**1. Overview**

In this in-depth tutorial, we'll go through the practical usage of Java 8 Streams from creation to parallel execution.

To understand this material, readers need to have a basic knowledge of Java 8 (lambda expressions, *Optional,*method references) and of the Stream API. If you aren’t familiar with these topics, please take a look at our previous articles – [New Features in Java 8](https://www.baeldung.com/java-8-new-features) and [Introduction to Java 8 Streams](https://www.baeldung.com/java-8-streams-introduction).

**Further reading:**

[**Lambda Expressions and Functional Interfaces: Tips and Best Practices**](https://www.baeldung.com/java-8-lambda-expressions-tips)

Tips and best practices on using Java 8 lambdas and functional interfaces.

[**Read more**](https://www.baeldung.com/java-8-lambda-expressions-tips) →

[**Guide to Java 8's Collectors**](https://www.baeldung.com/java-8-collectors)

The article discusses Java 8 Collectors, showing examples of built-in collectors, as well as showing how to build custom collector.

[**Read more**](https://www.baeldung.com/java-8-collectors) →

**2. Stream Creation**

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.

**2.1. Empty Stream**

The ***empty()*** method should be used in case of a creation of an empty stream:

|  |  |
| --- | --- |
| 1 | Stream<String> streamEmpty = Stream.empty(); |

Its often the case that the *empty()*method is used upon creation to avoid returning *null* for streams with no element:

|  |  |
| --- | --- |
| 1  2  3 | public Stream<String> streamOf(List<String> list) {      return list == null || list.isEmpty() ? Stream.empty() : list.stream();  } |

**2.2. Stream of *Collection***

Stream can also be created of any type of *Collection*(*Collection, List, Set*):

|  |  |
| --- | --- |
| 1  2 | Collection<String> collection = Arrays.asList("a", "b", "c");  Stream<String> streamOfCollection = collection.stream(); |

**2.3. Stream of Array**

Array can also be a source of a Stream:

|  |  |
| --- | --- |
| 1 | Stream<String> streamOfArray = Stream.of("a", "b", "c"); |

They can also be created out of an existing array or of a part of an array:

|  |  |
| --- | --- |
| 1  2  3 | String[] arr = new String[]{"a", "b", "c"};  Stream<String> streamOfArrayFull = Arrays.stream(arr);  Stream<String> streamOfArrayPart = Arrays.stream(arr, 1, 3); |

**2.4.*Stream.builder()***

**When builder is used the desired type should be additionally specified in the right part of the statement,** otherwise the *build()* method will create an instance of the *Stream<Object>:*

|  |  |
| --- | --- |
| 1  2 | Stream<String> streamBuilder =    Stream.<String>builder().add("a").add("b").add("c").build(); |

**2.5.*Stream.generate()***

The ***generate()*** method accepts a *Supplier<T>*for element generation. As the resulting stream is infinite, developer should specify the desired size or the *generate()* method will work until it reaches the memory limit:

|  |  |
| --- | --- |
| 1  2 | Stream<String> streamGenerated =    Stream.generate(() -> "element").limit(10); |

The code above creates a sequence of ten strings with the value – *“element”*.

**2.6.*Stream.iterate()***

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

|  |  |
| --- | --- |
| 1 | Stream<Integer> streamIterated = Stream.iterate(40, n -> n + 2).limit(20); |

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

**2.7. Stream of Primitives**

Java 8 offers a 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.***

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

|  |  |
| --- | --- |
| 1  2 | 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 with only one difference – the second element is included. These two methods can be used to generate any of the three types of streams of primitives.

Since Java 8 the [*Random*](https://docs.oracle.com/javase/8/docs/api/java/util/Random.html) class provides a wide range of methods for generation streams of primitives. For example, the following code creates a *DoubleStream,*which has three elements:

|  |  |
| --- | --- |
| 1  2 | Random random = new Random();  DoubleStream doubleStream = random.doubles(3); |

**2.8. Stream of *String***

*String* can also be used as a source for creating a stream.

With the help of the *chars()* method of the *String* class. Since there is no interface *CharStream*in JDK, the *IntStream* is used to represent a stream of chars instead.

|  |  |
| --- | --- |
| 1 | IntStream streamOfChars = "abc".chars(); |

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

|  |  |
| --- | --- |
| 1  2 | Stream<String> streamOfString =    Pattern.compile(", ").splitAsStream("a, b, c"); |

**2.9. Stream of File**

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

|  |  |
| --- | --- |
| 1  2  3  4 | 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.

**3. Referencing a Stream**

It is possible to instantiate a stream and to have an accessible reference to it as long as only intermediate operations were called. Executing a terminal operation makes a stream inaccessible*.*

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

|  |  |
| --- | --- |
| 1  2  3 | Stream<String> stream =    Stream.of("a", "b", "c").filter(element -> element.contains("b"));  Optional<String> anyElement = stream.findAny(); |

But an attempt to reuse the same reference after calling the terminal operation will trigger the *IllegalStateException:*

|  |  |
| --- | --- |
| 1 | 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.**

This kind of behavior is logical because streams were designed to provide an ability to apply a finite sequence of operations to the source of elements in a functional style, but not to store elements.

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

|  |  |
| --- | --- |
| 1  2  3  4  5 | 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(); |

**4. Stream Pipeline**

To perform a sequence of operations over the elements of the data source and aggregate their results, three parts are needed – 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:

|  |  |
| --- | --- |
| 1  2 | Stream<String> onceModifiedStream =    Stream.of("abcd", "bbcd", "cbcd").skip(1); |

If more than one modification is needed, intermediate operations can be chained. Assume that we also need to substitute every element of current *Stream<String>* with a sub-string of first few chars. This will be done by chaining the *skip()* and the *map()* methods:

|  |  |
| --- | --- |
| 1  2 | Stream<String> twiceModifiedStream =    stream.skip(1).map(element -> element.substring(0, 3)); |

As you can see, the *map()* method takes a lambda expression as a parameter. If you want to learn more about lambdas take a look at our tutorial [Lambda Expressions and Functional Interfaces: Tips and Best Practices](https://www.baeldung.com/java-8-lambda-expressions-tips).

A stream by itself is worthless, the real thing a user is interested in is a result of the terminal operation, which can be a value of some type or an action applied to every element of the stream. **Only one terminal operation can be used per stream.**

The right and most convenient way to use streams are by a **stream pipeline, which is a chain of stream source, intermediate operations, and a terminal operation.**For example:

|  |  |
| --- | --- |
| 1  2  3 | List<String> list = Arrays.asList("abc1", "abc2", "abc3");  long size = list.stream().skip(1)    .map(element -> element.substring(0, 3)).sorted().count(); |

**5. Lazy Invocation**

**Intermediate operations are lazy.** This means that **they will be invoked only if it is necessary for the terminal operation execution.**

To demonstrate this, imagine that we have method*wasCalled(),*which increments an inner counter every time it was called:

|  |  |
| --- | --- |
| 1  2  3  4  5 | private long counter;    private void wasCalled() {      counter++;  } |

Let's call method was*Called()* from operation *filter()*:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | 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 method *filter()* will be called three times and the value of the *counter*variable will be 3. But running this code doesn't change *counter*at all, it is still zero, so, the *filter()* method wasn't called even 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 an ability to track an order of method calls with a help of logging:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | 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(); |

Resulting log shows that the *filter()* method was called twice and the *map()* method just once. It is so because the pipeline executes vertically. In our example the first element of the stream didn't satisfy filter's predicate, then the *filter()* method was invoked for the second element, which passed the filter. Without calling the *filter()* for third element we went down through pipeline to the *map()* method.

The *findFirst()* operation satisfies by just one element. So, in this particular example the lazy invocation allowed to avoid two method calls – one for the *filter()* and one for the *map().*

**6. 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:**

|  |  |
| --- | --- |
| 1  2  3  4 | 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 the *map()* method of the stream was called three times. But the value of the *size* is one. So, resulting stream has just one element and we executed the expensive *map()* operations for no reason twice out of three times.

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

|  |  |
| --- | --- |
| 1  2  3  4 | long size = list.stream().skip(2).map(element -> {      wasCalled();      return element.substring(0, 3);  }).count(); |

This brings us up to the rule: **intermediate operations which reduce the size of the stream should be placed before operations which are applying to each element.** So, keep such methods as s*kip(), filter(), distinct()*at the top of your stream pipeline.

**7. Stream Reduction**

The API has many terminal operations which aggregate a stream to a type or to a primitive, for example, *count(), max(), min(), sum(),*but these operations work according to the predefined implementation. And what **if a developer needs to customize a Stream's reduction mechanism?** There are two methods which allow to do this – the ***reduce()***and the ***collect()*** methods.

**7.1. The *reduce()* Method**

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

**identity –**the initial value for an accumulator or a default value if a stream is empty and there is nothing to accumulate;

**accumulator –**a function which specifies a logic of aggregation of elements. As accumulator creates a new value for every step of reducing, the quantity of new values equals to the stream's size and only the last value is useful. This is not very good for the performance.

**combiner –**a function which aggregates results of the accumulator. Combiner is called only in a parallel mode to reduce results of accumulators from different threads.

So, let's look at these three methods in action:

|  |  |
| --- | --- |
| 1  2 | OptionalInt reduced =    IntStream.range(1, 4).reduce((a, b) -> a + b); |

*reduced*= 6 (1 + 2 + 3)

|  |  |
| --- | --- |
| 1  2 | int reducedTwoParams =    IntStream.range(1, 4).reduce(10, (a, b) -> a + b); |

*reducedTwoParams* = 16 (10 + 1 + 2 + 3)

|  |  |
| --- | --- |
| 1  2  3  4  5 | int reducedParams = Stream.of(1, 2, 3)    .reduce(10, (a, b) -> a + b, (a, b) -> {       log.info("combiner was called");       return a + b;    }); |

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:

|  |  |
| --- | --- |
| 1  2  3  4  5 | int reducedParallel = Arrays.asList(1, 2, 3).parallelStream()      .reduce(10, (a, b) -> a + b, (a, b) -> {         log.info("combiner was called");         return a + b;      }); |

The result here is different (36) and the combiner was called twice. Here the reduction works by the following algorithm: accumulator ran three times by adding every element of the stream to *identity* to every element of the stream. 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).

**7.2. The *collect()* Method**

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.

In this section we will use the following *List* as a source for all streams:

|  |  |
| --- | --- |
| 1  2  3 | 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*):**

|  |  |
| --- | --- |
| 1  2 | List<String> collectorCollection =    productList.stream().map(Product::getName).collect(Collectors.toList()); |

**Reducing to *String*:**

|  |  |
| --- | --- |
| 1  2 | String listToString = productList.stream().map(Product::getName)    .collect(Collectors.joining(", ", "[", "]")); |

The *joiner()*method can have from one to three parameters (delimiter, prefix, suffix). The handiest thing about using *joiner()* – 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:**

|  |  |
| --- | --- |
| 1  2 | double averagePrice = productList.stream()    .collect(Collectors.averagingInt(Product::getPrice)); |

**Processing the sum of all numeric elements of the stream:**

|  |  |
| --- | --- |
| 1  2 | int summingPrice = productList.stream()    .collect(Collectors.summingInt(Product::getPrice)); |

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

**Collecting statistical information about stream’s elements:**

|  |  |
| --- | --- |
| 1  2 | IntSummaryStatistics statistics = productList.stream()    .collect(Collectors.summarizingInt(Product::getPrice)); |

By using the resulting instance of type *IntSummaryStatistics* developer can create a statistical report by applying *toString()* method. The result will be a *String* common to this one *“IntSummaryStatistics{count=5, sum=86, min=13, average=17,200000, max=23}”.*

It is also easy to extract from this object separate values for *count, sum, min, average* by applying methods *getCount(), getSum(), getMin(), getAverage(), getMax().*All these values can be extracted from a single pipeline.

**Grouping of stream’s elements according to the specified function:**

|  |  |
| --- | --- |
| 1  2 | Map<Integer, List<Product>> collectorMapOfLists = productList.stream()    .collect(Collectors.groupingBy(Product::getPrice)); |

In the example above the stream was reduced to the *Map* which groups all products by their price.

**Dividing stream’s elements into groups according to some predicate:**

|  |  |
| --- | --- |
| 1  2 | Map<Boolean, List<Product>> mapPartioned = productList.stream()    .collect(Collectors.partitioningBy(element -> element.getPrice() > 15)); |

**Pushing the collector to perform additional transformation:**

|  |  |
| --- | --- |
| 1  2  3 | Set<Product> unmodifiableSet = productList.stream()    .collect(Collectors.collectingAndThen(Collectors.toSet(),    Collections::unmodifiableSet)); |

In this particular case, the collector has converted a stream to a *Set* and then created the unmodifiable *Set* out of it.

**Custom collector:**

If for some reason, a custom collector should be created, the most easier and the less verbose way of doing so – is to use the method *of()* of the type *Collector.*

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | Collector<Product, ?, LinkedList<Product>> toLinkedList =    Collector.of(LinkedList::new, LinkedList::add,      (first, second) -> {         first.addAll(second);         return first;      });    LinkedList<Product> linkedListOfPersons =    productList.stream().collect(toLinkedList); |

In this example, an instance of the *Collector* got reduced to the *LinkedList<Persone>.*

**Parallel Streams**

Before Java 8, parallelization was complex. Emerging of the [*ExecutorService*](https://www.baeldung.com/java-executor-service-tutorial) and the [*ForkJoin*](https://www.baeldung.com/java-fork-join)simplified developer’s life a little bit, but they still should keep in mind 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 creating 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:

|  |  |
| --- | --- |
| 1  2  3  4  5 | 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 stream is something different than a *Collection*or an *array*, the ***parallel()*** method should be used:

|  |  |
| --- | --- |
| 1  2 | IntStream intStreamParallel = IntStream.range(1, 150).parallel();  boolean isParallel = intStreamParallel.isParallel(); |

Under the hood, Stream API automatically uses the *ForkJoin*framework to execute operations in parallel. By default, the common thread pool will be used and there is no way (at least for now) to assign some custom thread pool to it. [This can be overcome by using a custom set of parallel collectors.](https://github.com/pivovarit/parallel-collectors)

When using streams in parallel mode, avoid blocking operations and use parallel mode when tasks need the similar amount of time to execute (if one task lasts much longer than the other, it can slow down the complete app’s workflow).

The stream in parallel mode can be converted back to the sequential mode by using the *sequential()* method:

|  |  |
| --- | --- |
| 1  2 | IntStream intStreamSequential = intStreamParallel.sequential();  boolean isParallel = intStreamSequential.isParallel(); |

**Conclusions**

The Stream API is a powerful but simple to understand set of tools for processing sequence of elements. It allows us to reduce a huge amount of boilerplate code, create more readable programs and improve app’s productivity when used properly.

In most of the code samples shown in this article streams were left unconsumed (we didn't apply the *close()* method or a terminal operation). In a real app, **don't leave an instantiated streams unconsumed as that will lead to memory leaks.**

How to print elements of a Stream in Java 8

Introduced in Java 8, [the Stream API](https://www.geeksforgeeks.org/stream-in-java/) is used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.  
The features of Java stream are –

* A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels.
* Streams don’t change the original data structure, they only provide the result as per the pipelined methods.
* Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.

There are 3 ways to print the elements of a Stream in Java:

1. forEach()
2. println() with collect()
3. peek()

Below are the three ways to print the Stream in detail:

1. [**Stream forEach(Consumer action)**](https://www.geeksforgeeks.org/stream-foreach-method-java-examples/): This method performs an action for each element of the stream. Stream forEach(Consumer action) is a ***terminal operation*** i.e, it may traverse the stream to produce a result or a side-effect.

**Syntax :**

**void forEach(Consumer<? super T> action)**

Where, Consumer is a functional interface

and T is the type of stream elements.

Below is how to print elements of Stream using forEach() method:

**Program 1:**

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // Java code to print the elements of Stream    import java.util.stream.\*;    class GFG {      public static void main(String[] args)      {            // Get the stream          Stream<String> stream = Stream.of("Geeks", "For",                                            "Geeks", "A",                                            "Computer", "Portal");            // Print the stream          stream.forEach(s -> System.out.println(s));      }  } |

**Output:**

Geeks

For

Geeks

A

Computer

Portal

**Program 2:** Using Short hand lambda expression

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // Java code to print the elements of Stream    import java.util.stream.\*;    class GFG {      public static void main(String[] args)      {            // Get the stream          Stream<String> stream = Stream.of("Geeks", "For",                                            "Geeks", "A",                                            "Computer", "Portal");            // Print the stream          stream.forEach(System.out::println);      }  } |

**Output:**

Geeks

For

Geeks

A

Computer

Portal

**Program 3:** This approach consumes the stream and makes it unavailable for future use. Hence the below code will throw an error since the stream is already consumed.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // Java code to print the elements of Stream    import java.util.stream.\*;    class GFG {      public static void main(String[] args)      {            // Get the stream          Stream<String> stream = Stream.of("Geeks", "For",                                            "Geeks", "A",                                            "Computer", "Portal");            // Print the stream          stream.forEach(s -> System.out.println(s));            // Since the stream has been already consumed          // this will throw exception          try {                // Print the stream              stream.forEach(s -> System.out.println(s));          }            catch (Exception e) {                System.out.println("\nException: " + e);          }      }  } |

**Output:**

Geeks

For

Geeks

A

Computer

Portal

Exception: java.lang.IllegalStateException:

stream has already been operated upon or closed

1. **Using println() with collect():** This method collects the elements of the stream as a collector instance, for example as List. Hence the printing of List can be done easily using println() method.

**Syntax:**

System.out.println(stream.collect(Collectors.toList()));

**Program 1:**

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // Java code to print the elements of Stream    import java.util.stream.\*;    class GFG {      public static void main(String[] args)      {            // Get the stream          Stream<String> stream = Stream.of("Geeks", "For",                                            "Geeks", "A",                                            "Computer", "Portal");            // Print the stream          System.out.println(stream.collect(Collectors.toList()));      }  } |

**Output:**

[Geeks, For, Geeks, A, Computer, Portal]

**Program 2:** This approach also consumes the stream and makes it unavailable for future use. Hence the below code will throw an error since the stream is already consumed.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // Java code to print the elements of Stream    import java.util.stream.\*;    class GFG {      public static void main(String[] args)      {            // Get the stream          Stream<String> stream = Stream.of("Geeks", "For",                                            "Geeks", "A",                                            "Computer", "Portal");            // Print the stream          System.out.println(stream.collect(Collectors.toList()));            // Since the stream has been already consumed          // this will throw exception          try {                // Print the stream              System.out.println(stream.collect(Collectors.toList()));          }            catch (Exception e) {                System.out.println("\nException: " + e);          }      }  } |

**Output:**

[Geeks, For, Geeks, A, Computer, Portal]

Exception: java.lang.IllegalStateException:

stream has already been operated upon or closed

1. [**Stream peek(Consumer action)**](https://www.geeksforgeeks.org/stream-peek-java-examples/): This method returns a stream consisting of the elements of this stream, additionally performing the provided action on each element as elements are consumed from the resulting stream. This is an ***intermediate operation*** i.e, it creates a new stream that, when traversed, contains the elements of the initial stream that match the given predicate.

**Syntax :**

**Stream<T>**

**peek(Consumer<? super T> action)**

Where, Stream is an interface and T is the type of

stream elements. **action** is a [non-interfering](https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html" \l "NonInterference) action

to perform on the elements as they are consumed

from the stream and the function returns the new stream.

**Program 1:**

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // Java code to print the elements of Stream    import java.util.stream.\*;    class GFG {      public static void main(String[] args)      {            // Get the stream          Stream<String> stream = Stream.of("Geeks", "For",                                            "Geeks", "A",                                            "Computer", "Portal");            // Print the stream using peek()          // by providing a terminal operation count()          stream.peek(s -> System.out.println(s)).count();      }  } |

**Output:**

Geeks

For

Geeks

A

Computer

Portal

**Program 2:** This approach **do not consumes** the stream. Hence the below code will not throw any error.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // Java code to print the elements of Stream    import java.util.stream.\*;    class GFG {      public static void main(String[] args)      {            // Get the stream          Stream<String> stream = Stream.of("Geeks", "For",                                            "GeeksForGeeks", "A",                                            "Computer", "Portal");            // Since the stream is not being consumed          // this will not throw any exception            // Print the stream          stream.filter(s -> s.startsWith("G"))              .peek(s -> System.out.println("Filtered value: " + s))              .map(String::toUpperCase)              .peek(s -> System.out.println("Uppercase value :" + s))              .count();      }  } |

**Output:**

Filtered value: Geeks

Uppercase value :GEEKS

Filtered value: GeeksForGeeks

Uppercase value :GEEKSFORGEEKS

Java 8 Stream

[Introduction to Stream](https://www.geeksforgeeks.org/stream-in-java/), [Java Intstream](https://www.geeksforgeeks.org/java-8-stream/" \l "Java Intstream), [Java Longstream](https://www.geeksforgeeks.org/java-8-stream/" \l "Java Longstream), [Java Doublestream](https://www.geeksforgeeks.org/java-8-stream/" \l "Java Doublestream)

* [anyMatch()](https://www.geeksforgeeks.org/stream-anymatch-java-examples/)
* [noneMatch()](https://www.geeksforgeeks.org/stream-nonematch-method-java-examples/)
* [mapToLong()](https://www.geeksforgeeks.org/stream-maptolongtolongfunction-mapper-java/)
* [findAny()](https://www.geeksforgeeks.org/java-stream-findany-with-examples/)
* [forEachOrdered()](https://www.geeksforgeeks.org/stream-foreachordered-method-java-examples/)
* [forEach()](https://www.geeksforgeeks.org/stream-foreach-method-java-examples/)
* [allMatch()](https://www.geeksforgeeks.org/stream-allmatch-java-examples/)
* [filter()](https://www.geeksforgeeks.org/stream-filter-java-examples/)
* [findFirst()](https://www.geeksforgeeks.org/stream-findfirst-java-examples/)
* [flatMapToInt()](https://www.geeksforgeeks.org/stream-flatmaptoint-java-examples/)
* [mapToInt()](https://www.geeksforgeeks.org/stream-maptoint-java-examples/)
* [map()](https://www.geeksforgeeks.org/stream-map-java-examples/)
* [peek()](https://www.geeksforgeeks.org/stream-peek-java-examples/)
* [counting()](https://www.geeksforgeeks.org/java-util-stream-collectors-counting-method-examples/)
* [Iterator()](https://www.geeksforgeeks.org/intstream-iterator-java/)
* [Generate()](https://www.geeksforgeeks.org/intstream-generate-method-java/)
* [Skip()](https://www.geeksforgeeks.org/intstream-skip-java/)
* [SummaryStatistics()](https://www.geeksforgeeks.org/intstream-summarystatistics-java/)
* [Builder()](https://www.geeksforgeeks.org/stream-builder-java-examples/)
* [Empty()](https://www.geeksforgeeks.org/stream-empty-java-examples/)
* [Stream toArray()](https://www.geeksforgeeks.org/stream-toarray-java-examples/)
* [Sum of list with stream filter](https://www.geeksforgeeks.org/sum-list-stream-filter-java/)

**Java Intstream**

* [forEachOrdered()](https://www.geeksforgeeks.org/intstream-foreachordered-method-java/)
* [IntUnaryOperator mapper()](https://www.geeksforgeeks.org/intstream-mapintunaryoperator-mapper-java/)
* [flatMap(IntFunction mapper)](https://www.geeksforgeeks.org/intstream-flatmapintfunction-mapper-java/)
* [forEach()](https://www.geeksforgeeks.org/intstream-foreach-method-java/)
* [sum()](https://www.geeksforgeeks.org/intstream-sum-java/)
* [Range()](https://www.geeksforgeeks.org/intstream-range-java/)
* [Sorted()](https://www.geeksforgeeks.org/intstream-sorted-java/)
* [AnyMatch()](https://www.geeksforgeeks.org/intstream-anymatch-java-examples/)
* [Concat()](https://www.geeksforgeeks.org/intstream-concat-java/)
* [Filter()](https://www.geeksforgeeks.org/intstream-filter-java-examples/)
* [FindAny()](https://www.geeksforgeeks.org/intstream-findany-examples/)
* [AllMatch()](https://www.geeksforgeeks.org/intstream-allmatch-java-examples/)
* [ToDouble()](https://www.geeksforgeeks.org/stream-flatmaptodouble-java-examples/)
* [Max()](https://www.geeksforgeeks.org/intstream-max-in-java-with-examples/)
* [Min()](https://www.geeksforgeeks.org/intstream-min-java-examples/)
* [Count()](https://www.geeksforgeeks.org/intstream-count-in-java-with-examples/)
* [Average()](https://www.geeksforgeeks.org/intstream-average-in-java-with-examples/)
* [Peek()](https://www.geeksforgeeks.org/intstream-peek-in-java-with-examples/)
* [Empty()](https://www.geeksforgeeks.org/intstream-empty-in-java/)
* [Distinct()](https://www.geeksforgeeks.org/intstream-distinct-in-java/)
* [Parallel()](https://www.geeksforgeeks.org/intstream-parallel-java/)
* [Builder()](https://www.geeksforgeeks.org/intstream-builder-java/)
* [Boxed()](https://www.geeksforgeeks.org/intstream-boxed-java/)
* [mapToDouble()](https://www.geeksforgeeks.org/intstream-maptodouble-java/)
* [IntStream asDoubleStream()](https://www.geeksforgeeks.org/intstream-asdoublestream-java/)
* [Limit()](https://www.geeksforgeeks.org/intstream-limit-java/)
* [MapToObj()](https://www.geeksforgeeks.org/intstream-maptoobj-java/)
* [MapToLong()](https://www.geeksforgeeks.org/intstream-maptolong-java/)
* [IntStream asLongStream()](https://www.geeksforgeeks.org/intstream-aslongstream-java/)
* [IntBinaryOperator](https://www.geeksforgeeks.org/intstream-reduceintbinaryoperator-op-java-examples/)
* [IntStream toArray()](https://www.geeksforgeeks.org/intstream-toarray-java-examples/)
* [mapToInt()](https://www.geeksforgeeks.org/longstream-maptoint-java-examples/)
* [Reduce()](https://www.geeksforgeeks.org/intstream-reduceint-identity-intbinaryoperator-op-java-examples/)

**Java Longstream**

* [FindAny()](https://www.geeksforgeeks.org/longstream-findany-examples/)
* [NoneMatch()](https://www.geeksforgeeks.org/longstream-nonematch-java-examples/)
* [Min()](https://www.geeksforgeeks.org/longstream-min-java-examples/)
* [Max()](https://www.geeksforgeeks.org/longstream-max-java-examples/)
* [map(LongUnaryOperator mapper)](https://www.geeksforgeeks.org/longstream-maplongunaryoperator-mapper-java/)
* [Iterator()](https://www.geeksforgeeks.org/longstream-iterator-java/)
* [ForEach()](https://www.geeksforgeeks.org/longstream-foreach-method-java/)
* [ForEachOrdered()](https://www.geeksforgeeks.org/longstream-foreachordered-method-java/)
* [FlatMap(LongFunction mapper)](https://www.geeksforgeeks.org/longstream-flatmaplongfunction-mapper-java/)
* [Filter()](https://www.geeksforgeeks.org/longstream-filter-java-examples/)
* [Empty()](https://www.geeksforgeeks.org/longstream-empty-java-examples/)
* [FindFirst()](https://www.geeksforgeeks.org/longstream-findfirst-java/)
* [Of()](https://www.geeksforgeeks.org/longstream-of-in-java/)
* [AnyMatch()](https://www.geeksforgeeks.org/longstream-anymatch-java-examples/)
* [AllMatch()](https://www.geeksforgeeks.org/longstream-allmatch-java-examples/)
* [distinct()](https://www.geeksforgeeks.org/longstream-distinct-java-examples/)
* [Concat()](https://www.geeksforgeeks.org/longstream-concat-java/)
* [Average()](https://www.geeksforgeeks.org/longstream-average-java-examples/)
* [SummaryStatistics()](https://www.geeksforgeeks.org/longstream-summarystatistics-java/)
* [Sorted()](https://www.geeksforgeeks.org/longstream-sorted-java/)
* [Skip()](https://www.geeksforgeeks.org/longstream-skip-java/)
* [Sequential()](https://www.geeksforgeeks.org/longstream-sequential-java/)
* [Range()](https://www.geeksforgeeks.org/longstream-range-java/)
* [RangeClosed()](https://www.geeksforgeeks.org/longstream-rangeclosed-java/)
* [Peek()](https://www.geeksforgeeks.org/longstream-peek-java-examples/)
* [Parallel()](https://www.geeksforgeeks.org/longstream-parallel-java/)
* [Generate()](https://www.geeksforgeeks.org/longstream-generate-method-java/)
* [Sum()](https://www.geeksforgeeks.org/longstream-sum-java/)
* [Search an element](https://www.geeksforgeeks.org/search-element-array-using-java-util-stream-intstreamlongstream/)
* [Limit()](https://www.geeksforgeeks.org/longstream-limit-java/)
* [MapToObj()](https://www.geeksforgeeks.org/longstream-maptoobj-java/)
* [MapToDouble()](https://www.geeksforgeeks.org/longstream-maptodouble-java/)
* [LongStream asDoubleStream()](https://www.geeksforgeeks.org/longstream-asdoublestream-java/)
* [Reduce(Long binary operator)](https://www.geeksforgeeks.org/longstream-reducelong-identity-longbinaryoperator-op-java/)
* [Reduce()](https://www.geeksforgeeks.org/longstream-reducelongbinaryoperator-op-java/)
* [LongStream toArray()](https://www.geeksforgeeks.org/longstream-toarray-java/)
* [Boxed()](https://www.geeksforgeeks.org/longstream-boxed-java/)
* [Builder()](https://www.geeksforgeeks.org/longstream-builder-java/)

**Java Doublestream**

* [Empty()](https://www.geeksforgeeks.org/doublestream-empty-java-examples/)
* [FindFirst()](https://www.geeksforgeeks.org/doublestream-findfirst-examples/)
* [FlatMap()](https://www.geeksforgeeks.org/doublestream-flatmapdoublefunction-mapper-java/)
* [Sorted()](https://www.geeksforgeeks.org/doublestream-sorted-java/)
* [forEachOrdered()](https://www.geeksforgeeks.org/doublestream-foreachordered-method-java/)
* [ForEach()](https://www.geeksforgeeks.org/doublestream-foreach-method-java/)
* [Generate()](https://www.geeksforgeeks.org/doublestream-generate-method-java/)
* [SummaryStatistics()](https://www.geeksforgeeks.org/doublestream-summarystatistics-java/)
* [Filter()](https://www.geeksforgeeks.org/doublestream-filter-java-examples/)
* [Of()](https://www.geeksforgeeks.org/doublestream-of-in-java/)
* [Average()](https://www.geeksforgeeks.org/doublestream-average-java-examples/)
* [Map(DoubleUnaryOperator mapper)](https://www.geeksforgeeks.org/doublestream-mapdoubleunaryoperator-mapper-java/)
* [Iterator()](https://www.geeksforgeeks.org/doublestream-iterator-java/)
* [Max()](https://www.geeksforgeeks.org/doublestream-max-java-examples/)
* [AnyMatch()](https://www.geeksforgeeks.org/doublestream-anymatch-java-examples/)
* [Min()](https://www.geeksforgeeks.org/doublestream-min-java-examples/)
* [Count()](https://www.geeksforgeeks.org/doublestream-count-java-examples/)
* [Skip()](https://www.geeksforgeeks.org/doublestream-skip-java/)
* [Distinct()](https://www.geeksforgeeks.org/doublestream-distinct-java-examples/)
* [Concat()](https://www.geeksforgeeks.org/doublestream-concat-java/)
* [Sequential()](https://www.geeksforgeeks.org/doublestream-sequential-java/)
* [Peek()](https://www.geeksforgeeks.org/doublestream-peek-java-examples/)
* [FindAny()](https://www.geeksforgeeks.org/doublestream-findany-examples/)
* [AllMatch()](https://www.geeksforgeeks.org/doublestream-allmatch-java-examples/)
* [Parallel()](https://www.geeksforgeeks.org/doublestream-parallel-java/)
* [Sum()](https://www.geeksforgeeks.org/doublestream-sum-java/)
* [NoneMatch()](https://www.geeksforgeeks.org/doublestream-nonematch-java-examples/)
* [DoubleStream boxed()](https://www.geeksforgeeks.org/doublestream-boxed-java-examples/)
* [MapToInt()](https://www.geeksforgeeks.org/doublestream-maptoint-java/)
* [Limit()](https://www.geeksforgeeks.org/doublestream-limit-java-examples/)
* [Reduce()](https://www.geeksforgeeks.org/doublestream-reducedouble-identity-doublebinaryoperator-op-java/)
* [DoubleStream toArray()](https://www.geeksforgeeks.org/doublestream-toarray-java/)

#### **Variables used in lambda should be final or effectively final**

. On further research using SO forums and blog I learned that Java 8 has new concept called “Effectively final” variable. It means that a non-final local variable whose value never changes after initialization is called “Effectively Final”. This concept was introduced because prior to Java 8 we could not use a non-final local variable in an anonymous class. If you have access to a local variable in Anonymous class, you have to make it final.

When Lambdas was introduced, this restriction was eased. Hence to the need to make local variable final if it’s not changed once it is initialized as Lambda in itself is nothing but an anonymous class. Java 8 realized the pain of declaring local variable final every time developer used Lambda and introduced this concept and made it unnecessary to make local variables final. So if you see the rule for anonymous class still not changed, it’s just you don’t have to write final keyword every time when using lambdas.

OK. So now I understand what the error means but how do I solve my issue where I want to replace the variables using regex. Surprisingly, before I looked up

documentation and SO forums, I checked the options suggested by IntelliJ. It said – “Transform result variable into final one element array”. As I clicked continue, My code transformed to

**public String evaluate(){**

**final String[] result = {“${One}, ${two}, ${three}”};**

**variables.forEach((k,v)-&gt;{**

**String regex = "\\$\\{" + k + "\\}";**

**result[0] = result[0].replaceAll(regex, v);**

**});**

**return result[0];**

**}**

Now what the heck just happened? Why is a String not allowed to change but an Array is allowed to change? Going back to basics of Java I realized that when we decalre a variable final, it is the reference to the object that is set to final. So in this case when I write

##### **final String result= “abc”;**

The reference is STRING OBJECT **result —memory location —> “abc”** and because the result is declared final, it cannot change the memory reference and point to memory location who value is say**” DEF”**.

When I write

##### **final String [] result = {“abc”}**

In this case, result is pointing to a reference that is a Array and that array has value that reference **memory location** of **String “abc”**.

**FINAL STRING [] result —- Memory location –> Single Array —–Memory location —> “abc”.**

So as you see our result now references the memory location of Array element but the value Array element can change. Remember what is constant here is the reference, **not** its values.

**Stream.Iterate**

//This is like while true loop  
// IntStream.iterate(10,e->e).forEach(System.out::println);  
//iterrate(seed,function) ===> f(seed),f(f(seed)),f(f(f(seed)))...  
//eg "- 10-1=9 9-1=8 8-1=7  
IntStream.*iterate*(10,e->e-1).limit(10).forEach(System.*out*::println);  
System.*out*.println();  
Stream.*iterate*(new int[]{1,1},e->new int[]{e[1],e[0]+e[1]} ).limit(10)  
.forEach(e-> System.*out*.println(e[0]));  
System.*out*.println();  
Stream.*iterate*("\*",e->e+"\*").limit(10.forEach(System.*out*::println)

#### map

<R> [Stream](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html)<R> map([Function](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)<? super [T](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html),? extends R> mapper)

Returns a stream consisting of the results of applying the given function to the elements of this stream.

This is an [intermediate operation](https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html" \l "StreamOps).

**Type Parameters:**

R - The element type of the new stream

**Parameters:**

mapper - a [non-interfering](https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html" \l "NonInterference), [stateless](https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html" \l "Statelessness) function to apply to each element

**Returns:**

the new stream

#### filter

[Stream](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html)<[T](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html)> filter([Predicate](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html)<? super [T](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html)> predicate)

Returns a stream consisting of the elements of this stream that match the given predicate.

This is an [intermediate operation](https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html" \l "StreamOps).

**Parameters:**

predicate - a [non-interfering](https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html" \l "NonInterference), [stateless](https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html" \l "Statelessness) predicate to apply to each element to determine if it should be included

**Returns:**

the new stream

#### forEach

void forEach([Consumer](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html)<? super [T](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html)> action)

Performs an action for each element of this stream.

This is a [terminal operation](https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html" \l "StreamOps).

The behavior of this operation is explicitly nondeterministic. For parallel stream pipelines, this operation does not guarantee to respect the encounter order of the stream, as doing so would sacrifice the benefit of parallelism. For any given element, the action may be performed at whatever time and in whatever thread the library chooses. If the action accesses shared state, it is responsible for providing the required synchronization.

**Parameters:**

action - a [non-interfering](https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html" \l "NonInterference) action to perform on the elements