**Java 8 Functional Programing (Lamda Expresions, Functional Interface and Streams)**

**Q**) Can we do whatever stream will do the functionality using collections, if yes then why do we need to go to streams and what is difference between Streams and collections?

1. Major difference is collections and its methods **operates/work on source/data structure** directly that means it will **modify/change the structure** of source.
2. **Streams** never modify the source data, it will process the data and it has its own operations like internal methods and terminal methods which will produce another collections object based on our requirements.

There are many ways to create a stream instance of different sources. Once created, the instance will not modify its source, therefore allowing the creation of multiple instances from a single source.

**Streams:**

**How many ways we can create Stream object?**

**1. Stream of Array:**

* An array can also be the source of a stream:

|  |
| --- |
| Stream<String> streamOfArray = Stream.of("a", "b", "c"); |

* We can also create a stream out of an existing array or of part of an array:

|  |
| --- |
| String[] arr = new String[]{"a", "b", "c"};  Stream<String> streamOfArrayFull = **Arrays.stream**(arr);  Stream<String> streamOfArrayPart = Arrays.**stream**(arr, 1, 3); |

* We can also create a stream of any type of Collection **(Collection, List, Set)**

|  |
| --- |
| Collection<String> collection = Arrays.asList("a", "b", "c");  Stream<String> streamOfCollection = collection.stream(); |

**2.Stream.iterate():**

* Another way of creating an infinite stream is by using the iterate() method:

|  |
| --- |
| Stream<Integer> streamIterated = Stream.iterate(40, n -> n + 2).limit(20); |

* The first element of the resulting stream is the first parameter of the iterate() method. When creating every following element, the specified function is applied to the previous element. In the example above the second element will be 42.

**3.Stream of String:**

* We can also use String as a source for creating a stream with the help of the **chars**() method of the String class.
* Since there is no interface for CharStream in JDK, we use the IntStream to represent a stream of chars instead.

|  |
| --- |
| **IntStream streamOfChars = "abc".chars();** |

* The following example breaks a String into sub-strings according to specified RegEx:

|  |
| --- |
| Stream<String> streamOfString = Pattern.compile(", ").**splitAsStream**("a, b, c"); |

**4. Stream of File:**

* Furthermore, Java NIO class Files allows us to generate a Stream<String> of a text file through the **lines**() method.
* Every line of the text becomes an element of the stream:

|  |
| --- |
| Path path = Paths.get("C:\\file.txt");  Stream<String> streamOfStrings = **Files**.**lines**(path);  Stream<String> streamWithCharset =  **Files.lines(path, Charset.forName("UTF-8"));** |

The Charset can be specified as an argument of the lines() method.

**5. Referencing a Stream**:

**Important point:**

* We can instantiate a stream, and have an accessible reference to it, as long as only intermediate operations are called.
* Executing a terminal operation makes a stream inaccessible.

**Example:**

* To demonstrate this, we will forget for a while that the best practice is to chain the sequence of operation.
* Besides its unnecessary verbosity, technically the following code is valid:

|  |
| --- |
| Stream<String> stream =  Stream.of("a", "b", "c").filter(element -> element.contains("b"));  Optional<String> anyElement = stream.findAny(); |

* However, an attempt to reuse the same reference after calling the terminal operation will trigger the IllegalStateException: This kind of behavior is logical. We designed streams to apply a finite sequence of operations to the source of elements in a functional style, not to store elements.

|  |
| --- |
| Optional<String> firstElement = stream.findFirst(); |

As the IllegalStateException is a RuntimeException, a compiler will not signalize about a problem. So it is very important to remember that Java 8 streams can’t be reused.

So to make the previous code work properly, some changes should be made:

|  |
| --- |
| List<String> **elements** = Stream.of("a", "b", "c").filter(element -> element.contains("b")) .collect(Collectors.toList());  Optional<String> anyElement = **elements**.stream().findAny();  Optional<String> firstElement = **elements**.stream().findFirst(); |

**6. Stream Pipeline:**

* To perform a sequence of operations over the elements of the data source and aggregate their results, we need three parts: **the source, intermediate operation(s) and a terminal operation.**
* Intermediate operations return a new modified stream. For example, to create a new stream of the existing one without few elements, the skip() method should be used:

|  |
| --- |
| Stream<String> onceModifiedStream = Stream.of("abcd", "bbcd", "cbcd").skip(1); |

* If we need more than one modification, we can chain intermediate operations. Let’s assume that we also need to substitute every element of the current Stream<String> with a **sub-string of** the first few chars. We can do this by chaining the **skip**() and **map**() methods:

|  |
| --- |
| Stream<String> twiceModifiedStream =  stream.skip(1).map(element -> element.substring(0, 3)); |

* A stream by itself is worthless; the user is interested in the result of the terminal operation, which can be a value of some type or an action applied to every element of the stream. We can only use one terminal operation per stream.
* The correct and most convenient way to use streams is by a stream pipeline, which is a chain of the stream source, intermediate operations, and a terminal operation:

|  |
| --- |
| List<String> list = Arrays.asList("abc1", "abc2", "abc3");  long size = list.stream().skip(1)  .map(element -> element.substring(0, 3)).sorted().count(); |

**7. Lazy Invocation:**

* Intermediate operations are lazy. This means that they will be invoked only if it is necessary for the terminal operation execution.
* For example, let’s call the method wasCalled(), which increments an inner counter every time it’s called:

**Example:**

|  |
| --- |
| private long **counter**;//Instance attribute    private void **wasCalled**() {  counter++;  } |

* Now let’s call the method wasCalled() from operation filter():

|  |
| --- |
| List<String> list = Arrays.asList(“abc1”, “abc2”, “abc3”);  counter = 0;  Stream<String> stream = list.stream().filter(element -> {  **wasCalled();**  return element.contains("2");  }); |

* As we have a source of three elements, we can assume that the filter() method will be called three times, and the value of the counter variable will be 3.
* However, running this code doesn’t change counter at all, it is still zero, so the filter() method wasn’t even called once.
* The reason why is missing of the terminal operation.

Let’s rewrite this code a little bit by adding a map() operation and a terminal operation, findFirst(). We will also add the ability to track the order of method calls with the help of logging

|  |
| --- |
| Optional<String> stream = list.stream().filter(element -> {  log.info("filter() was called");  return element.contains("2");  }).map(element -> {  log.info("map() was called");  return element.toUpperCase();  }).findFirst(); |

* The resulting log shows that we called the filter() method twice and the map() method once.
* This is because the pipeline executes vertically. In our example, the first element of the stream didn’t satisfy the filter’s predicate.
* Then we invoked the filter() method for the second element, which passed the filter. Without calling the filter() for the third element, we went down through the pipeline to the map() method.
* The findFirst() operation satisfies by just one element. So in this particular example, the lazy invocation allowed us to avoid two method calls, one for the filter() and one for the map().

**8. Order of Execution:**

* From the performance point of view, the right order is one of the most important aspects of chaining operations in the stream pipeline:

|  |
| --- |
| long size = list.stream().map(element -> {  wasCalled();  return element.substring(0, 3);  }).skip(2).count(); |

* Execution of this code will increase the value of the counter by three. This means that we called the map() method of the stream three times, but the value of the size is one.
* So the resulting stream has just one element, and we executed the expensive map() operations for no reason two out of the three times.

If we change the order of the skip() and the map() methods, the counter will increase by only one. So we will call the map() method only once:

|  |
| --- |
| long size = list.stream().skip(2).map(element -> {  wasCalled();  return element.substring(0, 3);  }).count(); |

* This brings us to the following rule: intermediate operations which reduce the size of the stream should be placed before operations which are applying to each element.
* So we need to keep methods such as skip(), filter(), and distinct() at the top of our stream pipeline.

**9. Stream Reduction:**

* The API has many terminal operations which aggregate a stream to a type or to a primitive: count(), max(), min(), and sum().
* However, these operations work according to the predefined implementation.
* So what if a developer needs to customize a Stream’s reduction mechanism? There are two methods which allow us to do this,

1. the **reduce**() and
2. the **collect**() methods.

**The reduce() Method:**

There are three variations of this method, which differ by their signatures and returning types. They can have the following parameters:

1. **identity** – the initial value for an accumulator, or a default value if a stream is empty and there is nothing to accumulate
2. **accumulator** – a function which specifies the logic of the aggregation of elements. As the accumulator creates a new value for every step of reducing, the quantity of new values equals the stream’s size and only the last value is useful. This is not very good for the performance.
3. **combiner** – a function which aggregates the results of the accumulator. We only call combiner in a parallel mode to reduce the results of accumulators from different threads.

Now let’s look at these three methods in action:

|  |
| --- |
| OptionalInt reduced = IntStream.range(1, 4).reduce((a, b) -> a + b); |

Output: reduced = 6 (1 + 2 + 3)

|  |
| --- |
| int reducedTwoParams =  IntStream.range(1, 4).reduce(10, (a, b) -> a + b); |

Output: reducedTwoParams = 16 (10 + 1 + 2 + 3)

|  |
| --- |
| int reducedParams = Stream.of(1, 2, 3)  .reduce(10, (a, b) -> a + b, (a, b) -> {  log.info("combiner was called");  return a + b;  }); |

Note: The result will be the same as in the previous example (16), and there will be no login, which means that combiner wasn’t called.

To make a combiner work, a stream should be parallel:

|  |
| --- |
| int reducedParallel = Arrays.asList(1, 2, 3).**parallelStream()**  .reduce(10, (a, b) -> a + b, (a, b) -> {  log.info("combiner was called");  return a + b;  }); |

**Explanation:**

* The result here is different (36), and the combiner was called twice. Here the reduction works by the following algorithm: the accumulator ran three times by adding every element of the stream to identity. These actions are being done in parallel.
* As a result, they have (10 + 1 = 11; 10 + 2 = 12; 10 + 3 = 13;). Now combiner can merge these three results. It needs two iterations for that (12 + 13 = 25; 25 + 11 = 36).

**The collect() Method:**

* The reduction of a stream can also be executed by another terminal operation, the collect() method.
* It accepts an argument of the type Collector, which specifies the mechanism of reduction.
* There are already created, predefined collectors for most common operations.
* They can be accessed with the help of the Collectors type.

|  |
| --- |
| List<Product> productList = Arrays.asList(new Product(23, "potatoes"),  new Product(14, "orange"), new Product(13, "lemon"),  new Product(23, "bread"), new Product(13, "sugar")); |

Converting a stream to the Collection (Collection, List or Set):

|  |
| --- |
| List<String> collectorCollection productList.stream().map(Product::getName).collect(Collectors.toList()); |

Reducing to String:

|  |
| --- |
| String listToString = productList.stream().map(Product::getName) .collect(Collectors.joining(", ", "[", "]")); |

**Explanation**:

* The joiner() method can have from one to three parameters (delimiter, prefix, suffix).
* The most convenient thing about using joiner() is that the developer doesn’t need to check if the stream reaches its end to apply the suffix and not to apply a delimiter.
* Collector will take care of that.

Processing the average value of all numeric elements of the stream:

|  |
| --- |
| double averagePrice = productList.stream() .collect(Collectors.averagingInt(Product::getPrice)); |

Explanation:

The methods averagingXX(), summingXX() and summarizingXX() can work with primitives (int, long, double) and with their wrapper classes (Integer, Long, Double). One more powerful feature of these methods is providing the mapping. As a result, the developer doesn’t need to use an additional map() operation before the collect() method.

**10. Parallel Streams:**

* Before Java 8, parallelization was complex. The emergence of the **ExecutorService and the ForkJoin simplified a developer’s** life a little bit, but it was still worth remembering how to create a specific executor, how to run it, and so on. Java 8 introduced a way of accomplishing parallelism in a functional style.
* The API allows us to create parallel streams, which perform operations in a parallel mode. When the source of a stream is a Collection or an array, it can be achieved with the help of the parallelStream() method:

|  |
| --- |
| Stream<Product> streamOfCollection = productList.parallelStream();  boolean isParallel = streamOfCollection.isParallel();  boolean bigPrice = streamOfCollection  .map(product -> product.getPrice() \* 12)  .anyMatch(price -> price > 200); |

If the source of a stream is something other than a Collection or an array, the parallel() method should be used:

|  |
| --- |
| IntStream intStreamParallel = IntStream.range(1, 150).parallel();  boolean isParallel = intStreamParallel.isParallel(); |

**Conclusion:**

* The Stream API is a powerful, but simple to understand set of tools for processing the sequence of elements. When used properly, it allows us to reduce a huge amount of boilerplate code, create more readable programs, and improve an app’s productivity.
* In most of the code samples shown in this article, we left the streams unconsumed (we didn’t apply the close() method or a terminal operation). In a real app, don’t leave an instantiated stream unconsumed, as that will lead to memory leaks.
* The complete code samples that accompany this article are available.

**Stream Methods:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **S No** | **Method name** | **Method description/situation** | **Input** | **Output** |  | **Example Program** |
| 1 | **empty**()  Stream<String> streamEmpty = Stream.empty(); | We should use the **empty()** method in case of the creation of an empty stream  We often use the empty() method upon creation to avoid returning null for streams with no element | NA | Stream<T> |  | public Stream<String> streamOf(List<String> list) {  return list == null ||  list.isEmpty() ?  Stream.empty() : list.stream();  } |
| **2** |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

**Stream of Primitives:**

* Java 8 offers the possibility to create streams out of three primitive types: **int, long and double.**
* As **Stream<T> is a generic interface**, and there is no way to use primitives as a type parameter with generics, three new special interfaces were created: **IntStream, LongStream, DoubleStream**.

|  |
| --- |
| IntStream intStream = IntStream.**range**(1, 3);  LongStream longStream = LongStream.**rangeClosed**(1, 3); |

* The **range(int startInclusive, int endExclusive**) method creates an ordered stream from the first parameter to the second parameter. It increments the value of subsequent elements with the step equal to 1. The result doesn’t include the last parameter, it is just an upper bound of the sequence.
* The rangeClosed(int startInclusive, int endInclusive) method does the same thing with only one difference, the second element is included. We can use these two methods to generate any of the three types of streams of primitives.
* Since Java 8, the Random([Random (Java SE 17 & JDK 17) (oracle.com)](https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Random.html))class provides a wide range of methods for generating streams of primitives.
* For example, the following code creates a DoubleStream, which has three elements:

|  |
| --- |
| Random random = new Random();  DoubleStream doubleStream = random.doubles(3); |

Using the new interfaces alleviates unnecessary auto-boxing, which allows for increased productivity:

**Primitive Streams:**

1. String.chars() --- return IntStream
2. boxed() --- we can apply to any primitive streams to convert wrapper streams.
3. codePoints – it is similar like chars() but it give more feature like java.lang.Character
4. mapToObj
5. Stream.of
6. mapToInt
7. Stream.of()
8. Diff betweem map and flatmap

**Lombok :**

* @Builder
* @SuperBuilder – if we have IS-A relationship and if you want to get all the properties in the builder design pattern then you have ti use @SuperBuilde rin both child and parent class from v 1.87.2 version onword

**Wrapper Stream:**

Collectors Utility Classes:

Programs:

1. Convert character array to string in java

**PermGem vs MetaSpace:** [Java Memory: PermGen vs MetaSpace | LinkedIn](https://www.linkedin.com/pulse/java-memory-permgen-vs-metaspace-incus-data-pty-ltd-drxbe/)