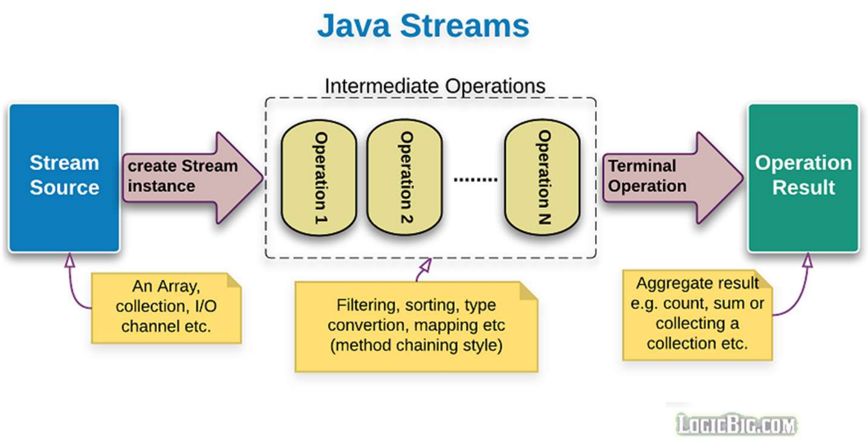
Streams in Java 8

- Streams *(java.util.stream)* are wrappers around a data source, that operate on the data source and make bulk processing convenient and fast.
- Streams do not store data, and hence is not a data structure.
- It is a sequence of objects that supports many different methods that can be pipelined to get the desired result.
- It is **functional** in nature.
- Designed for **lambdas**.
- It does not modify its source.
- Internal iteration.

Stages of a Stream:

creation -> intermediate operations -> terminal operation


Java Streams
Intermediate Operations
Stream Source
create Stream instance
Operation 1
Operation 2
....
Operation N
Terminal Operation
Operation Result
An Array, collection, I/O channel etc.
Filtering, sorting, type convertion, mapping etc (method chaining style)
Aggregate result e.g. count, sum or collecting a collection etc.
LogicBig.com

Intermediate operations can be chained (methods that return a stream)

Terminal operations do not return streams, and hence cannot be chained. They mark the end of the stream pipeline.

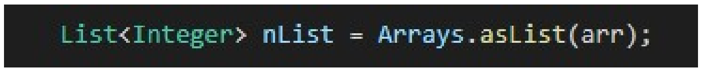
Note: Streams are different from I/O streams

Common Stream Operations

1. forEach()
2. map()
3. filter()
4. reduce()
5. collect()
6. findFirst()
7. toArray()
8. mapToInt()
9. peek()
10. count()
11. sorted()
12. min()
13. max()
14. distinct()

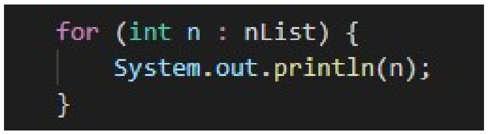
So why exactly are streams used?

Something as simple as iterating through a list. Consider an ArrayList of 100 integers. We have to loop through this list and process each element one by one. The traditional or enhanced for loop can be used for this, but it requires atleast 3 lines of code. The streams variant for the same, forEach() method, does the job in just 1 line.



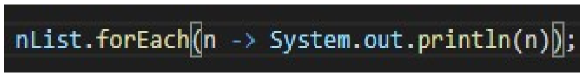
```
List<Integer> nList = Arrays.asList(arr);
```

<u>Enhanced for-loop</u>



```
for (int n : nList) {
    System.out.println(n);
}
```

<u>For-Each stream method</u>



```
nList.forEach(n -> System.out.println(n));
```

Creating a Stream

Method 1: Collection.stream()

Method 2: Stream.of(collection), StreamBuilder.add().add()….build(), Stream.iterate(), Stream.generate()

Creating an empty stream using Stream.empty()

Examples on stream creation:

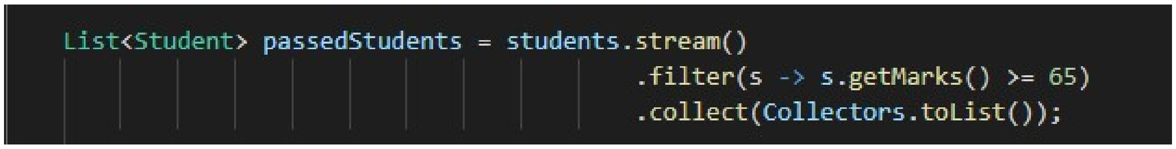
```
Stream<Integer> stream =
Stream.of(1,2,3,4,5,6,7,8,9);

Stream<Integer> stream = Stream.of(new
Integer[]{1,2,3,4,5,6,7,8,9});

Stream<Integer> stream = list.stream();
Stream<Date> stream = Stream.generate(() -> {
return new Date(); });
```

Intermediate Operations

Intermediate operations like map, filter, reduce, etc. accept a method reference as a parameter. We can also pass a lambda expression to these methods. This style is called functional programming.

For e.g. If we have a list of students from which we want to fetch only those students who have a score of more than 65%. This can be done easily using streams.

```
class Student {name: String, marks: int}
```



```
List<Student> passedStudents = students.stream()
                                        .filter(s -> s.getMarks() >= 65)
                                        .collect(Collectors.toList());
```

Wait, what did just happen?

Let’s break this down step-by-step…

Let’s say, we have a list of students by the name `students`

Ok… Looks like we’re trying to create a stream from this list… cool

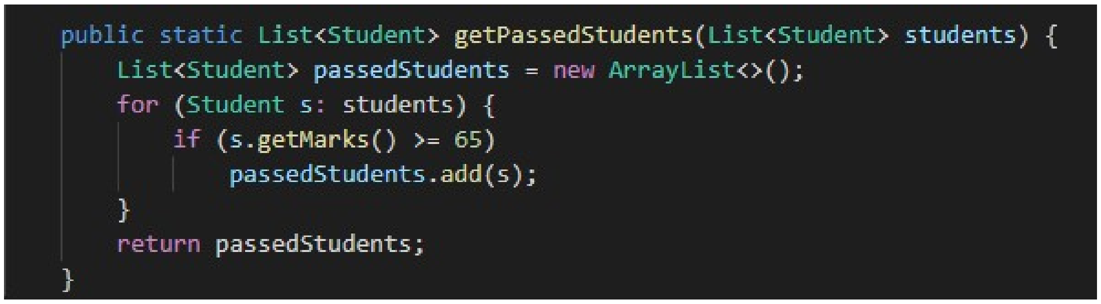
Then, we’re filtering the data from that stream based on some condition, ok… It is checking if the student’s marks are above 65 (that’s a lambda by the way)

And what is this collect method doing…? Maybe it’s gathering the processed data into a list and returning it…

That’s correct! Remember, streams don’t modify the source data? So, the results have to be stored somewhere… duh!

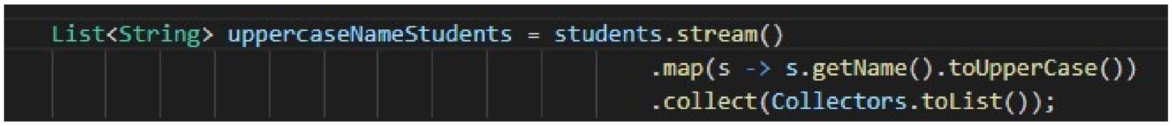
Ok… But, if collect() is returning a list, that means it must be one of the terminal operations. Correct again!

Let’s have a look at the normal way of doing the same thing…



```java
public static List<Student> getPassedStudents(List<Student> students) {
    List<Student> passedStudents = new ArrayList<>();
    for (Student s: students) {
        if (s.getMarks() >= 65)
            passedStudents.add(s);
    }
    return passedStudents;
}
```

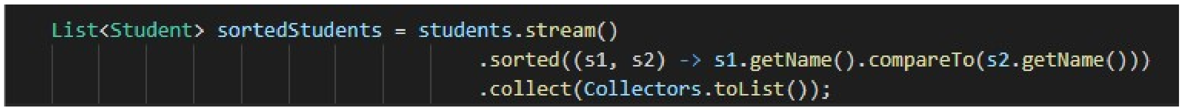
Let’s take a look at another example. We want to convert all the student names to uppercase for uniformity. How will we do this…? Streams to the rescue…!!!



```java
List<String> uppercaseNameStudents = students.stream()
                                             .map(s -> s.getName().toUpperCase())
                                             .collect(Collectors.toList());
```

Here, the map() method is invoking toUpperCase() for every Student object in the list. The map() method basically takes an input and “maps” it to some other output. This method is usually used for carrying out some tranformations.

Now, we want to sort these students in ascending order of their names…



```java
List<Student> sortedStudents = students.stream()
                                       .sorted((s1, s2) -> s1.getName().compareTo(s2.getName()))
                                       .collect(Collectors.toList());
```

The sorted() method is accepting a Comparable as an input. If we observe carefully, the lambda expression passed to this method is actually overriding the compare() method.

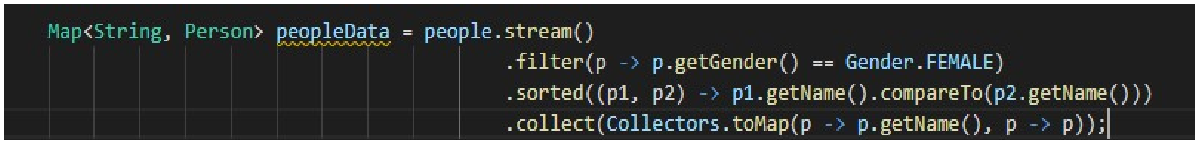
So, we can see that streams are getting the work done in just a few lines of code, and they are quite expressive too. The code is so natural, we just follow the specification of what we have to do at every step. Let’s elaborate a little bit on this…

- Removing boilerplate code
- More expressive code: What the code is actually doing can be easily understood just by looking at the code
- We just specify what is to be done, and the ‘how’ is taken care of by the streams API

Putting it all together…

Let’s say, we want to save peoples’ data where the people are women, and sort it according to their names. These people should be uniquely identified by their names, i.e. they are to be stored in a map.

```
class Person {name: String, gender: Gender}

enum Gender { MALE, FEMALE, OTHER }
```



```
Map<String, Person> peopleData = people.stream()
                                       .filter(p -> p.getGender() == Gender.FEMALE)
                                       .sorted((p1, p2) -> p1.getName().compareTo(p2.getName()))
                                       .collect(Collectors.toMap(p -> p.getName(), p -> p));
```

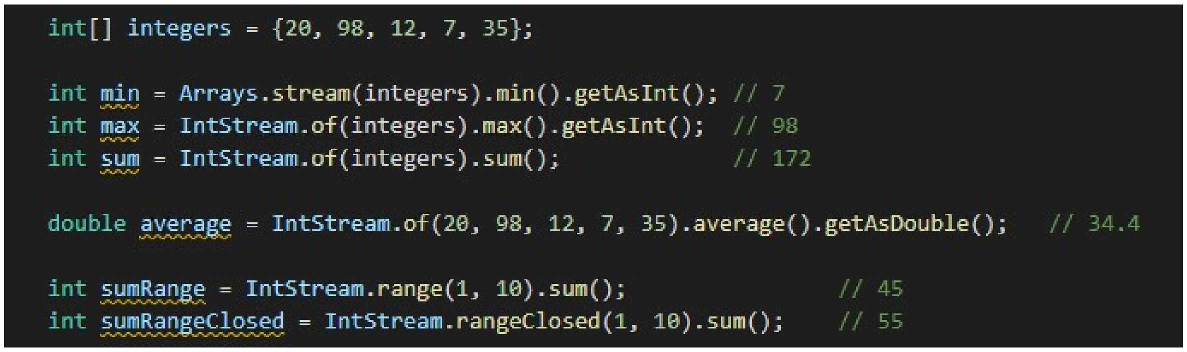
Try to make sense of this code…

Primitive Streams

Three variants – IntStream, LongStream and DoubleStream

These implementations are limited because of the autoboxing overhead.

Popular stream methods are min(), max(), sum(), average()



```
int[] integers = {20, 98, 12, 7, 35};

int min = Arrays.stream(integers).min().getAsInt(); // 7
int max = IntStream.of(integers).max().getAsInt();  // 98
int sum = IntStream.of(integers).sum();             // 172

double average = IntStream.of(20, 98, 12, 7, 35).average().getAsDouble();   // 34.4

int sumRange = IntStream.range(1, 10).sum();                // 45
int sumRangeClosed = IntStream.rangeClosed(1, 10).sum();    // 55
```

As we can see, streams have saved us from writing those boring for-loops just to get some simple results. All streams say is, “You just tell us, and we’ll get it done for you!”

Terminal Operations

We have been using terminal operations on streams in the previous examples. The data processing in streams is done only when the terminal operation is called. A terminal operation does not return a stream, and hence, further stream methods cannot be chained to it.

Following are some of the terminal methods:

- Stream.forEach() -- iterating over all elements of a stream and perform some operation on each of them.
- Stream.collect() -- used to receive elements from a steam and store them in a collection
- Stream.match() -- Various matching operations can be used to check whether a certain predicate matches the stream. All of those operations are terminal and return a boolean result.
- Stream.count() -- returns the number of elements in the stream as a long
- Stream.reduce() -- performs a reduction on the elements of the stream with the given function. The result is an Optional holding the reduced value.

The reduce() method keeps applying the same logic to all the elements of the stream recursively.

Short-circuit Operations

Some stream operations are performed on all elements inside a collection satisfying some condition (technically called a Predicate). If we want to break the operation whenever a matching element is found, we use short-circuit methods. In external iteration, we might end up using an if-else condition with break. Short-circuit methods are basically doing the same thing in internal iteration.

Some short-circuit methods:

- Stream.anyMatch() -- returns true once a condition passed as predicate satisfy. It will not process any more elements.
- Stream.findFirst() -- returns first element from stream and then will not process any more element.
- Stream.allMatch() -- used to check if all the elements of the stream match the provided predicate.

Parallel Streams

The Stream API enables developers to create the parallel streams that can take advantage of multi-core architectures and enhance the performance of Java code. In a parallel stream, the operations are executed in parallel and there are two ways to create a parallel stream.

1) Using the parallelStream() method on a collection
2) Using the parallel() method on a stream

Parallel Streams must be used only with stateless, non-interfering, and associative operations i.e.

1) A stateless operation is an operation in which the state of one element does not affect another element
2) A non-interfering operation is an operation in which data source is not affected
3) An associative operation is an operation in which the result is not affected by the order of operands

How Parallel Streams work:

- Parallel streams reside in a ThreadPool called the ForkJoinPool.
- Streams give us the flexibility to iterate over the list in a parallel pattern and can give the total in quick fashion.
- Stream implementation in Java is by default sequential unless until it is explicitly mentioned in parallel.
- When a stream executes in parallel, the Java runtime partitions the stream into multiple sub-streams.
- Aggregate operations iterate over and process these sub-streams in parallel and then combine the results.
- The only thing to keep in mind to create parallel stream is to call the parallelStream() method on the collection else by default the sequential stream gets returned by stream() method.

Disadvantages:

1) Since each sub-stream is a single thread running and acting on the data, it has overhead compared to the sequential stream
2) Inter-thread communication is dangerous and takes time for coordination

WHEN TO USE PARALLEL STREAMS?

1) They should be used when the output of the operation is not needed to be dependent on the order of elements present in source collection (i.e. on which the stream is created)
2) Parallel Streams can be used in case of aggregate functions
3) Parallel Streams quickly iterate over the large-sized collections
4) Parallel Streams can be used if developers have performance implications with the Sequential Streams
5) If the environment is not multi-threaded, then Parallel Stream creates thread and can affect the new requests coming in