**Functional Interface, Lambda and Stream in Java**

Diagram

Description automatically generated

Java 8 has brought functional programming into the language - or at least some parts of that. It is not perfect, verbose but still a nice and easy-to-use tool in our belt. In this article we will review this part of Java through many-many examples.

**Functional Interface**

Funcional interfaces are the basis of Lambda functions and Streams.  
It must contain only one abstract method. The @FunctionalInterface annotation can be used for compile time checking of this constraint.

Let’s create an interface which converts an input object from type T to type U .

@FunctionalInterface

interface Converter<T, U> {

U convert(T from);

}

Before Java 8 anonymous class had to be created from the interface.  
In this example we implement *String*to *Integer*converter.

Convert String to Integer using anonymous inner class

Converter<String, Integer> converter = new Converter<String, Integer>() {

@Override

public Integer convert(String from) {

return Integer.valueOf(from);

}

};

System.out.println(converter.convert("10"));

**Lambda expression**

From Java 8 we can use *lambda expressions* (and *method references* in some specific cases). These are syntactic sugars over inner classes. Before the *arrow*the String input parameter is declared and after the *arrow*the method body. The following picture shows how the*anonymous class* was transformed into *lambda function*.

A screenshot of a computer

Description automatically generated with medium confidence

Transform anonymous class into lambda function

As the function call in the body of the method (Integer.valueOf(from)) has the same input type as the input type of the lambda expression, we can shorten the expression even further by omiting the input parameter. This is what we call as *method reference*.

Convert String to Integer using lambda expression and method reference

Converter<String, Integer> fnConverter = (String from) -> Integer.valueOf(from);

System.out.println(fnConverter.convert("10"));

Converter<String, Integer> rfConverter = Integer::valueOf;

System.out.println(rfConverter.convert("10"));

Unfortunatelly the function name is still needed to the execution of the lambda expression thus there can be only one function on the interface so the language could omit that as well. The following expression would be nicer, maybe we will get this in the furute.

var result = rfConverter(10)

Another important thing is variables outside from the lambda’s context has to be at least effectively final. In the following example the count is incremented in the 5th line, therefore the variable is not effectively final so we will get compilation error. If we remove that line, the compilation will succeed.

void effectivelyFinal() {

int count = 10;

Converter<Integer, String> converter = from -> String.valueOf(from \* count);

System.out.println(converter.convert(10));

//count++;

}

**Built-In Functional Interface**

There are multiple built-in functional interfaces in Java. The best practice is using them and only define custom ones if it’s really necessary.

* [**Supplier<T>**](https://cr.openjdk.java.net/~iris/se/17/latestSpec/api/java.base/java/util/function/Supplier.html) has only return value. It is useful for object generation.

|  |
| --- |
| Supplier<String> strSupplier = () -> "Hello world"; |
|  | String str = strSupplier.get(); |

* [**Predicate<T>**](https://cr.openjdk.java.net/~iris/se/17/latestSpec/api/java.base/java/util/function/Predicate.html) has an input parameter and returns with *boolean*value. We can use it for testing values. Predicate has many default methods. We can combine multiple Predicates using them.

Pass predicate to a method to alter it’s behaviour

void printWhenLT(int value, Predicate<Integer> testValue) {

if (testValue.test(value)) {

System.out.println(value);

}

}

Predicate<Integer> lt10 = value -> value < 10;

Predicate<Integer> lt20 = value -> value < 20;

printWhenLT(5, lt10);

printWhenLT(15, lt10);

printWhenLT(25, lt10);

printWhenLT(10, lt20);

printWhenLT(20, lt20);

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

Predicate<Integer> gt5 = value -> value > 5;

Predicate<Integer> combined = lt10.and(gt5);

System.out.println(combined.test(5)); //false

System.out.println(combined.test(10)); //false

System.out.println(combined.test(7)); //true

* [**Function<T , R>**](https://cr.openjdk.java.net/~iris/se/17/latestSpec/api/java.base/java/util/function/Function.html) useful for convert elements from one type to another. It’s similar to the Converter interface we defined earlier but it contains *default*methods for composing functions. Composition has to be assembled in reverse order and the type of the preceding function’s output must be the same as the current function’s input type.

Function<String, Integer> convertStringToInt = Integer::valueOf;

Function<Integer, Integer> doubleInt = it -> it \* 2;

Function<Integer, String> convertIntToString = Object::toString;

var composed = convertIntToString.compose(doubleInt).compose(convertStringToInt);

composed.apply("10");

// "10" -> 10 -> 20 -> "20"

* [**Consumer<T>**](https://cr.openjdk.java.net/~iris/se/17/latestSpec/api/java.base/java/util/function/Consumer.html) has an input parameter and *void*return type. It is useful for processing values (save into database, write into file…).

Consumer<String> consumer = in -> System.out.println(in);

consumer.accept("Hello World");

Here are the definitions of the discussed interfaces for reference.

@FunctionalInterface

public interface Supplier<T> {

T get();

}

@FunctionalInterface

public interface Predicate<T> {

boolean test(T t);

[... other default methods]

}

@FunctionalInterface

public interface Function<T, R> {

R apply(T t);

[... other default methods]

}

@FunctionalInterface

public interface Consumer<T> {

void accept(T t);

[... other default methods]

}

There are multiple useful interfaces in the java.util.function package like *[BiFunction<T, U, R>](https://cr.openjdk.java.net/~iris/se/17/latestSpec/api/java.base/java/util/function/BiFunction.html" \t "_blank)***,***[BinaryOperator<T>](https://cr.openjdk.java.net/~iris/se/17/latestSpec/api/java.base/java/util/function/BinaryOperator.html" \t "_blank)*and so on*.*Feel free to discover all of them.

BinaryOperator extends from BiFunction. It has 2 input and 1 output value of the same type

BinaryOperator<Integer> binaryOperator = (a, b) -> a \* b;

binaryOperator.apply(3, 5);

Stream was the greatest novelty of Java 8. It can be used on various collections. We can define declarative pipelines using distinct operations. Each operations accept a lambda function and returns with the mutated view of the collection. The original collection won’t be modified though. Using stream is easier then using loops and also it is more secure.

There are two type of operations:

* **Intermediate**: a pipeline can contains multiple intermediate operations.
* **Terminal**: produce the final value (or *void*). Stream can have only one at the end of the pipeline. After the terminal operation the stream becomes closed and cannot be reused anymore.

Let’s jump into the available operations.

**Stream operators**

***forEach***accepts a Consumer, executes the given function and returns with void.

List<String> words= Arrays.asList("foo","bar","home","sword","play","animal","sword","car","sun","java");

words.stream().forEach(s-> System.out.println(s));

***count***is another terminal operation. As its name implies it returns with the number of the elements of the list.

var size = words.stream().count();

***filter***accepts a Predicate and drop values not pass on it. This is an intermediate operation, so after it, a terminal operation must be used. The .toList terminal operation collects the remaining elements into a new list.

var result=words.stream().filter(str -> str.length() < 4).toList();

The result is*[foo, bar, car, sun]*

***sort***is an intermediate operator which sorts the list using the input Comparator or sorts into natural order if no comparator was provided.

var result = words.stream().sorted((a,b)->a.compareTo(b)).toList();

or

var ordered = words.stream().sorted().toList();

Result: *[animal, bar, car, foo, home, java, play, sun, sword, sword]*

The natural order of strings are alphabet.

***map***transforms all elements of the list using the passed Function. We are going to *map*the list of strings into a list of Word objects.

class Word {

String name;

public Word(String name) {

this.name = name;

}

@Override

public String toString() {

return "word = " + name;

}

}

List<Word> mapped = words.stream().map(it -> new Word(it)).toList();

Result: *[word = foo, word = bar, word = home, word = sword, word = play, word = animal, word = sword, word = car, word = sun, word = java]*

While *map*transforms an element into another one, ***flatMap***transforms a list of lists into a list of elements. In this example the phone number list of each person will be flattened into a list of phone numbers.

class Person {

String name;

List<String> phoneNumber;

public Person(String name, List<String> phoneNumber) {

this.name = name;

this.phoneNumber = phoneNumber;

}

}

var people = Arrays.asList(

new Person("zlaval", Arrays.asList("1234567", "3485765", "2348924")),

new Person("zalerix", Arrays.asList("856774", "234652", "123653"))

);

var flatMapped = people.stream().flatMap(person -> person.phoneNumber.stream()).toList();

// result: [1234567, 3485765, 2348924, 856774, 234652, 123653]

***allMatch, anyMatch, noneMatch***terminal operations return with *boolean*value. The first one is true if the given Predicate is true for all elements, the second one is true if at least one element matches an the third one is only true if non of the elements match to the given Predicate.

boolean anyWordLongerThenFiveCharacter = words.stream().anyMatch(it -> it.length() > 5);

***distinct***filters all duplicates from the list.

var distinctList = words.stream().distinct().toList();

Result: *[foo, bar, home, sword, play, animal, car, sun, java]*

**reduce**is a terminal operation. It takes an element from the list in every rounds and combines it with the *accumulator*. The final result will be the latest value of the accumulator property. For example we can use it for summarizing the values in an Integer list.

var sum = IntStream.*range*(0, 100).reduce((accumulator, number) -> accumulator += number); //4950

Let’s see how it works inside.

The elements of the list: [1,3,7,10]

1, accumulator = 0 , element = 1 -> 0 + 1 (= 1)

2, accumulator = 1 , element = 3 -> 1 + 3 (= 4)

3, accumulator = 4 , element = 7 -> 4 + 7 (= 11)

4, accumulator = 11 , element = 10 -> 11 + 10 (= 21)

final result = 21

We can define the initial value of the accumulator field as well.

IntStream.*range*(0, 100).reduce(100,(accumulator, number) -> accumulator += number); //5050

Using **peek**we can take a look into the stream. It is useful for logging and debugging. For example ‘log’ the elements of the stream after applying uppercase transformation on them.

words.stream()

.map(String::toUpperCase)

.peek(System.out::println)

.toList();

Result: *FOO BAR HOME SWORD PLAY ANIMAL SWORD CAR SUN JAVA*

The real power of streams are the combination of the different operations. In this example we will filter duplicated elements, transform the words to uppercase letter, sort them into alphabetize order and transform the elements into Word objects and collect it into a collection of *List<Word>*.

var result = words.stream()

.distinct()

.filter(it -> it.length() > 3)

.map(String::toUpperCase)

.sorted()

.map(it -> new Word(it + " is a word"))

.toList();

// result: [word = ANIMAL is a word, word = HOME is a word, word = JAVA is a word, word = PLAY is a word, word = SWORD is a word]

Building this logic using loop consumes more time, it would be also error prone and difficult to read .. and we are not even talked about parallelization.

**Order of execution.**

We have to talk about the order of execution as well. We are gonna create a list with 4 elements and in the first example we will sort the list, then apply a transformation and at last we are gonna filter the elements. In the second one, we will filter the elements, then transform and at the end sort it.

List<String> words = Arrays.asList("foo", "bar", "home", "sword");

words.stream()

.sorted((a, b) -> {

System.out.println("Sort element, compare " + a + " to " + b);

return a.compareTo(b);

})

.map(str -> {

System.out.println("Map element: " + str);

return str;

})

.filter(str -> {

System.out.println("Filter element: " + str);

return str.length() > 3;

})

.forEach(str -> System.out.println("forEach loop: " + str));

Result:

Sort element, compare bar to foo

Sort element, compare home to bar

Sort element, compare home to foo

Sort element, compare sword to foo

Sort element, compare sword to home

Map element: bar

Filter element: bar

Map element: foo

Filter element: foo

Map element: home

Filter element: home

forEach loop: home

Map element: sword

Filter element: sword

forEach loop: sword

So what was happening? First of all, as you may see the **sort**operation made the flow stop, sorted the whole list and after thatthe elements were pushed to the next stage of the pipeline. Order is a stateful operation so it was executed horizontally. After sort, the elements went through to the remaining steps of the pipeline one by one vertically as the rest of the operations are stateless. The whole pipeline has taken 14 steps.

Now let’s see the second version of the pipline.

words.stream()

.filter(str -> {

System.out.println("Filter element: " + str);

return str.length() > 3;

})

.map(str -> {

System.out.println("Map element: " + str);

return str;

})

.sorted((a, b) -> {

System.out.println("Sort element, compare " + a + " to " + b);

return a.compareTo(b);

})

.forEach(str -> System.out.println("forEach loop: " + str));

Result:

Filter element: foo

Filter element: bar

Filter element: home

Map element: home

Filter element: sword

Map element: sword

Sort element, compare sword to home

forEach loop: home

forEach loop: sword

It has only taken 8 steps because we filtered out the unnecessary elements before the map and filter steps.  
The order of the pipeline has impact on the performance as well as the final result. Consider the case when we transform the words into longer words and then filtering the elements by length versus the reverse of this two steps.

**Numeric streams**

Primitive numeric values can be processed using this specific kind of streams. The main problem if the *Stream<Number>* is it can only handle *boxed*numbers. When we would like to apply operations on it, the value has to be outboxed which has high cost. Also for example an *Integer*object is 16 bytes while an *int*primitive is only 4 bytes. *IntStream*, *DoubleStream*.. solve this problem and also contain numerical value specific operations.

**SummaryStatistic**generates stats from the elements of the collection.

var stats = IntStream.*of*(1, 3, 5, 7, 9).summaryStatistics();

Result: *IntSummaryStatistics{count=5, sum=25, min=1, average=5,000000, max=9}*

We can also boxing the stream into *Stream<Integer>*

Stream<Integer> boxedStream = IntStream.*of*(1, 3, 5, 7, 9).boxed();

**Collectors**

We have already seen the most used terminal collector (*.toList()*) but sometimes different type of results are needed. Let’s see other built-in collectors and write our own as well.

**groupingBy** collects the elements into *map*by the given key. In this example we are gonna use *Student*’s birth year as key.

class Student{

String name;

Integer birth;

public Student(String name, Integer birth) {

this.name = name;

this.birth = birth;

}

@Override

public String toString() {

return name + " [" + birth + "]";

}

}

var students = List.of(

new Student("Joe", 2013),

new Student("Jill", 1987),

new Student("Alize", 1991),

new Student("Zalan", 1987),

new Student("Eric", 2013),

new Student("Alex", 2015),

new Student("Laura", 1991),

new Student("Brian", 2001),

new Student("Deniell", 2013),

new Student("Clark", 2005)

);

var studentsGroupedByBirthYear = students.stream().collect(Collectors.groupingBy(student -> student.birth));

/\* Result:

{

2001 = [Brian [2001]],

1987 = [Jill [1987], Zalan [1987]],

2005 = [Clark [2005]],

1991 = [Alize [1991], Laura [1991]],

2013 = [Joe [2013], Eric [2013], Deniell [2013]],

2015 = [Alex [2015]]

}

\*/

**averagingInt**calculates the average of the given numeric properties. The average of the birth year for the afforementioned students is 2001.6

Double averageBirthYear = students.stream().collect(Collectors.*averagingInt*(s -> s.birth));

**partitioningBy**divides the collection into two parts based on the given *Predicate*. The result will be a map which contains two keys, the records not match to the Predicate will be placed to the *false*key’s bucket while the others to the *true*’sbucket.

var result = students.stream().collect(Collectors.partitioningBy(s -> s.birth < 2005));

/\* Result:

{

false=[Joe [2013], Eric [2013], Alex [2015], Deniell [2013], Clark [2005]],

true=[Jill [1987], Alize [1991], Zalan [1987], Laura [1991], Brian [2001]]

}

\*/

There are many other operations, here is some non-exhaustive examples.

students.stream().mapToInt(s -> s.birth).max(); //2015

students.stream().map(s -> s.birth).max(Integer::compareTo); //2015

students.stream().collect(Collectors.summingInt(s -> s.birth)); //20016

students.stream().mapToInt(s -> s.birth).sum(); //20016

students.stream().map(s -> s.name).collect(Collectors.joining(" -|- "));

// Joe -|- Jill -|- Alize -|- Zalan -|- Eric -|- Alex -|- Laura -|- Brian -|- Deniell -|- Clark

**Parallel streams**

Concurrency is effective but difficult to do it in the right way. Fortunately we can handle it easily using the Stream API. We have to call parallelStream() function and that’s it.

students.parallelStream()

.map(s->s.name)

.toList();

Let’s compare the execution time of a sequential and a parallel streams. Here we create a list from one million unique strings and sort it sequentially and then concurrently.

var randomStrings = IntStream.range(1, 1\_000\_000)

.boxed()

.map(i -> UUID.randomUUID().toString()).toList();

long streamStartTime = System.nanoTime();

randomStrings.stream().sorted().toList();

long streamEndTime = System.nanoTime();

long ms = TimeUnit.NANOSECONDS.toMillis(streamEndTime - streamStartTime);

System.out.printf("Sequential sort: %d ms\n", ms);

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

long parallelStreamStartTime = System.nanoTime();

randomStrings.parallelStream().sorted().toList();

long parallelStreamEndTime = System.nanoTime();

long pms = TimeUnit.NANOSECONDS.toMillis(parallelStreamEndTime - parallelStreamStartTime);

System.out.println(String.format("Concurrent sort: %d ms", pms));

/\*

Sequential sort: 464 ms

=================================

Concurrent sort: 258 ms

\*/

Parallel stream was almost two time faster than the sequential.  
Before we instantly start to rewrite all streams to parallel, we have to ensure that the parallel version can be executed faster than the sequential one. Thread handling, context switches, merging the results of the threads add extra costs to the concurrent execution which makes it slower in many cases. Sequential pipelines over small collections are almost always faster. Splitting LinkedList for parallel processing has enormous cost compared to splitting an ArrayList. But the choice also depends on the number of CPU cores, the cost of the operations and so on.

Also worth to mention that every parallel stream use the common ForkJoinPool by default. The number of threads mimic to the number of CPU cores (or hyperthreads). We can check it using the following command:

var cores=ForkJoinPool.*commonPool*().getParallelism();

Pay attention to the slow stream operations as they can slow down other parts of the application whose are using the same common - on JVM level - ForkJoinPool. We can easily write application which runs out of threads quickly.

At last, let’s see how parallelStream splits the work and schedules them into threads.

List.of("1", "2", "3", "4", "5", "6", "7").parallelStream()

.map(s -> {

System.out.println("Map " + s + " value in thread: " + Thread.currentThread().getName());

return s;

})

.forEach(s -> System.out.println("Foreach " + s + " value in thread: " + Thread.currentThread().getName()));

/\*

Map 2 in thread: ForkJoinPool.commonPool-worker-1

Map 1 in thread: ForkJoinPool.commonPool-worker-2

Map 3 in thread: ForkJoinPool.commonPool-worker-6

Map 7 in thread: ForkJoinPool.commonPool-worker-3

Foreach 1 in thread: ForkJoinPool.commonPool-worker-2

Map 4 in thread: ForkJoinPool.commonPool-worker-4

Foreach 3 in thread: ForkJoinPool.commonPool-worker-6

Map 5 in thread: main

Foreach 4 in thread: ForkJoinPool.commonPool-worker-4

Foreach 7 in thread: ForkJoinPool.commonPool-worker-3

Map 6 in thread: ForkJoinPool.commonPool-worker-5

Foreach 2 in thread: ForkJoinPool.commonPool-worker-1

Foreach 6 in thread: ForkJoinPool.commonPool-worker-5

Foreach 5 in thread: main

\*/

My ForkJoinPool has 24 threads so every element could have been processed in different threads.

**Custom collector**

Let’s take a look at the signatura of the Stream’s collect method. There are multiple overloads of it. Now we are gonna use one of the simplest signature.

<R> R collect(Supplier<R> supplier,  
 BiConsumer<R, ? super T> accumulator,  
 BiConsumer<R, R> combiner);

It waits three parameter. *Supplier*creates new instance(s) of the container for holding the result. Accumulator appends the actual element of the current iteration to the container. Combiner is only necessary for parallel streams, where every thread creates its own container and collect the data into it and in the end those collections are merged into one by the combiner.

We create a collector which collects the elements into *LinkedList*.

students.stream().filter(s -> s.birth > 2005).collect(  
 LinkedList::new,  
 LinkedList::add,  
 LinkedList::addAll  
);

In the background the *LinkedList::new* supplier creates a new LinkedList instance. Then on every iterations the *LinkedList::add* *BiConsumer*gets two parameters, a list and an object and put the object into the list *[(list, student)->list.add(student)]*. In case of parallel stream, every threads do this steps and at the end all result lists are combined together by *LinkedList::addAll* *[(leftList,rightList)->leftList.addAll(rightList)]*

In the second example we are gonna simulate message receiving over network. The sender splits the message into byte chunks, wrap it into Message objects with the order and sends it over the network. The client receives the messages in random order, sort them and restores the original message from the chunks.

package stream;

import java.io.ByteArrayOutputStream;

import java.io.IOException;

import java.io.UncheckedIOException;

import java.nio.charset.StandardCharsets;

import java.util.\*;

import java.util.stream.Collector;

public class NetworkSim {

private static final int CHUNK\_SIZE\_IN\_BYTE = 5;

class Message {

byte[] data;

int order;

public Message(byte[] data, int order) {

this.data = data;

this.order = order;

}

}

public static void main(String[] args) {

var sim = new NetworkSim();

sim.execute();

}

void execute() {

var messages = generateMessages();

var collector = getCollector();

var comparator=getComparator(); //Comparator.comparingInt

var result = messages.stream()

.sorted(comparator)

.map(m -> m.data)

.collect(collector);

System.out.println(new String(result, StandardCharsets.UTF\_8));

}

private Comparator<Message> getComparator(){

return (o1, o2) -> Integer.compare(o1.order, o2.order);

}

private List<Message> generateMessages() {

byte[] inputData = "Welcome to the Java Stream article.".getBytes(StandardCharsets.UTF\_8);

List<Message> messages = new ArrayList<>();

for (int i = 0; i < inputData.length; i += CHUNK\_SIZE\_IN\_BYTE) {

byte[] slice = Arrays.copyOfRange(inputData, i, Math.min(inputData.length, i + CHUNK\_SIZE\_IN\_BYTE));

messages.add(new Message(slice, messages.size()));

}

Collections.shuffle(messages);

return messages;

}

private Collector<byte[], ByteArrayOutputStream, byte[]> getCollector() {

return Collector.of(

ByteArrayOutputStream::new,

(accumulator, element) -> {

try {

accumulator.write(element);

} catch (IOException e) {

throw new UncheckedIOException(e);

}

},

(buffer1, buffer2) -> {

try {

buffer2.writeTo(buffer1);

return buffer1;

} catch (IOException e) {

throw new UncheckedIOException(e);

}

},

ByteArrayOutputStream::toByteArray

);

}

}

The collector creates new ByteArrayOutputStream(s) and writes the received messages’ data into it one-by-one. If parallel stream was used, then it combines the buffers from each threads. At last, the toByteArray method creates the final merged array which contains the original message.

**File handling**

We can use stream for file handling as well. In the following example we read the*name.txt* file line-by-line, skip the first 20 rows and filter the remaining names by length. Then we take the first 10 rows from the result and map it into lowercase format and collect them into a list.  
The [*names.txt*](https://github.com/zlaval/article-java_stream/blob/master/src/main/resources/names.txt) contains 100 different names.

//Get the file URI. The file is placed into the resources folder.

var uri = getClass().getClassLoader().getResource("names.txt").toURI();

Predicate<String> lengthPredicate = (s) -> s.length() < 5;

var names = Files.readAllLines(Path.of(uri))

.stream()

.skip(20)

.filter(lengthPredicate)

.limit(10)

.map(String::toLowerCase)

.toList();

**Reusing stream**

As we discussed earlier, when the terminal operation is called on a stream it becomes terminated and we cannot use it anymore. The next example shows the exception thrown by Java in this case.

Stream<String> stringStream = words.stream()

.filter(s -> s.length() > 3)

.map(String::toUpperCase);

var sortedList = stringStream.sorted().toList();

var distinctList = stringStream.distinct().toList();

// Exception in thread "main" java.lang.IllegalStateException: stream has already been operated upon or closed

// at java.base/java.util.stream.AbstractPipeline.<init>(AbstractPipeline.java:203)

If we would like to create reusable streams, *Supplier*interface can be used. It generates new stream every time its get method is called.

Supplier<Stream<String>> streamSupplier = () -> words.stream()

.filter(s -> s.length() > 3)

.map(String::toUpperCase);

var sortedList = streamSupplier.get().sorted().toList();

var distinctList = streamSupplier.get().distinct().toList();

**Stream debugger**

Debugging stream can be difficult but [IntelliJ](https://www.jetbrains.com/idea/) contains the [Stream Trace](https://www.jetbrains.com/help/idea/analyze-java-stream-operations.html) plugin by default, where we can visualize the execution of the pipeline.

A picture containing graphical user interface

Description automatically generated

Stream Trace

As you may see, stream is a very powerful tool for data processing but sometimes it is not enough.  
There are tools/extensions also worth to mention.  
The first one is [Vavr](https://docs.vavr.io/" \t "_blank). It is a complete funtional programming library for Java. It contains funtional interfaces, collections, extended stream operations and so on.

import io.vavr.collection.List;

List<String> words = List.of("foo", "bar", "home", "sword", "play", "animal", "sword", "car", "sun", "java");

var result = words.filter(s -> s.length() > 3)

.drop(2)

.patch(2, List.of("goku", "kalel"), 4)

.sorted()

.map(String::toUpperCase);

// List(ANIMAL, GOKU, KALEL, PLAY)

[StreamEx](https://github.com/amaembo/streamex) and [Eclipse collections](https://github.com/eclipse/eclipse-collections/blob/master/docs/guide.md) are another great tools and worth to check them.

The source code available on [GitHub](https://github.com/zlaval/article-java_stream).