\* We can use Java 8 map function to transform each element of list

\* @param listOfString

\* @return list of elements with upper case

\*/

public static List<String> transform(List<String> listOfString) {

return listOfString.stream() // Convert list to Stream

.map(String::toUpperCase) // Convert each element to upper case

.collect(toList()); // Collect results into a new list

}

}

Output

run:

Original list : [London, HongKong, Paris, NewYork]

list transformed using Java 8 :[LONDON, HONGKONG, PARIS, NEWYORK]

list transformed using loop before Java 8 : [LONDON, HONGKONG, PARIS, NEWYORK]

original list of numbers : [1, 2, 3, 4, 5, 6, 7, 8, 9]

transformed list of integers using Map in Java 8 : [1, 4, 9, 16, 25, 36, 49,

64, 81]

BUILD SUCCESSFUL (total time: 0 seconds)

Explanation of Code

In this program, we have learned how to transform a List using map function in Java 8 and how it was done before Java 8 without using lambdas and **method reference**. In the first example, we have a list of String which contains the name of the famous cities in a small letter.We then converted that list into another list where names are in capital letters. This is achieved by applying the map() function on each element. The map method was passed the toUppercase() method to convert a small letter value to capital letter value.  
  
In the next example, we had a list of numbers from 1 to 9 and we transformed it into another list where numbers are square of each number. This is achieved by again applying a map function with calculating the square of each number passed to it.  
  
**Q. What does the filter() method do? when we use it?**

**Ans**. The filter method is used to filter elements that satisfy a certain condition that is specified using a Predicate function.  
**A predicate function** is nothing but a function that takes an Object and returns a boolean.

For example, if we have a List of Integer and we want a list of even integers.  
In this case, we can use the filter to achieve that. We supply a function to check if a number is even or odd, just like this function and filter will apply this to stream elements and filter the elements which satisfy the condition and which doesn't.  
  
**The filter() method** as its name suggests is used to **perform filtering based upon some boolean conditions.** The condition is applied to each element of Stream and those who pass the condition moves to the next stage and those who don't get filtered out.  
  
For example,  if we have a stream of integral numbers that contains both even and odd numbers then by using the filter method, we can create another stream of even numbers or odd numbers by filtering out others.  
  
Though filter() method is a little bit counter-intuitive, I mean, in order to create a stream of even number we call **filter( i -> i % 2 == 0)** which means we do **filter(isEven())** but, we are actually filtering out odd numbers to create a new stream of even numbers, but that's how it works.  
  
I think select() would have been a positive and proper name for this operation, but, we don't have any control over that can't change that now.  
The key benefit of using the filter() method is lazy evaluation i.e. no data comparison is performed unless we call a terminal operation on stream like findFirst() or forEach().  
  
The filter() method just sets up some pointers when we first call them on stream and only performs real filtering when we call the terminal method.

Let's see an example of a filter() method to understand the lazy evaluation it does.  
 Suppose we have a list of integer numbers and we want to find the first number which is divisible by both 2 and 3, let' see how to solve this problem in Java 8.

List<Integer> listOfNumbers = Arrays.asList(1, 2, 3, 4, 5, 6, 12, 18);

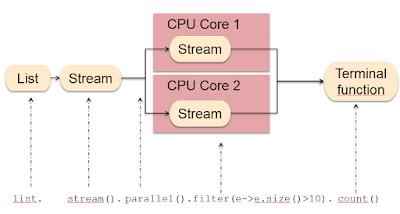
Integer lcm = listOfNumbers.stream()

.filter(i -> i % 2 == 0)

.filter(i -> i % 3 == 0)

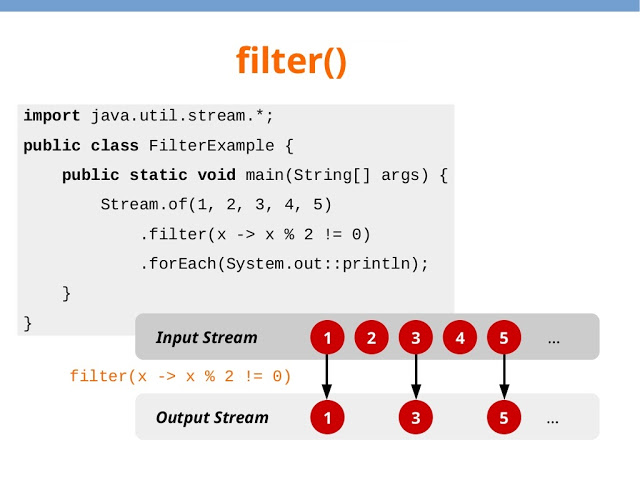
.findFirst().get();

This code is returning the first number which is divisible by both 2 and 3. Now, let's see how this code will execute. When we call the filter() method nothing happens until we call the findFirst().  
At this time, Java knows that it just needs to find the first element which satisfies the criterion imposed by the two chained filter() methods.  
The findFirst() asks the filter() method prior to it in the chain of any number, the filter doesn't have any record so it asks the first filter() method, which in turn then scans the list and returns a number which is divisible by 2.  
At this time, the second filter method checks if this number is divisible by 3, if yes then it returns that number to findFirst() otherwise it asks another number from the first filter() method.  
  
This process continues until a number is found which satisfies both filter() methods. Once that number is found it presented to the findFirst() method. The job of findFirst() is to return that number.  
  
This is an example of lazy evaluation because nothing happens until the call to findFirst() is a method, this also presents an opportunity to stop as soon as we find the first number which satisfies our criterion.  
**There is no need to process the entire list again and again**, as it happens in the case of iterative eager evaluation.



Java 8 filter Example

Here are a couple of more examples of the Stream.filter() method in Java 8. I have created a list of String containing Android versions like  Lollipop, KitKat, etc.  
  
The first example just uses one filter() method to print Strings whose length is greater than 10. The second example prints String which contains the letter "e" like Gingerbread.  
  
The Third examples combine these two filter methods to create a chain of filter methods to print String whose length is greater than 5 and starts with a letter "G".  
  
By the way, for testing purposes, we can also create a stream of integers number by using Stream.of() static factory methods as shown in the following example:



We can see that the input stream contains numbers from 1 to 5 but the output stream just contains odd numbers. This means even numbers were filtered out because they didn't satisfy the boolean condition specified by Predicate.  
  
I mean for even number x%2 == 0 and we are checking for x%2 !=0 so they didn't pass the condition and hence not progressed to the output stream.

Here is a sample Java program to demonstrate how to use the filter() method of Stream class to filter elements from a List or Stream, based upon some conditions, specified by the Predicate functional interface of Java 8.

package test;

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

/\*\*

\* Java 8 filter example. We can use filter() method to perform lazy filtering

\* in Java.

\*/

public class Java8FilterExample {

public static void main(String[] args) {

List<String> versions = new ArrayList<>();

versions.add("Lollipop");

versions.add("KitKat");

versions.add("Jelly Bean");

versions.add("Ice Cream Sandwidth");

versions.add("Honeycomb");

versions.add("Gingerbread");

// Using one filter()

// print all versions whose length is greater than 10 character

System.out.println("All versions whose length greater than 10");

versions.stream()

.filter(s -> s.length() > 10)

.forEach(System.out::println);

System.out.println("first element which has letter 'e' ");

String first = versions.stream()

.filter(s -> s.contains("e"))

.findFirst().get();

System.out.println(first);

// Using multiple filter

System.out.println("Element whose length is > 5 and startswith G");

versions.stream()

.filter(s -> s.length() > 8)

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

.forEach(System.out::println);

// another example of filter() method in Java 8

List<Integer> listOfNumbers = Arrays.asList(1, 2, 3, 4, 5, 6, 12, 18);

Integer lcm = listOfNumbers.stream()

.filter(i -> i % 2 == 0)

.filter(i -> i % 3 == 0)

.findFirst().get();

System.out.println("first number divisible by 2 and 3 in the list is : "

+ lcm);

}

}

Output

All versions whose length greater than 10

Ice Cream Sandwidth

Gingerbread

first element which has letter 'e'

Jelly Bean

Element whose length is > 5 and starts with G

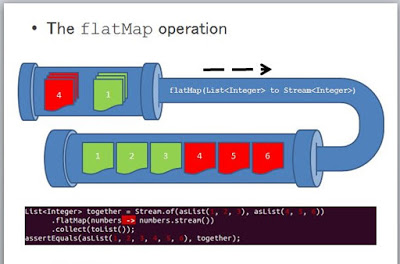
Gingerbread

a first number divisible by 2 and 3 in the list is : 6

The filter() method just set up a couple of pointers and no data comparison is performed until a terminal method like forEach() or findFirst() is called.  
  
**Q. What does the flatmap() function do? why we need it?**

**Ans**. The flatmap function is an extension of the map function. Apart from transforming one object into another, it can also flatten it.  
 For example, if we have a list of the list but we want to combine all elements of lists into just one list. In this case, we can use flatMap() for flattening. At the same time, we can also transform an object like we do use map() function.  
 In order to understand the flatMap() method, we first need to understand the map() function of Java 8. The map() function is declared in  the java.util.stream.Stream class and uses to transform one Stream into another, for example, it can convert a stream of integer numbers into another stream of ints where each element is the square of the corresponding element in the source stream. In the map() operation, a function is applied to each element of the source Stream and return values are inserted into a new Stream which is returned to the caller. The key point to note here is that the function used by map() operation returns a single value.  
  
Now, if the map operation uses a function which instead of returning a single value returns a Stream of values like when we give a number and it returns all prime factors of the number then we have a Stream of Stream of integers. **That's where the flatmap function helps.**  
  
**The flatMap() method** can be used to flatten that stream into a stream of integers. For example, suppose, we have a list of numbers like [21, 23, 42] and we call getPrimeFactors() method along with the map() operation to transform this stream.  
  
The result would be [[3,7],[23],[2,3,7]]. If we want to flatten this stream of a stream into a stream of values, we can use the flatMap() which will finally return [3,7,2,3,2,3,7].  
  
In short, the flatMap() function is used to convert a Stream of Stream into a list of values.

If we haven't got the whole picture and struggle to understand how exactly the flatMap function works then here is a diagram, which may help we to clear our debut.



We can see that there are two lists, one is denoted with red color while the other is denoted with green color, both of them contain three elements each. Once we apply flatMap on them, we can get another list that contains elements from them.  
  
We can see that in the second pipe where we have 3 elements from green lists denoted by 1, 2, and 3, and another three elements from red lists which is denoted by 4, 5, and 6.  
  
Java 8 FlatMap Example

Here is a sample Java program to demonstrate how to use the flatMap() function in Java 8. we can use the flatMap() to flatten a Stream of Stream of values into just a Stream of values.  
  
In our example, we have a Stream of the list of String and by using the flatMap() we convert this into just a Stream of String to get the full list of players participating in cricket world cup 2019.  
  
Here is our Java program to demonstrate how to use the Stream.flatMap() in Java 8.

package test;

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

/\*\*

\* Java Program to demonstrate how to use the flatMap() function in Java 8.

\* The flatMap() function is used to convert a Stream of list of values to

\* just a Stream of values. This is also called flattening of stream.

\*

\* @author Javin Paul

\*/

public class Test {

public static void main(String args[]) {

List<String> teamIndia = Arrays.asList("Virat", "Dhoni", "Jadeja");

List<String> teamAustralia = Arrays.asList("Warner", "Watson", "Smith");

List<String> teamEngland = Arrays.asList("Alex", "Bell", "Broad");

List<String> teamNewZeland = Arrays.asList("Kane", "Nathan", "Vettori");

List<String> teamSouthAfrica = Arrays.asList("AB", "Amla", "Faf");

List<String> teamWestIndies = Arrays.asList("Sammy", "Gayle", "Narine");

List<String> teamSriLanka = Arrays.asList("Mahela", "Sanga", "Dilshan");

List<String> teamPakistan = Arrays.asList("Misbah", "Afridi", "Shehzad");

List<List<String>> playersInWorldCup2016 = new ArrayList<>();

playersInWorldCup2016.add(teamIndia);

playersInWorldCup2016.add(teamAustralia);

playersInWorldCup2016.add(teamEngland);

playersInWorldCup2016.add(teamNewZeland);

playersInWorldCup2016.add(teamSouthAfrica);

playersInWorldCup2016.add(teamWestIndies);

playersInWorldCup2016.add(teamSriLanka);

playersInWorldCup2016.add(teamPakistan);

// **Let's print all players before Java 8**

List<String> listOfAllPlayers = new ArrayList<>();

for(List<String> team : playersInWorldCup2016){

for(String name : team){

listOfAllPlayers.add(name);

}

}

System.out.println("Players playing in world cup 2016");

System.out.println(listOfAllPlayers);

**// Now let's do this in Java 8 using FlatMap**

List<String> flatMapList = playersInWorldCup2016

.stream()

.flatMap(pList -> pList.stream())

.collect(Collectors.toList());

System.out.println("List of all Players using Java 8");

System.out.println(flatMapList);

}

}

Output

run:

Players playing in world cup 2016

[Virat, Dhoni, Jadeja, Warner, Watson, Smith, Alex, Bell, Broad, Kane,

Nathan, Vettori, AB, Amla, Faf, Sammy, Gayle, Narine, Mahela, Sanga, Dilshan,

Misbah, Afridi, Shehzad]

List of all Players using Java 8

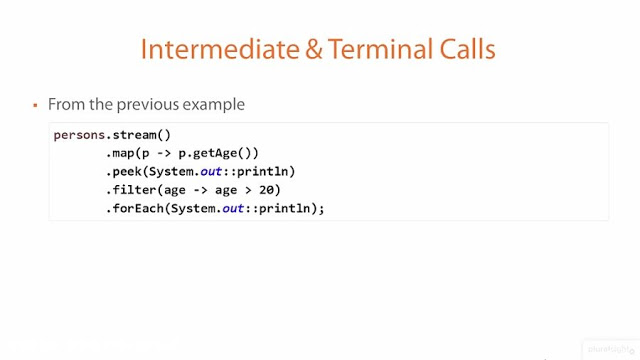
[Virat, Dhoni, Jadeja, Warner, Watson, Smith, Alex, Bell, Broad, Kane, Nathan,

Vettori, AB, Amla, Faf, Sammy, Gayle, Narine, Mahela, Sanga, Dilshan, Misbah,

Afridi, Shehzad]

BUILD SUCCESSFUL (total time: 0 seconds)

We can see that the final list contains all the elements from each list. So flatMap() is working fine to flatten a Stream of List of String into just a Stream of String, that's the true power of flat map operation in Java 8.  
  
**Q. What is difference between flatMap() and map() functions?**

Ans. Even though both map() and flatMap() can be used to transform one object to another by applying a method on each element.  
  
The main difference is that flatMap() can also flatten the Stream. For example, if we have a list of the list, then we can convert it to a big list by using flatMap() function.  
  


The map() and flatmap() are two important operations in new functional Java 8. Both represent functional operation and they are also methods in java.util.stream.Stream class. The key difference between map() and flatmap() function is that **when we use a map(), 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 key thing to remember is that the function used for transformation in the **map() returns a single value.** If map() uses a function, which, instead of returning a single value returns a Stream of values than we have a Stream of Stream of values, and **flatmap() is used to flat that into a Stream of values.**  
  
For example, if we have a Stream of String containing {"12", "34"}, and a method getPermutations() which returns a list of permutations of given String. When we apply that function into each String of Stream using map we will get something like [["12","21"],["34","43"]], but if we use flatmap, we get a Stream of Strings e.g. ["12", "21", "34", "43"].  
  
In this article, we'll see a couple of working examples to understand the difference between map() and flatmap() in Java better.

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 and store return value into new Stream. This way we can transform a Stream of String into a Stream of Integer where Integer could be length of String if we 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.

**Stream.flatMap() works in Java 8**

The Stream.flatMap() function, as the name suggests, is the combination of a map and a flat operation. This means we first apply map function and then flattens the result. The key difference is the function used by map operation returns a Stream of values or a list of values rather than a single value, that's why we need flattening. When we flat a Stream of Stream, it gets converted into Stream of values.  
  
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  
  
Here is a code example to understand the flatMap() function better:

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]

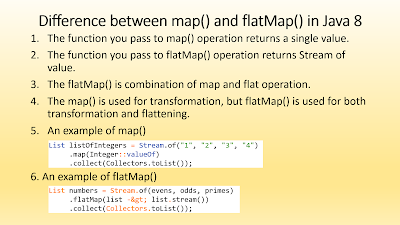
We can see that we have three lists that are merged into one by using a flatMap() function. For mapping, we can see we have used a list.stream() function which returns multiple values instead of a single value. Finally, we have collected the flattend stream into a list. If  we want, we can print the final list using the forEach() method.

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

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

* The function we pass to **map() operation returns a single value.**
* The function we 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.**

Now let's see a sample Java program to understand the differnce between flatMap() and map() better.



Here is our sample Java program to demonstrate the real difference between the map() and the flatMap() function of Stream class in Java 8. As I told before, map() is used to transform one Stream into another by applying a function on each element, and flatMap() does both transformations as well as flattening.  
  
The flatMap() function can take a Stream of List and return Stream of values combined from all those lists. In the example below, we have collected the result in a List but we can also print them using the forEach() method of Java 8.

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

/\*\*

\* Java Program to demonstrate difference between map()

\* vs flatMap() function in Java 8. Both are defined

\* in Stream class.

\*

\* @author WINDOWS 8

\*/

public class Java8Demo {

public static void main(String args[]) {

// foods which helps in weight loss

List<String> loseWeight = new ArrayList<>();

loseWeight.add("avocados");

loseWeight.add("beans");

loseWeight.add("salad");

loseWeight.add("oats");

loseWeight.add("broccoli");

System.out.println("list of String : " + loseWeight);

// let's use map() method to convert list of weight

// lose food, which are String to list of ints

// which are length of each food String

List listOfInts = loseWeight.stream()

**.map**(s -> s.length())

.collect(Collectors.toList());

System.out.println("list of ints generate by map(): " + listOfInts);

// flatMap() example, let's first creat a list of list

List<List> listOfListOfNumber = new ArrayList<>();

listOfListOfNumber.add(Arrays.asList(2, 4));

listOfListOfNumber.add(Arrays.asList(3, 9));

listOfListOfNumber.add(Arrays.asList(4, 16));

System.out.println("list of list : " + listOfListOfNumber);

// let's use flatMap() to flatten this list into

// list of integers i.e. 2,4,3,9,4,16

List listOfIntegers = listOfListOfNumber.stream()

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

.collect(Collectors.toList());

System.out.println("list of numbers generated by flatMap : "

+ listOfIntegers);

}

}

Output

list of String : [avocados, beans, salad, oats, broccoli]

list of ints generate by map(): [8, 5, 5, 4, 8]

list of list : [[2, 4], [3, 9], [4, 16]]

list of numbers generated by flatMap : [2, 4, 3, 9, 4, 16]

We can see that in the first example, the function used by map() method returns a single value, the length of the string passed to it, while in case of flatMap() the method returns a stream, which is basically our multiple values.  
  
**Q7. What does the peek() method do? When should we use it?**

**Ans**. The peek() method of Stream class allows we to see through a Stream pipeline. We can peek through each step and print meaningful messages on the console. **It's generally used for debugging issues related to lambda expression and Stream processing**.  
  
The peek() method returns a stream consisting of the elements of this stream and performs the action requested by the client. The peek() method expects a Consumer functional interface to perform a non-interfering action on the elements of this stream, usually printing them using the forEach() method.  
  
Btw, the **sole reason of using peek() is debugging** the Stream pipeline, even the API itself says that peek() is only there for debugging, but it does help to understand the lazy evaluation technique Stream uses to improve performance.  
  
**Lazy evaluation** means nothing is evaluated in the Stream until a terminal method like **forEach(), collect(), or reduce() is called** **and processing stops as soon as the result is obtained,** which means not all the elements of Stream is processed always.  
  
It all depends upon what kind of result we want from Stream. For example, if we call the findFirst() method then as soon as it finds the first element fulling the criterion, processing stops.

**Example**:

List<String> result = Stream.of("EURO/INR", "USD/AUD", "USD/GBP", "USD/EURO")

.filter(e -> e.length() > 7)

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

.map(String::toLowerCase)

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

.collect(Collectors.toList());

In this example, we have a Stream of String and then we are filtering all Strings whose length is greater than 7 and then we are converting them to lowercase using the map() function.  
  
Many of we will think that after the first filter() execution we will get a Stream containing two elements "EURO/INR" and "USD/EURO" and **peek() will print those two elements.  
  
Well, that's not the case**, **since Streams are executed lazily,** **nothing will happen until the** **collect() method will execute**, which is the terminal method.  
  
This is proved by the following output from running the above code into Eclipse IDE or command prompt, it will print the following lines:

Filtered value: EURO/INR

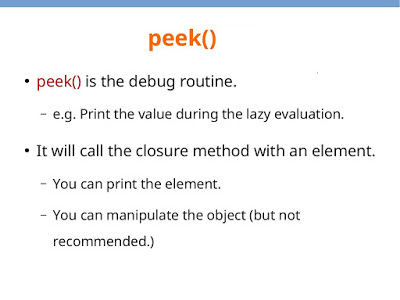
Mapped value: euro/inr

Filtered value: USD/EURO

Mapped value: usd/euro

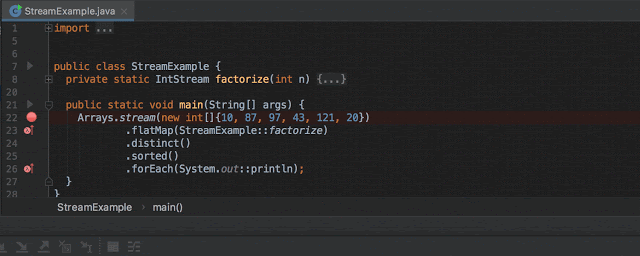
The key point to note here is that **values are filtered and mapped one by one,** not together. It means the **code is executed backward when the collect() method calls the Collectors.toList() to get the result in a List**, it asks map() function which in turn asks the filter() method.  
  
**Since filter() is lazy it returns the first element whose length is greater than 7 and sits back until map() ask again.**

So, the second filter() again ask the first() filter for an element, it returns Jelly Bean, Ice Cream Sandwich, and HoneyComb one by one. Since HoneyComb made passed the second filter it is collected by Collector and again the same process happens but aborted after GingerBread because all elements in Stream are already processed.



Important points

1) The **peek() method of Stream class is an intermediate method**, **hence we 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.  
Btw, peek() is not the only way to figure out what goes inside a Stream pipeline, we can also use our IDEs to do the heavy work.



**Q. What do we mean by saying Stream is lazy?**

**Ans**. When we say Stream is lazy, we mean that most of the **methods defined on Java .util.stream.Stream class is lazy** i.e. they will not work by just including them on Stream pipeline.  
They only **work when we call a terminal method** on the Stream and **finish as soon as they find the data they are looking for rather than scanning through the whole set of data.**  
**Q. How to Filter Collections in Java 8 with Streams and Predicates**

**Ans**. Java 8 provides excellent features to support the filtering of elements in Java Collections. Prior to Java 8, the only better way to filter elements is by using a foreach loop or iterating over Collection using the Iterator and selecting the required object, leaving out rest. Though that approach work**, it was very difficult to run them in parallel and take advantage of multiple CPU available in modern-day servers**. **Java 8 provides Streams, which not only makes it easy to run any operation parallel but also** **support lazy loading and lazy evaluation, which means as soon as filtering condition is satisfied, it stooped doing work,** doesn't matter how many object collection contains.  
  
We can filter Java Collections like List, Set or Map in Java 8 by using filter() method of Stream class. We first need to obtain a stream from Collection **by calling stream**() method and then **we can use filter() method**, which takes a Predicate as the only argument.  
 **Predicate is a functional interface with a couple of boolean-valued methods e.g. test(), which returns boolean true or false**. Java 8 uses this method to filter Collection. Just remember that filter doesn't remove elements that match the condition given in predicate, instead, it selects them in the output stream.  
  
The Predicate is applied to each element of Collection to check whether a particular element should be included in a filtered stream or not. Predicate should be stateless and non-interfering so that if needed filter operation can run in parallel using parallel stream.   
  
In Java8, **Predicate is a functional interface with a couple of boolean valued methods used to test input against the condition.**  
In our case test(T t) method is used, which evaluates this predicate in a given argument. Since the **filter() method accepts a functional interface**, we can also pass a lambda expression to it, which is what we will do. We can pass any condition to filter elements like either by using relational operator e.g. less than, greater than or equal to or by using methods like equals() and equalsIgnoreCase() as shown in our sample program.

Java 8 Example of Filtering List using Stream

This is our sample program to demonstrate the power of the Java 8 Stream and Filter method. By using these Java 8 enhancements we can perform SQL like operation in Java e.g.  
  
SELECT \* FROM Deals WHERE type = 'ELECTRONIC'

can be written using Java 8 stream and filter method as

deals.stream()

.filter(deal -> deal.type() == Deal.Type.ELECTRONIC)

.forEach(System.out::println)

In this example, we are passing a lambda expression to the filter method, which returns a boolean. It's actually anonymous function test(), we have omitted type information for deal variable because it will be inferred by JVM easily, making our code concise.  
In short, we can pass a lambda expression to the filter method until the result is boolean. The forEach() method is a terminal operation and used here to print all deals present in the filtered collection.  
  
Now let's see the second example of a filtering List in Java. Now we need all deals which are expiring in March. In SQL we can write a query like this :

SELECT \* FROM Deals WHERE validity='MARCH'

and in Java 8, we can write the following code :  
deals.stream()

.filter(deal -> deal.validity().getMonth() == Month.MARCH)

.forEach(System.out::println);

Our third example is about filtering elements based upon greater than condition. How about getting all deals which with 30% or more discounts? We can write the following query in SQL :

SELECT \* FROM Deals WHERE discountPercentage >= 30;

In Java 8 we can use filter method  like below to do the same :

deals.stream()

.filter(deal -> deal.discount()

.compareTo(new BigDecimal("00.30")) > 0)

.forEach(System.out::println);

Let's see one more example of filtering collection in Java. How about finding all deals from Apple, don't we know iPad and iPhone? We can write SQL query like following to get those deals :

SELECT \* FROM Deals WHERE provider='Apple';

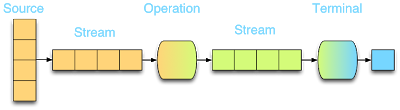
In Java 8, We can do  :

deals.stream()

.filter(deal -> deal.provider().equalsIgnoreCase("Apple"))

.forEach(System.out::println);

Here is the complete code which we can run in our favorite IDE or command prompt if we are a down to earth programmer.



How to use Filter method in Java 8

package test;

import java.math.BigDecimal;

import java.time.LocalDate;

import java.time.Month;

import java.util.ArrayList;

import java.util.List;

/\*\*

\* Simple Java value class to represent a deal

\*/

public class Deal {

public enum Type {

BOOK,

ELECTRONIC,

TRAVEL,

COSMATIC,

ACTIVITY,

}

private final String provider;

private final Type type;

private final BigDecimal price;

private final BigDecimal discount;

private final String title;

private final LocalDate validity;

public Deal(String provider, Type type, BigDecimal price,

BigDecimal discount, String title, LocalDate validity) {

this.provider = provider;

this.type = type;

this.price = price;

this.discount = discount;

this.title = title;

this.validity = validity;

}

public String provider() {

return provider;

}

public Type type() {

return type;

}

public BigDecimal price() {

return price;

}

public BigDecimal discount() {

return discount;

}

public String title() {

return title;

}

public LocalDate validity() {

return validity;

}

@Override

public String toString() {

StringBuilder sb = new StringBuilder();

sb.append(title).append(" from ").append(provider).

append(", price : ").append(price).

append(", offer valid till ").append(validity).

append(" category : ").append(type);

return sb.toString();

}

}

/\*\*

\* Java 8 Example to filter Collection on Predicates using Stream API. this

\* program also uses new Date and Time API, lambdas, method reference etc.

\*

\* @author Javin Paul

\*/

public class Java8FilterDemo {

public static void main(String args[]) {

List deals = loadDeals();

System.out.println("All Deals");

System.out.println("--------------------------------");

// this will print all deals from list

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

System.out.println("--------------------------------");

// Filtering elements from a Collection in Java 8

// filtering on category

System.out.println("All deals for Electornic items");

deals.stream()

.filter(deal -> deal.type() == Deal.Type.ELECTRONIC)

.forEach(System.out::println);

System.out.println("--------------------------------");

// filter all deals which are expiring on March

System.out.println("Deals expiring on March");

deals.stream()

.filter(deal -> deal.validity().getMonth() == Month.MARCH)

.forEach(System.out::println);

System.out.println("--------------------------------");

**// filter all deals which has more than 30% discount**

System.out.println("All deals with 30% or more discount");

deals.stream()

.filter(deal -> deal.discount()

.compareTo(new BigDecimal("00.30")) > 0)

.forEach(System.out::println);

System.out.println("--------------------------------");

// filter all deals from companies

System.out.println("All deals from Apple");

deals.stream()

.filter(deal -> deal.provider().equalsIgnoreCase("Apple"))

.forEach(System.out::println);

System.out.println("--------------------------------");

}

private static List loadDeals() {

List deals = new ArrayList<>();

deals.add(new Deal("Manning", Deal.Type.BOOK,

new BigDecimal("30.00"), new BigDecimal(".50"),

"Save 50% on Java 8 Books",

LocalDate.of(2014, Month.MARCH, 20)));

deals.add(new Deal("Amazon", Deal.Type.BOOK,

new BigDecimal("20.00"), new BigDecimal(".20"),

"Save 20% on Clean Code",

LocalDate.of(2014, Month.FEBRUARY, 10)));

deals.add(new Deal("Kathy Pacific", Deal.Type.TRAVEL,

new BigDecimal("300.00"), new BigDecimal(".40"),

"Save 40% on flight to USA",

LocalDate.of(2014, Month.FEBRUARY, 19)));

deals.add(new Deal("Luftanse", Deal.Type.TRAVEL,

new BigDecimal("30.00"), new BigDecimal(".50"),

"Save 50% on flight to Berlin",

LocalDate.of(2014, Month.MARCH, 27)));

deals.add(new Deal("Trekking", Deal.Type.ACTIVITY,

new BigDecimal("400.00"), new BigDecimal(".50"),

"Save 50% on Trekking",

LocalDate.of(2014, Month.MARCH, 25)));

deals.add(new Deal("Apple", Deal.Type.ELECTRONIC,

new BigDecimal("800.00"), new BigDecimal(".10"),

"10% discount on iPhone 5S",

LocalDate.of(2014, Month.APRIL, 19)));

deals.add(new Deal("Samsung", Deal.Type.ELECTRONIC,

new BigDecimal("700.00"), new BigDecimal(".20"),

"20% discount on Galaxy S4",

LocalDate.of(2014, Month.MARCH, 18)));

deals.add(new Deal("LG", Deal.Type.ELECTRONIC,

new BigDecimal("390.00"), new BigDecimal(".50"),

"Save 40% on LG Smartphones",

LocalDate.of(2014, Month.FEBRUARY, 17)));

deals.add(new Deal("Sony", Deal.Type.ELECTRONIC,

new BigDecimal("500.00"), new BigDecimal(".50"),

"Save 50% on Sony Viao Laptops",

LocalDate.of(2014, Month.APRIL, 10)));

return deals;

}

}

Output:

All Deals

--------------------------------

Save 50% on Java 8 Books from Manning, price : 30.00,

offer valid till 2014-03-20 category : BOOK

Save 20% on Clean Code from Amazon, price : 20.00,

offer valid till 2014-02-10 category : BOOK

Save 40% on flight to USA from Kathy Pacific, price : 300.00,

offer valid till 2014-02-19 category : TRAVEL

Save 50% on flight to Berlin from Luftanse, price : 30.00,

offer valid till 2014-03-27 category : TRAVEL

Save 50% on Trekking from Trekking, price : 400.00,

offer valid till 2014-03-25 category : ACTIVITY

10% discount on iPhone 5S from Apple, price : 800.00,

offer valid till 2014-04-19 category : ELECTRONIC

20% discount on Galaxy S4 from Samsung, price : 700.00,

offer valid till 2014-03-18 category : ELECTRONIC

Save 40% on LG Smartphones from LG, price : 390.00,

offer valid till 2014-02-17 category : ELECTRONIC

Save 50% on Sony Viao Laptops from Sony, price : 500.00,

offer valid till 2014-04-10 category : ELECTRONIC

--------------------------------

All deals for Electornic items

10% discount on iPhone 5S from Apple, price : 800.00,

offer valid till 2014-04-19 category : ELECTRONIC

20% discount on Galaxy S4 from Samsung, price : 700.00,

offer valid till 2014-03-18 category : ELECTRONIC

Save 40% on LG Smartphones from LG, price : 390.00,

offer valid till 2014-02-17 category : ELECTRONIC

Save 50% on Sony Viao Laptops from Sony, price : 500.00,

offer valid till 2014-04-10 category : ELECTRONIC

--------------------------------

Deals expiring on March

Save 50% on Java 8 Books from Manning, price : 30.00,

offer valid till 2014-03-20 category : BOOK

Save 50% on flight to Berlin from Luftanse, price : 30.00,

offer valid till 2014-03-27 category : TRAVEL

Save 50% on Trekking from Trekking, price : 400.00,

offer valid till 2014-03-25 category : ACTIVITY

20% discount on Galaxy S4 from Samsung, price : 700.00,

offer valid till 2014-03-18 category : ELECTRONIC

--------------------------------

All deals with 30% or more discount

Save 50% on Java 8 Books from Manning, price : 30.00,

offer valid till 2014-03-20 category : BOOK

Save 40% on flight to USA from Kathy Pacific, price : 300.00,

offer valid till 2014-02-19 category : TRAVEL

Save 50% on flight to Berlin from Luftanse, price : 30.00,

offer valid till 2014-03-27 category : TRAVEL

Save 50% on Trekking from Trekking, price : 400.00,

offer valid till 2014-03-25 category : ACTIVITY

Save 40% on LG Smartphones from LG, price : 390.00,

offer valid till 2014-02-17 category : ELECTRONIC

Save 50% on Sony Viao Laptops from Sony, price : 500.00,

offer valid till 2014-04-10 category : ELECTRONIC

--------------------------------

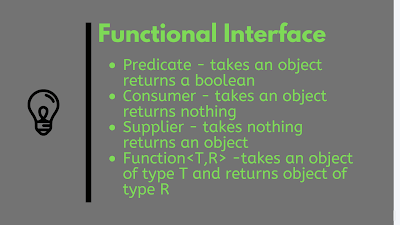
All deals from Apple

10% discount on iPhone 5S from Apple, price : 800.00,

offer valid till 2014-04-19 category : ELECTRONIC

**Q. What is a functional interface in Java 8?**

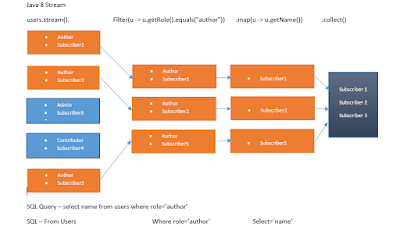
**Ans**. As the name suggests, a **functional interface is an interface** that represents a function. Technically, an **interface with just one abstract method is called a functional interface.**  
  
We can also use **@FunctionalInterface to annotated a functional interface**. In that case, the compiler will **verify if the interface actually contains just one abstract method or not.** It's like the @Override annotation, which prevents we from accidental errors.  
  
Another useful thing to know is that **If a method accepts a functional interface**, then **we can pass a lambda expression to it.**  
Some examples of the functional interface are **Runnable**, **Callable**, **Comparator**, and **Comparable** from old API and **Supplier, Consumer, and Predicate**, etc. from new function API.

These interface with just one abstract method was used to pass around code, just like we pass a function in functional programming language and that's why they are called functional interface.  
  
For example, we can directly pass the code to compare objects by creating an Anonymous class by implementing the Comparator interface as shown below:  
  
Collections.sort(list, new Comparator(){  
  
   public int compare(String s1, String s2){  
      return s1.length() - s2.length();  
   }  
});  
So, if we look closely, we can see we are passing code to a function using these interfaces. **They are also known as strategy interfaces** because this is an implementation of a Strategy pattern where the code which forms the Strategy is injected into the code which runs that strategy at runtime.  
  


What does @FunctionalInterface annotation do?

Now, let's see what does @FunctionalInterface annotation does? Will it make an interface functional if we just put the @FunctionalInterface annotation on top of that? **Well, no, it doesn't do that. In fact, it's optional.**

This means we can create a functional interface without using @Functioanl annotation just like we can override a method without putting the @Override annotation on top of a method. Then, what is the real purpose of @FunctionalInterface annotation?  
  
Well, it can ensure that the interface actually has just one abstract method and it also provides a hint to the tools like Javadoc that this interface is a functional interface. It's pretty much similar to @Override annotation, which helps **to prevent human error by verifying that we actually overriding a method.**  
Similar to @Override its best practice to put the @FunctionalInterface annotation on top of the method with single abstract methods to indicate to the tools like Javadoc that they are a functional interface.  
  
All the new functional interfaces added **in java.util.function package** is annotated with the @FunctionalInterface annotation.  
  
Btw, Yes, we have got more functional interfaces in JDK 8, particularly **general-purpose functional interfaces** like **Predicate**, **Supplier**, **Consumer**, Function, **BiFunction**, **UnaryOperator**, etc.  
  
These functional interfaces allow we to the passcode to a function in form of lambda expression and enable the creation of powerful methods which can operate on those code like filter() method which accepts a Predicate and allows we to pass a code which accepts one argument and return boolean.



**A BiFunction** is a functional interface that has a method that accepts two arguments T and U and returns an object R.  
  
This means we can pass a lambda to this method which works on two arguments and return an object **like merge(key, value, (v1, v2) -> v1 + v2)**here (v1, V2) -> v1 + v2 is a lambda expression which can be converted into an instance of BiFunction functional interface.  
  
A rather simpler example is Predicate which accepts a type T and returns a boolean. If we look filter() method of Stream class it accepts a Predicate:  
  
filter(Predicate predicate)  
  
This **means we can pass any lambda expression which accepts one argument and return boolean** to this method e.g. age -> age > 15 or s -> s.length == 15, both are acceptable, but if we don't know what is a Predicate interface then we can't do that.  
  
Another example of a functional interface is Consumer which accepts an argument of type T and returns nothing. **Good use of this is made in the forEach() method of Iterable in JDK 8, as shown below:**  
**forEach(Consumer action)**  
  
We can see that forEach() accepts a Consumer, which means we can pass it a lambda expression which has one argument and returns nothing or the void e.g.  
  
s -> System.out.println(s)  
  
The code System.out.println() return nothing, it just prints line in the console.

**Q. What is the difference between the findFirst() and findAny() method?**

**Ans**. The findFirst() method will return the first element meeting the criterion i.e. Predicate, while the findAny() method will return any element meeting the criterion, very useful while working with a parallel stream.

**findFirst() Example**

In Java 8, we can use the**Stream.findFirst() method to get the first element of Stream in Java. This is a terminal operation** and often used after applying several intermediate operations e.g. filter, mapping, flattening, etc. For example, if we have a List of String and we want to find the first String whose length is greater than 10, we can use the findFirst() method along with stream() and filter() to get that String. The stream() method gets the Stream from a List, which then allows we to apply several useful methods defined in the java.util.Stream class e.g. filter(), map(), flatMap() etc.  
  
**One of the important things to know while using for writing such logic is that all intermediate operations** e.g. filter(), map(), etc are lazy and they are only executed when a terminal operation like findFirst() or forEach() is called.

Let's say, we have a List of String and we want to find out the first String which has a length greater than 10, how do we solve this problem before Java 8? Well, in Java 6 or 7 we can use the enhanced for loop as shown below to get the first String which satisfies our requirement as shown below:

for (String gadget : gadgets) {

if (gadget.length() > 10) {

System.out.println("Prior ot Java 8: " + gadget);

break;

}

}

In Java 8, we can achieve the same effect by using the findFirst() method of java.util.stream.Stream class as shown below:

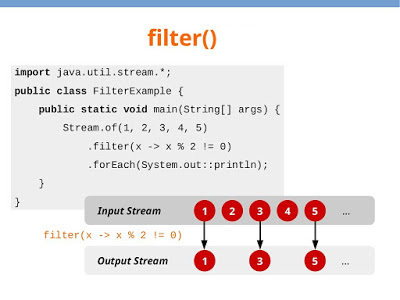
String item = gadgets.stream()

.filter(s -> s.length() > 10)

.findFirst()

.orElse("");

Many Java Programmers think that this is an inefficient approach because it looks like the filter() method is scanning the whole list as opposed to breaking on the first element as we have done in our prior to Java 8 example.  
  
This is not true, Java 8 example is as efficient as the one before. The filter() is an intermediate operation that is lazy and only evaluated when we call a terminal method e.g. forEach() method or findFirst() method as in this case.



This means, the filter() will not scan the whole list but only first few elements until it got the one whose length is greater than 10. We can verify this by using peek() method which prints output of what happening during stream processing steps as shown below:

String item = gadgets.stream()

.peek(s -> System.out.println("processing: " + s))

.filter(s -> s.length() >8)

.findFirst()

.orElse("");

Output:

processing: SmartPhone

result: SmartPhone

We can see that filter() has processed just one element from the Stream. Once it found the "SmartPhone", it stops because that satisfies the findFirst() method.  
  
Also, it's worth knowing that all intermediate stream operations are lazy (See Java 8 in Action). They are the operation that returns another Stream instead of value. The terminal operation returns a value or creates a side-effect . So, don't worry about performance, one of the key reason for using stream is lambda is a lazy evaluation which presents optimization opportunity depending upon the algorithm.  
  
Java Program to find the first element of Stream in JDK 8

Here is our Java program to demonstrate how we can use stream utility methods like map(), flatMap(), and findFirst() to filter and find out the first element from Stream in Java 8.

import java.util.ArrayList;

import java.util.List;

/\*

\* Java Program to show how to find the first element

\* of Stream in Java 8.

\*/

public class Java8Demo {

public static void main(String args[]) {

// a list of electronic gadgets

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

gadgets.add("SmartPhone");

gadgets.add("SmartWatch");

gadgets.add("SmartTV");

gadgets.add("SmartDoor");

gadgets.add("iPhone");

// printing gadgets whose length is greater than 7

// using pre Java 8 techniques

for (String gadget : gadgets) {

if (gadget.length() > 7) {

System.out.println("Prior ot Java 8: " + gadget);

break;

}

}

// retrieving gadgets with length greater than 8

// using lambda expression and stream methods

// in Java 8

String item = gadgets.stream()

.filter(s -> s.length() > 8)

.findFirst()

.orElse("");

System.out.println("In Java 8, first item: " + item);

// we can further use peek() to see

// what's going inside filter or other stream

// methods.

String myItem = gadgets.stream()

.peek(s -> System.out.println("processing: " + s))

.filter(s -> s.length() > 8)

.findFirst()

.orElse("");

System.out.println("result: " + myItem);

}

}

Output:

Prior ot Java 8: SmartPhone

In Java 8, first item: SmartPhone

processing: SmartPhone

result: SmartPhone

That's all about how to find the first element in Stream in Java 8. We can use the findFist() method to get the first object. It's a terminal operation and should be the last call on stream because after that we cannot call any method on Stream. Intermediate operations are evaluated lazily.

**Q. What is a Predicate interface?**

A Predicate is a functional interface that represents a function, which takes an Object and returns a boolean. It is used in several Stream methods like filter(), which uses Predicate to filter unwanted elements.  
  
here is how a Predicate function looks like:  
  
public boolean test(T object){  
   return boolean;   
}  
  
We can see it just has one test() method which takes an object and returns a boolean. The method is used to test a condition if it passes; it returns true otherwise false.  
  
  
**Q. What are Supplier and Consumer Functional interface?**

**Ans**. The Supplier is a functional interface that returns an object. It's similar to the factory method or **new(), which returns an object.**  
  
**The Supplier has get() functional method**, which doesn't take any argument and return an object of type T. This means we can use it anytime we need an object.  
  
Since it is a functional interface, we can also use it as the assignment target for a lambda expression or method reference.  
  
**A Consumer is also a functional interface in JDK 8,** which represents an operation that accepts a single input argument and returns no result.  
  
Unlike other functional interfaces, Consumer is expected to operate via side-effects. The functional method of Consumer is accept(T t), and because it's a functional interface, we can use it as the assignment target for a lambda expression or method interface in Java 8.

**Q. What is the parallel Stream? How can we get a parallel stream from a List?**

**Ans**. A parallel stream can parallel execute stream processing tasks. For example, if we have a parallel stream of 1 million orders and we are looking for orders worth more than 1 million, then we can use a filter to do that.  
Unlike sequential Stream, the parallel Stream can launch multiple threads to search for those orders on the different part of Stream and then combine the result.  
In short, the parallel Stream can paralyze execution, but, as Cay S. Horstman mentioned in Core Java S.E. 9 for the Impatient, there is significant overhead or parallelism, which only pays off if we are doing bulk data operation.  
  
**Q. What New Features Were Added in Java 8?**

Java 8 ships with several new features but the most significant are the following:

**Lambda Expressions** − **a new language feature allowing treating actions as objects**

**Method References** − enable defining Lambda Expressions by referring to methods directly using their names

**Optional** − special wrapper class used for expressing optionality

**Functional Interface** – an interface with maximum one abstract method, implementation can be provided using a Lambda Expression

**Default methods** − give us the ability to add full implementations in interfaces besides abstract methods

**Nashorn, JavaScript Engine** − Java-based engine for executing and evaluating JavaScript code

**Stream API** − a special iterator class that allows processing collections of objects in a functional manner

**Date API** − an improved, immutable JodaTime-inspired Date API

Along with these new features, lots of feature enhancements are done under-the-hood, at both compiler and JVM level.

**Q1. What Is a Method Reference?**

**Ans.** **A method reference is a Java 8 construct that can be used for referencing a method without invoking it.** It is used for treating methods as Lambda Expressions. They only work as syntactic sugar to reduce the verbosity of some lambdas. This way, the following code:

(o) -> o.toString();

**can become:**

Object::toString();

A method reference can be identified by a double colon separating a class or object name and the name of the method. It has different variations such as

constructor reference: - String::new;

Static method reference: -String::valueOf;

Bound instance method reference: - str::toString;

Unbound instance method reference: - String::toString;

**Q1. What Is Optional? How Can It Be Used?**

**Ans.** **Optional is a new class in Java 8** that encapsulates an optional value i.e. a value that is either there or not. **It is a wrapper around an object, and we can think of it as a container of zero or one element.**

**Optional has a special Optional.empty() value instead of wrapped null.** Thus it can be used instead of a nullable value to **get rid of NullPointerException in many cases**.

The main purpose of Optional, as designed by its creators, was to be a return type of methods that previously would return null. Such methods would require we to write boilerplate code to check the return value and sometimes could forget to do a defensive check. In Java 8, an Optional return type explicitly requires we to handle null or non-null wrapped values differently.

For instance, the Stream.min() method calculates the minimum value in a stream of values. But what if the stream is empty? If it was not for Optional, the method would return null or throw an exception.

But it returns an Optional value which may be Optional.empty() (the second case). This allows us to easily handle such case:

int min1 = Arrays.stream(new int[]{1, 2, 3, 4, 5})

.min()

.orElse(0);

assertEquals(1, min1);

int min2 = Arrays.stream(new int[]{})

.min()

.orElse(0);

assertEquals(0, min2);

It's worth noting that Optional is not a general purpose class like Option in Scala. It is not recommended to be used as a field value in entity classes, which is clearly indicated by it not implementing the Serializable interface.

**Q1. Describe Some of the Functional Interfaces in the Standard Library.**

**Ans.** There are a lot of functional interfaces in the java.util.function package, the more common ones include but not limited to:

* **Function** – it takes one argument and returns a result
* **Consumer** – it takes one argument and returns no result (represents a side effect)
* **Supplier** – it takes not argument and returns a result
* **Predicate** – it takes one argument and returns a boolean
* **BiFunction** – it takes two arguments and returns a result
* **BinaryOperator** – it is similar to a BiFunction, taking two arguments and returning a result. The two arguments and the result are all of the same types
* **UnaryOperator** – it is similar to a Function, taking a single argument and returning a result of the same type

For more on functional interfaces, see the article “Functional Interfaces in Java 8”.

**Q. What Is a Functional Interface? What Are the Rules of Defining a Functional Interface?**

**Ans.** A functional interface is an interface with no more, no less but one single abstract method (default methods do not count).

Where an instance of such interface is required, a Lambda Expression can be used instead. More formally put: Functional interfaces provide target types for lambda expressions and method references.

The arguments and return type of such expression directly match those of the single abstract method.

For instance, the Runnable interface is a functional interface, so instead of:

Thread thread = new Thread(new Runnable() {

public void run() {

System.out.println("Hello World!");

}

});

we could simply do:

**Thread thread** **= new Thread(**() -> System.out.println("Hello World!"));

Functional interfaces are usually annotated with the @FunctionalInterface annotation – which is informative and does not affect the semantics.

**Q. What Is a Default Method and When Do We Use It?**

**Ans.** **A default method is a method with an implementation** – which can be found in an interface.

**We can use a default method to add a new functionality to an interface while maintaining backward compatibility with classes that are already implementing the interface**:

public interface Vehicle {

public void move();

**default void hoot() {**

**System.out.println("peep!");**

**}**

}

**Usually, when a new abstract method is added to an interface, all implementing classes will break until they implement the new abstract method.** In Java 8, this problem has been solved by the use of default method.

For example, Collection interface does not have forEach method declaration. Thus, adding such method would simply break the whole collections API.

Java 8 introduces default method so that Collection interface can have a default implementation of forEach method without requiring the classes implementing this interface to implement the same.

**Q. Will the Following Code Compile?**

@FunctionalInterface

public interface Function2<T, U, V> {

**public V apply(T t, U u);**

**default void count() {**

**// increment counter**

**}**

}

**Ans. Yes.** The code will compile because it follows the functional interface specification of defining **only a single abstract method**. The **second method, count, is a default method** that does not increase the abstract method count.

**Q. What Is a Lambda Expression and What Is It Used for**

**Ans.** In very simple terms, **a lambda expression is a function** **that can be referenced** and **passed around as an object.**

**Lambda expressions introduce functional style processing in Java and facilitate the writing of compact and easy-to-read code.**

Because of this**, lambda expressions are a natural replacement for anonymous classes as method arguments**. One of their main uses is to define inline implementations of functional interfaces.

**Q . Explain the Syntax and Characteristics of a Lambda Expression**

**Ans.** A lambda expression consists of two parts: **the parameter part** and **the expressions part** separated by a forward arrow as below:

**params** -> expressions

Any lambda expression has the following characteristics:

**1. Optional type declaration –** when declaring the parameters on the left-hand side of the lambda, we don't need to declare their types as the compiler can infer them from their values. So **int param ->** … and **param ->…** **are all valid**

**2.Optional parentheses** – when only a single parameter is declared, we don't need to place it in parentheses. This means param -> … and (param) -> … are all valid. But when more than one parameter is declared, parentheses are required

**3. Optional curly braces** – when the expressions part only has a single statement, there is no need for curly braces. This means that param **– > statement** and param – > {statement;} are all valid. But curly braces are required when there is more than one statement

**4.Optional return statement** – when the expression returns a value and it is wrapped inside curly braces, then we don't need a return statement. That means (a, b) – > {return a+b;} and (a, b) – > {a+b;} are both valid

**Q. What Is a Stream? How Does It Differ from a Collection?**

**Ans.** In simple terms, **a stream is an iterator whose role is to accept a set of actions to apply on each of the elements it contains.**

**The stream represents a sequence of objects from a source such as a collection**, **which supports aggregate operations.** **They were designed to make collection processing simple and concise**. Contrary to the collections, **the logic of iteration is implemented inside the stream**, so we can use methods like map and flatMap for performing a declarative processing.

Another difference is that the Stream API is fluent and allows pipelining:

int sum = Arrays.stream(new int[]{1, 2, 3})

.filter(i -> i >= 2)

.map(i -> i \* 3)

.sum();

And yet another important distinction from collections is that **streams are inherently lazily loaded and processed.**

**Q. What Is the Difference Between Intermediate and Terminal Operations?**

**Ans.** Stream operations are combined into pipelines to process streams. **All operations are either intermediate or terminal.**

**Intermediate operations are those operations that return Stream itself** allowing for further operations on a stream.

**These operations are always lazy,** i.e. **they do not process the stream at the call site,** an **intermediate operation can only process data when there is a terminal operation**. Some of the **intermediate operations are filter, map and flatMap.**

**Terminal operations terminate the pipeline and initiate stream processing**. The stream is passed through all intermediate operations during terminal operation call. **Terminal operations include forEach, reduce, Collect and sum.**

To drive this point home, let us look at an example with side effects:

public static void main(String[] args) {

System.out.println("Stream without terminal operation"); //1

Arrays.stream(new int[] { 1, 2, 3 }).map(i -> {

System.out.println("doubling " + i**); // Nothing will print here becz not calling terminal method**

return i \* 2;

});

System.out.println("Stream with terminal operation");

Arrays.stream(new int[] { 1, 2, 3 }).map(i -> {

System.out.println("doubling " + i);

return i \* 2;

}).sum(); **// Print only after this line instead of above println line.**

}

The output will be as follows:

Stream without terminal operation //1

Stream with terminal operation

doubling 1

doubling 2

doubling 3

As we can see, the intermediate operations are only triggered when a terminal operation exists.

**Q. What Is Stream Pipelining in Java 8?**

**Ans**. Stream pipelining is the concept of chaining operations together. This is done by splitting the operations that can happen on a stream into two categories**: intermediate operations and terminal operations.**

**Each intermediate operation returns an instance of Stream itself** when it runs, an arbitrary number of intermediate operations can, therefore, be set up to process data forming a processing pipeline.

**There must then be a terminal operation which returns a final value** and terminates the pipeline.

**Q. Tell Us About the New Date and Time API in Java 8**

**Ans**. A long-standing problem for Java developers has been the inadequate support for the date and time manipulations required by ordinary developers.

The existing classes such as**java.util.Date and SimpleDateFormatter aren’t thread-safe, leading to potential concurrency issues for users.**

Poor API design is also a reality in the old Java Data API. Here's just a quick example – years **in java.util.Date start at 1900, months start at 1, and days start at 0 which is not very intuitive**.

These issues and several others have led to the popularity of third-party date and time libraries, such as Joda-Time.

In order to address these problems and provide better support in JDK, a new date and time API, which is free of these problems, has been designed for Java SE 8 under the package java.time.

**Q. Ways to Convert Java 8 Stream to an Array ?**

**Ans**. We can pass a lambda expression or constructor reference to this method to specify the type of array we want. This will return we an array of T i.e. if String contains String then it will return String array.  
  
For example, streamOfString.toArray(String[]::new) will convert a Stream of String to an array of String **and streamOfInts.toArray(int[]::new)** will return **an int[] from IntStream**.  
  
Now, let's see some code examples of converting a Java 8 Stream to an array. As I told, we can convert the Stream into an object array using toArray() method and an array of type T using overloaded Stream.toArray(IntFunction[] generator), we'll see examples of each of them to understand better.

Stream to an object array

This is quite straightforward, no trick about this one. We can simply call the toArray() method on Stream and it will give we an object array which contains all elements of the corresponding stream, as shown in the following example:

Stream<String> loans = Stream.of("Car Loan", "Home Loan", "Personal Loan");

Object[] objectArray = loans.toArray();

**System.out.println(Arrays.toString(objectArray));**

**Output**

**[Car Loan, Home Loan, Personal Loan]**

The only problem with this method is that it returns an object array that our client may not want because even if the object array contains String, we cannot pass it to a method expecting String[] in Java. The array is not covariant in Java.

A stream of T to an array of T **via Stream.toArray()** and lambda expression

If we want an array of type T from Stream of T i.e. a String array from Stream of String then we need to use the overloaded version of toArray() method which expects a function which takes an integer and returns an array of the specified type. We can pass a lambda expression to this method to create an array of T as shown below:

Stream powerOfTen = Stream.of(1, 10, 100, 1000, 10000);

Integer[] array = powerOfTen.toArray(size -> new Integer[size]);

System.out.println(Arrays.toString(array));

Output

[1, 10, 100, 1000, 10000]

We can see that how easily the Stream of Integers is converted into Integer array. The lambda expression size -> new Integer[size] return an array of Integer by expecting size, which is an int parameter.  
  
Suppose, we want to convert a Stream of Integer to int[] and not Integer[] then we can use the mapToInt() function as shown below:

int[] intArray = powerOfTen.mapToInt(x -> x).toArray();

System.out.println(Arrays.toString(intArray));

Output

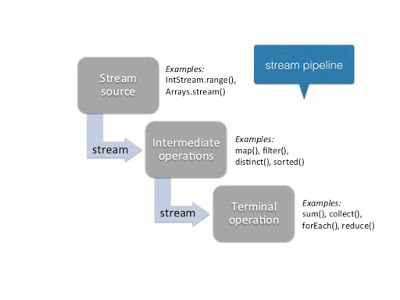
[1, 10, 100, 1000, 10000]

The Stream.mapToInt() function return an IntStream by converting all elements of Stream to int values. The java.util.strea.IntStream is a specialized Stream for int primitive values, hence it's toArray() method also return an int[] instead of Object array.  
  
Suppose, we have a Stream of String which is numbers e.g. "1", "2", or "3" then also we can use mapToInt() to first convert String to int and then into an int array as shown below:

Stream<String> numbers = Stream.of("1", "2", "3", "4", "5");

int[] ints = numbers.mapToInt(Integer::parseInt).toArray();

The Integer::parseInt() is equal to lambda expression String str -> Integer.parseInt(str), hence used to convert each number String to the int value.



T[] arrayOfT = streamOfT.toArray(T[]::new)

The **T[]::new is an array reference in Java 8**. It's a like a method reference which accepts an int value and returns an array of specified type. It is equal to lambda expression int i -> new T[i].  
  
Let's convert a Stream of String object into a String array using this technique, here is the example:

Stream<String> cities = Stream.of("London", "Paris", "Tokyo");

String[] arrayOfCities = cities.toArray(String[]::new);

System.out.println(Arrays.toString(arrayOfCities));

This constructor reference solves a major limitation of Java. If we remember, it's not possible to create a generic array in Java e.g. T[] = new T[] will give compile time error. By using this trick, we can overcome that limitation and return a typed array from Stream instead of Object[].  
  
If we want to learn more about this limitation and how constructor reference helps to get around them, I suggest reading Java SE 8 for Really Impatient by Cay S. Horstmann to learn more about that.

Stream to Array in Java 8 using Collector

This is the 3rd way to convert a Java 8 Stream to array. The idea is simple, we first convert Stream to ArrayList using any methods given here and then convert that ArrayList to an array using any methods given here. This is not exactly a Java 8 way to do the job but it allows we to use our existing knowledge of ArrayList to array conversion in Java 8.

Stream numbers = Stream.of(11, 22, 33, 44, 55);

ArrayList list = numbers.collect(Collectors.toCollection(ArrayList::new));

Integer[] iArray = list.toArray(new Integer[list.size()]);

The collect() method is used to accumulate all elements fo Stream into a Collection e.g. a list. Though, there is a method called Collectors.toList() which converts Stream to List, it doesn't provide any guarantee about the type of list returned by this method. Instead, by using Collectors.toCollection() we can convert it into any specific Collection e.g. ArrayList. This is particularly useful if we want to preserve the order of elements while converting Stream to array in Java 8. We can read Java 8 in  Action to learn more about Collectors.

Java Program to convert Stream to array

Here is our complete Java example to convert a Stram to array in Java 8. In this example, we will learn, how to convert the stream to object array, how to convert a stream of T to an array of T using lambda expression and constructor reference in Java, and finally, stream to array conversion using Collectors and ArrayList. We can choose whichever way we like, but the constructor reference example i.e. toArray(String[]::new) is the simplest, easiest and the best way to convert a Stream to array in Java.

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

import java.util.stream.Stream;

/\*\*

\* Java Program to convert a Java 8 Stream to Array

\*/

public class Java8Demo {

public static void main(String[] args) {

// stream to object array in Java

Stream<String> currencies = Stream.of("INR", "USD", "GBP", "EUR", "JPY");

Object[] objectArray = currencies.toArray();

System.out.println("Stream to object array in Java:");

System.out.println(Arrays.toString(objectArray));

// via - Stream.toArray() and lambda expression

Integer[] primes = {2, 3, 5, 7, 11};

List listOfInts = new ArrayList<>(Arrays.asList(primes));

Integer[] array = listOfInts.stream()

.toArray(size -> new Integer[size]);

System.out.println("Stream to Integer array using lambda expression in Java:");

System.out.println(Arrays.toString(array));

// via - method reference

array = listOfInts.stream()

.toArray(Integer[]::new);

System.out.println("Stream to Integer array using method reference in Java:");

System.out.println(Arrays.toString(array));

// via arraylist

ArrayList list = listOfInts.stream()

.collect(Collectors.toCollection(ArrayList::new));

Integer[] iArray = list.toArray(new Integer[list.size()]);

System.out.println("Stream to Integer array via ArrayList in Java:");

System.out.println(list);

}

}

}

Output

Stream to object array in Java:

[INR, USD, GBP, EUR, JPY]

Stream to Integer array using a lambda expression in Java:

[2, 3, 5, 7, 11]

Stream to Integer array using method reference in Java:

[2, 3, 5, 7, 11]

Stream to Integer array via ArrayList in Java:

[2, 3, 5, 7, 11]

Important points about Stream to array conversion

1. The java.util.stream.Stream class provides toArray() method to convert a Stream to array in Java. This method is overloaded to return an object array and an array of type T.  
  
2. We can use lambda expression or constructor reference to convert a Stream of T into array of T e.g. toArray(x -> new int[x]) will convert Stream of int to int[]. Similarly, toArray(x -> String[x]) will convert Stream of String to String array.  
  
3. The best way to convert a Stream to an array in Java 8 is by using constructor reference i.e. toArray(int[]::new), this will convert Stream to int array. The int[]::new is a constructor reference and it is similar to a method which expects an integer and returns an int[]. It is equivalent to x -> int[x] lambda expression, but it's slightly easier to read and write.  
  
  
4. We can also convert Stream to an array by first converting it into an ArrayList. For that, we can use the collect() method to accumulate stream elements into a Collection e.g. ArrayList. The Collectors.toList() method will return a list of Stream elements but type of list is not guaranteed. If we want an ArrayList, just use Collectors.toCollection() method with constructor reference e.g. Collectors.toCollection(ArrayList::new))  
  
5. We can use same ways to convert a parallel stream to array as well.  
  
6. The Stream.toArray() method performs a terminal operator on Stream, hence we cannot reuse the Stream after calling this method. Any attempt to reuse Stream after calling toArray() will throw the following error:  
  
Exception in thread "main" java.lang.IllegalStateException: stream has already been operated upon or closed  
  
7. If we a Stream of Integer but we want to convert them into int[] and not Integer[], then we can use the mapToInt() function to convert Integer to int before converting Stream to array in Java as shown below:

streamOfInteger.mapToInt(x -> x).toArray();

The mapToInt() function return an IntStream, a specialized stream for primitive int type, the toArray() method of IntStream return int[] instead of Object[].  
  
  
  
  
**The best way to convert Stream to array in Java is by using toArray() and constructor reference i.e. toArray(T[]::new),** it is both concise and clear. It's slightly less readable for first timers but once we know that **T[]::new** **create an array of T** and **expect an integer as size,** it's much easier to write and read. Constructor reference can really make our code looks good,