Lambada Expression

Java 8 Lambadas

REVATURE

- *Java 8 is first released in 2014 and introduced lot of new features.*
- *Lambda expressions are the most significant change in java 8 version.*
- *As of java 7, Oracle/Sun people have given importance to the object-oriented programming languages but in java 8, they have introduced functional programming to compete with other programming languages such as Scala, C# etc.*

What is Lambda Expression?

REVATURE

- *Any function which is having no name is called as Lambda expression.*
- *Which is also called as anonymous function.*

Rules:

1. *function should not have access modifier*
2. *Should not have any return type (even void also not allowed)*
3. *Should not have name for function*
4. *Should use arrow symbol "->"*

Before JAVA 8:

```
public void print() {
        System.out.println("Hello World");
 }
```

After JAVA 8:

```
() -> {
        System.out.println("Hello World");
};
```

Or

```
() -> System.out.println("Hello World");
```

We have removed-

- *Function's access modifier (public),*
- *return type(void) and*
- *method name (println) in the lambda expression and*
- *added -> symbol*

NOTE: If method body has only statement then curly braces are optional

Optional Class

*java.util.*Optional Class

REVATURE

- *Optional class was added in Java 8 release under java.util package.*
- *Java 8 Optional is used to represent Optional object or empty value instead of null reference.*
- *This will help to avoid null pointer exception which occurs when we try to perform some operation on null reference object.*
- *It works as container or wrapper for actual value which may or may not have null.*
- *Optional class has private constructor so we can not create object using new keyword.*

Why java.util.Optional?

REVATURE

- *While developing any application or program every developer has faced* `NullPointerException`.
- *Consequently, in order to avoid that null pointer, we need to add null checks which could require nesting of if statements.*
- *As a result, leading to ugly and unreadable code.*
- *Let's look at the following example for better insight:*
- *This code could lead to null pointer if person or person's account does not exist-*

```
Person person=new Person()
Double balance = person.getAccount().getBalance();

//Traditional method of handling null checks
Double balance = 0.0;
if(person != null ){
     Account account = person.getAccount();
     if(account != null){
          balance = account.getBalance();
     }
}
```

Methods in Java 8 Optional class

REVATURE

Create instance	Check value	Fetch value	Operational methods
`empty()`	`isPresent()`	`get()`	`ifPresent(Consumer consumer)`
`of(T value)`	`filter(Predicate predicate)`	`orElse(T other)`	`map(Function mapper)`
`ofNullable(T value)`		`orElseGet(Supplier other)`	`flatMap(Function mapper)`
		`orElseThrow(Supplier exception)`	

REVATURE

Streams API

java.util.stream.* API

REVATURE

- *Java 8 introduced a new API which is called as Stream.*
- *This API supports processing the large data sets in a sequential and parallel model.*

- *Java 8 stream API consumes all available core effectively and improves the performance significantly.*
- Java Stream API is developed mainly for Big Data application to process the large data sets effectively using all the hardware.

Example:
Find the error in the log file.
First converting the log content into stream.

```
Stream<String> = Stream.of(responseLogContent.split("\n"));
Optional<String> firstErrorOrException = stream.filter(line -> line.indexOf("ERROR") > -1 ||
line.indexOf("Exception) > -1).findFirst();
```

StreamLogProcessing.java

Stream Vs Collection

REVATURE

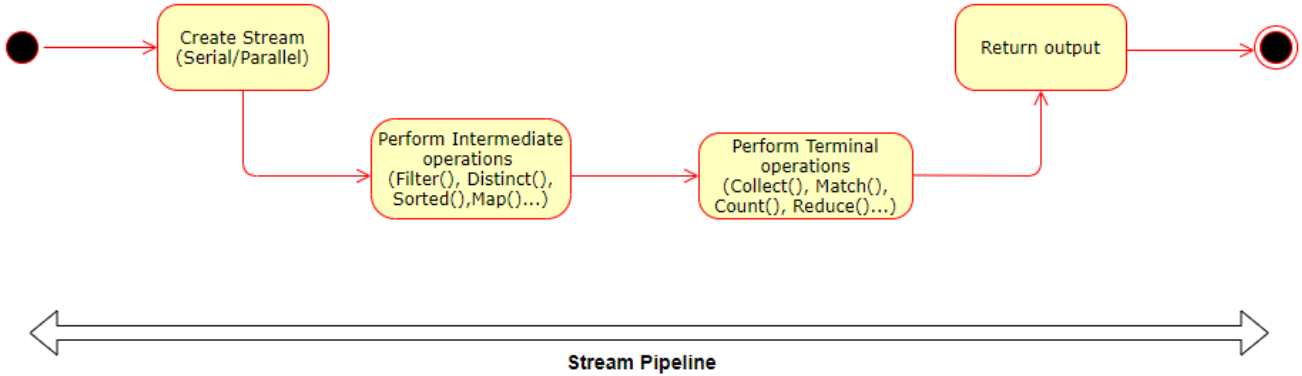
- *Java stream produces pipe-lined data to perform different operations on group of elements.*
- *Streams internally iterate over the data and perform operations, then returns the result of functions performed if any.*
- *Streams are different than Collection.*
- *Streams API is not data structure.*
- *Stream iterates over the data that is used to perform different operations.*
- *Collection is in memory data structure.*
- *Collection requires all data to be loaded in memory.*

Note: java.util.stream is a package and java.util.stream.Stream is an interface

Stream API workflow

- *Java stream involves*
  - *creation of stream,*
  - *performing intermediate operations followed by*
  - *terminal operations on the stream.*

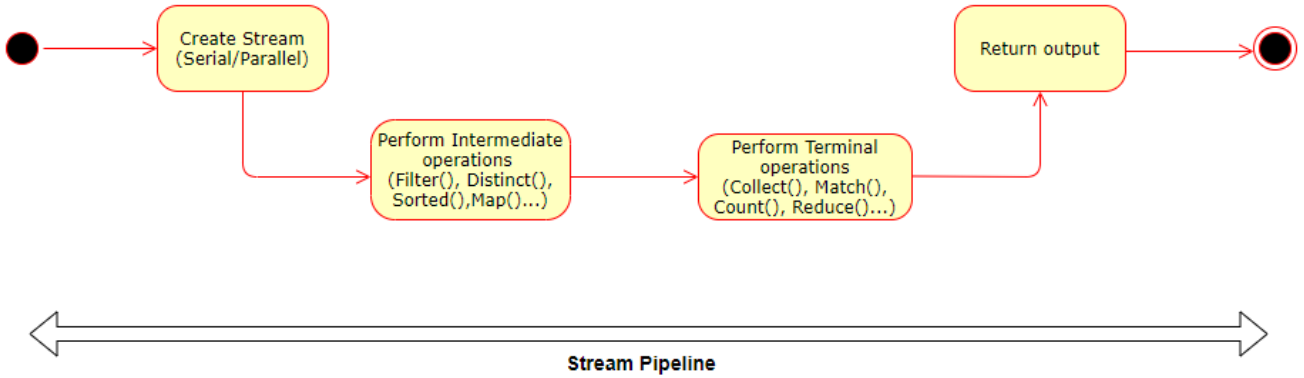
This process or this flow is known as the Stream pipeline. A general high-level stream workflow looks as below



Stream API workflow

REVATURE

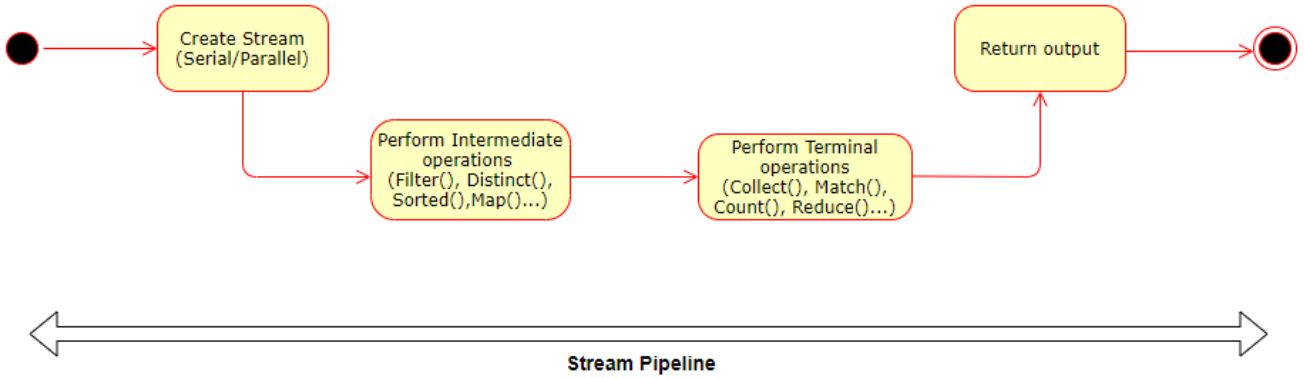
- *Java stream involves*
  - *creation of stream,*
  - *performing intermediate operations followed by*
  - *terminal operations on the stream.*
- *This process or this flow is known as the Stream pipeline. A general high-level stream workflow looks as below*
- *As we can see, in the first step a stream is created with the given input. This input can be any type of data ranging from arrays, collections also primitives. Then this stream is passed on to the intermediate operations. These in turn produce the output streams and give them to the terminal operations. As a result of which output is generated form the terminal operations.*



Stream API workflow

REVATURE

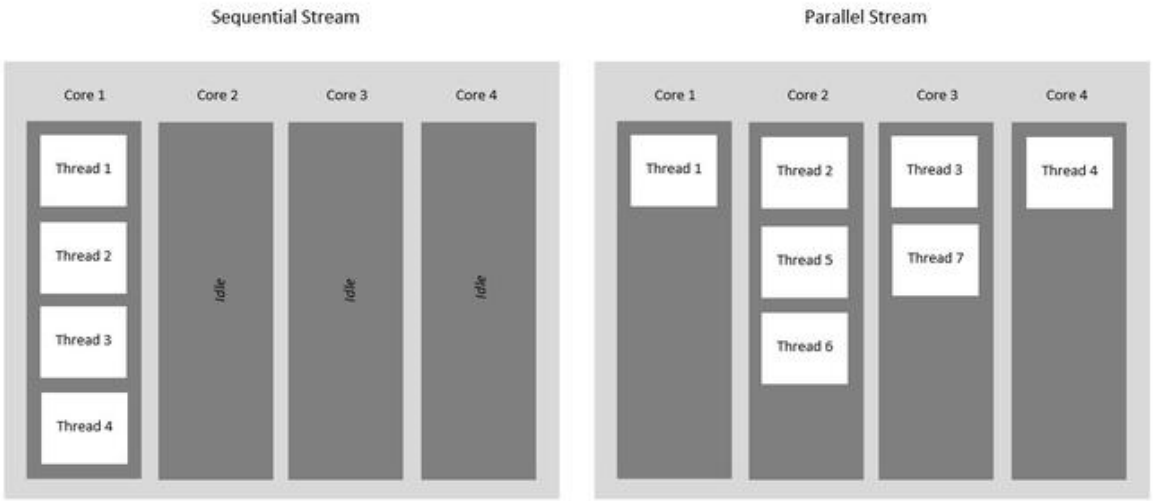
- *Java stream involves*
  - *creation of stream,*
  - *performing intermediate operations followed by*
  - *terminal operations on the stream.*
- *This process or this flow is known as the Stream pipeline. A general high-level stream workflow looks as below*
- *As we can see, in the first step a stream is created with the given input. This input can be any type of data ranging from arrays, collections also primitives. Then this stream is passed on to the intermediate operations. These in turn produce the output streams and give them to the terminal operations. As a result of which output is generated form the terminal operations.*



Parallel vs Sequential Stream in Java

REVATURE

- *A stream in Java is a sequence of objects which operates on a data source such as an array or a collection and supports various methods.*
- *It was introduced in Java 8's java.util.stream package.*
- *Stream supports many aggregate operations like filter, map, limit, reduce, find, and match to customize the original data into a different form according to the need of the programmer.*
- *The operations performed on a stream do not modify its source hence a new stream is created according to the operation applied to it. The new data is a transformed copy of the original form.*



Sequential Stream

- *Sequential Streams are non-parallel streams that use a single thread to process the pipelining. Any stream operation without explicitly specified as parallel is treated as a sequential stream. Sequential stream's objects are pipelined in a single stream on the same processing system hence it never takes the advantage of the multi-core system even though the underlying system supports parallel execution. Sequential stream performs operation one by one.*
- *stream() method returns a sequential stream in Java.*

```
import java.io.*;
import java.util.*;
import java.util.stream.*;
class SequentialStreamDemo {
    public static void main(String[] args)
    {
      // create a list
        List<String> list = Arrays.asList( "Hello ",
                "G", "E", "E", "K", "S!");
      // we are using stream() method for sequential stream
      // Iterate and print each element of the stream
           list.stream().forEach(System.out::print);
    }
}
```

Output:

```
Hello GEEKS!
```

In *this example the list.stream() works in sequence on a single thread with the print() operation and in the output of the preceding program, the content of the list is printed in an ordered sequence as this is a sequential stream.*

Parallel Stream

REVATURE

- *It is a very useful feature of Java to use parallel processing, even if the whole program may not be parallelized. Parallel stream leverage multi-core processors, which increases its performance. Using parallel streams, our code gets divide into multiple streams which can be executed parallelly on separate cores of the system and the final result is shown as the combination of all the individual core's outcomes. It is always not necessary that the whole program be parallelized, but at least some parts should be parallelized which handles the stream. The order of execution is not under our control and can give us unpredictably unordered results and like any other parallel programming, they are complex and error-prone.*
- *The Java stream library provides a couple of ways to do it. easily, and in a reliable manner.*
    - ❑ *One of the simple ways to obtain a parallel stream is by invoking the parallelStream() method of Collection interface.*
    - ❑ *Another way is to invoke the parallel() method of BaseStream interface on a sequential stream.*
- *It is important to ensure that the result of the parallel stream is the same as is obtained through the sequential stream, so the parallel streams must be stateless, non-interfering, and associative.*

Parallel Stream Example

```
import java.io.*;
import java.util.*;
import java.util.stream.*;
class ParallelStreamExample {
    public static void main(String[] args)
    {
      // create a list
       List<String> list = Arrays.asList("Hello ",
                          "G", "E", "E", "K", "S!");
      // using parallelStream()
      // method for parallel stream
       list.parallelStream().forEach(System.out::print);
    }
}
```

Output:

```
ES!KGEHello
```

Here *we can see the order is not maintained as the list.parallelStream() works parallelly on multiple threads. If we run this code multiple times then we can also see that each time we are getting a different order as output but this parallel stream boosts the performance so the situation where the order is not important is the best technique to use.*

Parallel Stream Ordered

```
import java.io.*;
import java.util.*;
import java.util.stream.*;
class ParallelStreamWithOrderedIteration {
    public static void main(String[] args)
    {
       // create a list
        List<String> list
            = Arrays.asList("Hello ","G", "E", "E", "K", "S!");
        // using parallelStream() method for parallel stream
        list.parallelStream().forEachOrdered(System.out::print);
    }
}
```

Output:
Hello GEEKS!

If *we want to make each element in the parallel stream to be ordered, we can use the forEachOrdered() method, instead of the forEach() method.*

NOTE: We can always switch between parallel and sequential very easily according to our requirements. If we want to change the parallel stream as sequential, then we should use the sequential() method specified by BaseStream Interface.

Differences between Sequential Stream and Parallel Stream

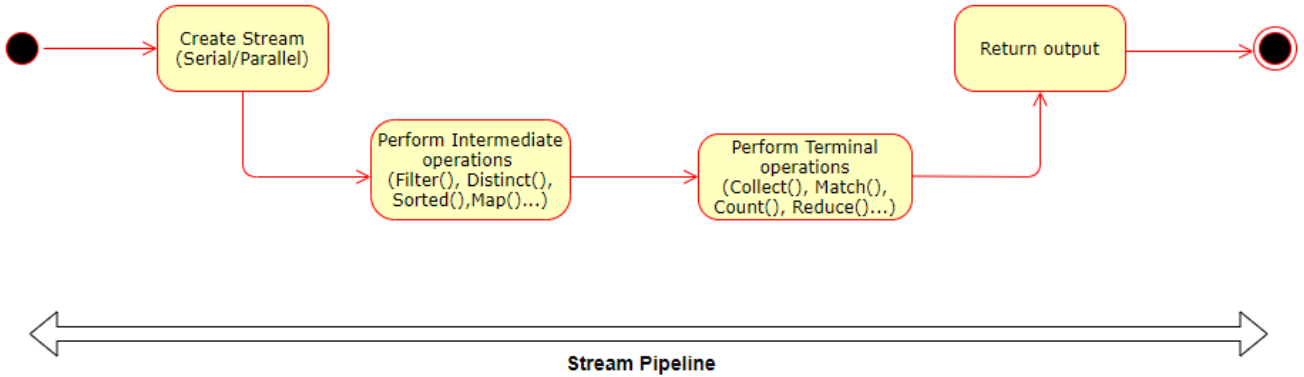
REVATURE

Sequential Stream	Parallel Stream
Runs on a single-core of the computer	Utilize the multiple cores of the computer.
Performance is poor	The performance is high.
Order is maintained	Doesn’t care about the order
Only a single iteration at a time just like the for-loop.	Operates multiple iterations simultaneously in different available cores.
Each iteration waits for currently running one to finish,	Waits only if no cores are free or available at a given time,
More reliable and less error	Less reliable and error-prone.
Platform independent	Platform dependent

Stream API Intermediate Operations

REVATURE

- *Every stream code or pipeline must-have a map() or filter() methods. These methods are called as Intermediate Functions because these methods again create a temporary Stream to hold the intermediate output.*
    1. *filter()*
    2. *map()*
    3. *flatMap()*
    4. *distinct()*
    5. *sorted()*
    6. *peek()*
    7. *limit()*
    8. *skip()*



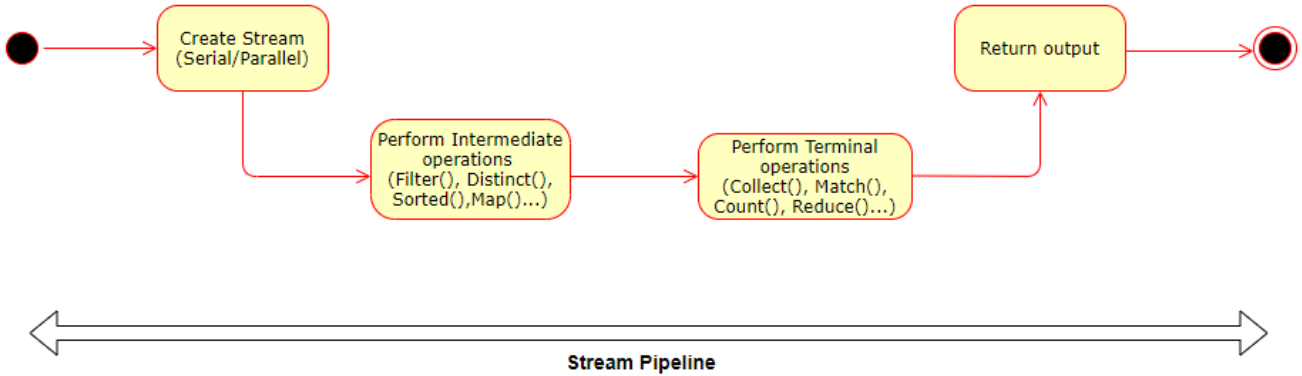
Stream API Terminal Operations

REVATURE

- *After calling the intermediate methods, to collect the final output stream api has another set of methods. Those are called Terminal Operations. These methods produce output such as Collection, List, Set, or Optional values.*

1. *toArray()*
2. *collect()*
3. *count()*
4. *reduce()*
5. *forEach()*
6. *forEachOrdered()*
7. *min()*
8. *max()*
9. *anyMatch()*
10. *allMatch()*
11. *noneMatch()*
12. *findAny()*
13. *findFirst()*

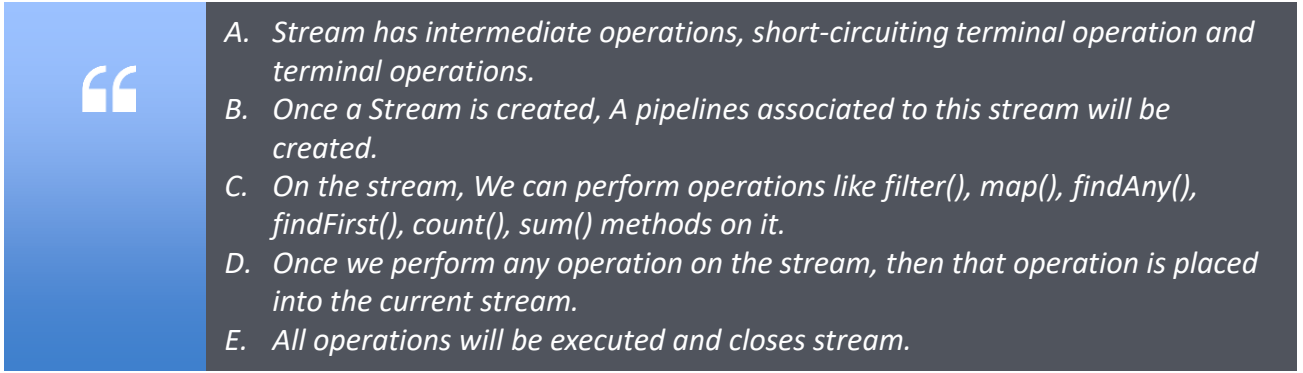
- *In contrast to the intermediate operations described above, the terminal operations do not return stream as output.*
  Terminal operation can produce result as primitive or collection or any Object or may not produce any output.
- *The intermediate operations always precede a terminal operation.*
- ***Although there can be multiple intermediate operations, but they must be followed by one and only one terminal operation.***



Java 8 IllegalStateException “Stream has already been operated upon or closed” Exception (Stream reuse)

REVATURE

- *A stream should be operated on (invoking an intermediate or terminal stream operation) only once.*
- *Taking a deeper look at what happens after stream creation.*



filter(), map() methods are **intermediate operations**.
findAny(), findFirst() are **short-circuiting terminal operations** *as it returns the first element and stops further processing.*
count() and sum() are **terminal operations.**

Stream Already In Operated Mode Example

REVATURE

```
String conent = "line 1 \n line 2 \\n line 3 \\n line 4 \\n line 5";
//stream is created with pipeline
Stream mainStream = Stream.of(conent.split("\n"));

//Now filter operation is placed in the current pipeline
mainStream.filter(line -> line.indexOf("line") > -1);

//Now again calling filter method on mainStream.
//That will try to put the filter operation in the pipeline.
//But the pipeline is currently in operated mode with above filter.
//So, it is not possible to create another pipeline on the same stream.
mainStream.filter(line -> line.indexOf("1") > -1);
```

Output (Runtime exception):

```
Exception in thread "main" java.lang.IllegalStateException: stream has already been operated upon
or closed
 at java.base/java.util.stream.AbstractPipeline.(AbstractPipeline.java:203)
```

Stream Already In Operated Mode Example

REVATURE

```
// Creating Stream from String.
String conent = "line 1 \n line 2 \\n line 3 \\n line 4 \\n line 5";
Stream mainStream = Stream.of(conent.split("\n"));

// Operation on mainStream
Optional firstValue = mainStream.findFirst();
System.out.println("firstValue : " + firstValue.get());

// Operation on mainStream
Optional anyValue = mainStream.findAny();
System.out.println("anyValue : " + anyValue.get());
```

First when calling findFirst() then it returns value. But next invoking findAny(), at this time stream or pipeline is already closed.
So it can not perform operation on the same stream.
Here firstValue is printed but while getting value from findAny() operation it has failed to get the stream because stream is already closed.

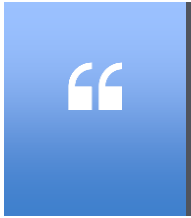
Output (Runtime exception):

```
firstValue : line 1
Exception in thread "main" java.lang.IllegalStateException: stream has already been operated upon or closed
 at java.base/java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:229)
```

How To Solve This Problem?

REVATURE

We have many scenarios where the same stream is required to perform many operations.
To solve this issue, We need to create two-stream instances for these two separate operations.



- *A Stream should be operated on (invoking an intermediate or terminal stream operation) only once. A Stream implementation may throw IllegalStateException if it detects that the Stream is being reused.*
- Instead of creating two stream objects separately, We can use built in API Functional Interface Supplier<T> which is super handy in usage.

Solution

REVATURE

```
Supplier> supplier = () -> Stream.of(conent.split("\n"));

// findFirst Operation
Optional firstValue = supplier.get().findFirst();
System.out.println("firstValue : " + firstValue.get());

// findAny Operation
Optional anyValue =supplier.get().findAny();
System.out.println("anyValue : " + anyValue.get());
```

Output:

```
firstValue : line 1
anyValue : line 5
```

We need to create a Supplier object with type of Stream<String>. Supplier has a abstract method that is get() which returns Stream<String>. We will use Lambda Expression for Supplier Functional Interface with () zero arguments.
So, Whenever we ***call supplier.get() method then a new fresh Stream<String> object will be created*** *and can perform any stream operation on it.*

Appendix

https://www.benchresources.net/java8-functional-interface/
https://stacktraceguru.com/java-8-functional-interface/
https://www.benchresources.net/java8-default-and-static-methods/

https://www.benchresources.net/java8-lambda-expression/
https://stacktraceguru.com/java-lambda-expression/

https://www.benchresources.net/java-8-stream-api-introduction/
https://www.benchresources.net/java-8-difference-between-map-and-flatmap-in-stream-api/
https://stacktraceguru.com/overview-java-8-stream-api/

https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html#overview

https://stacktraceguru.com/java-8-optional/
https://stacktraceguru.com/method-reference-java-8/