Object-Oriented Programming Using Java

Java Streams

Quan Thai Ha

2022

■ Suppose we want to iterate over a list of integers and find out sum of all the integers greater than 10. Prior to Java 8, the approach to do it would be:

```java
1   private static int sumIterator(
        List<Integer> list) {
3     Iterator<Integer> it = list.iterator();
      int sum = 0;
5     while (it.hasNext()) {
        int num = it.next();
7       if (num > 10) {
          sum += num;
9       }
      }
11    return sum;
   }
```

■ There are three major problems with the above approach:

▶ We just want to know the sum of integers but we would also have to provide how the iteration will take place, this is also called external iteration because client program is handling the algorithm to iterate over the list.

▶ The program is sequential in nature, there is no way we can do this in parallel easily.

▶ There is a lot of code to do even a simple task.

- To overcome all the above shortcomings, Java 8 Stream API was introduced.

  ▶ We can use Java Stream API to implement internal iteration, that is better because java framework is in control of the iteration. Internal iteration provides several features such as sequential and parallel execution, filtering based on the given criteria, mapping etc.

  ▶ Most of the Java 8 Stream API method arguments are functional interfaces, so lambda expressions work very well with them.

  ▶ Let's see how can we write above logic in a single line statement using Java Streams.

```java
private static int sumStream(List<Integer> list) {
  return list.stream().filter(i -> i > 10).mapToInt(i -> i).sum();
}
```

  ▶ Above program utilizes java framework iteration strategy, filtering and mapping methods and would increase efficiency.

Java Streams

- When we start watching a video on the Internet, a small portion of the video file is first loaded into our computer and starts playing. we don't need to download the complete video before we start watching it. This is called video streaming. At a very high level, we can think of the small portions of the video file as a stream and the whole video as a Collection.

- A Stream in Java can be defined as a sequence of elements from a source. The source of elements here refers to a Collection or Array that provides data to the Stream.

- Java streams are designed in such a way that most of the stream operations (called intermediate operations) return a Stream. This helps to create a chain of stream operations. This is called stream pipeline.

- Java streams also support the aggregate or terminal operations on the elements. The aggregate operations are operations that allow us to express common manipulations on stream elements quickly and clearly, for example, finding the max or min element, finding the first element matching giving criteria, and so on.

- An operation on a stream produces a result without modifying its source.

- Streams do not store data; rather, they provide data from a source such as a collection, array, or IO channel.
- Streams do no modify the data source. it operates on the source data structure (collection and array) and produce pipelined data that we can use and perform specific operations, in which elements are computed on demand. Such as we can create a stream from the list and filter it based on a condition.
- This concept gives rise to significant programming benefits. The idea is that a user will extract only the values they require from a Stream, and these elements are produced, invisibly to the user, as and when required. This is a form of a producer-consumer relationship.
- Java Stream operations use functional interfaces, that makes it a very good fit for functional programming using lambda expression. Using lambda expressions make our code readable and short.
- In Java, java.util.Stream interface represents a stream on which one or more operations can be performed. Stream operations are either intermediate or terminal.

- Many stream operations are lazily-evaluated. This allows for automatic code optimizations and short-circuit evaluation.
- Stream can be infinite. Method such as limit allow us to get some result from infinite streams.
- The elements of a stream can be reached only once during the life of a stream. Like an Iterator, a new stream must be generated to revisit the same elements of the source.
- Java Stream support sequential as well as parallel processing, parallel processing can be very helpful in achieving high performance for large collections.
- Streams have methods, such as forEach and forEachOrdered, for internal iteration of stream elements.
- Since we can use primitive data types such as int, long in the collections using auto-boxing and these operations could take a lot of time, there are specific classes for primitive types - IntStream, LongStream and DoubleStream.
- All the Java Stream API interfaces and classes are in the java.util.stream package.

Streams differ from Collections in serveral ways:

- **No Storage**: A Java Collection is an in-memory data structure with all elements contained within memory. Every element in the Collection has to be computed before it can be added to the Collection. A stream is not a data structure that stores elements; instead, it conveys elements from a source such as a Collection, an array, a generator function, or an I/O channel, through a pipeline of operations.

- **Functional-Programming Style and Immutable Source**: A stream operation on a source produces a new stream. It does not modify its source.

- **Laziness**: Streams are lazy because the intermediate operations, such as filtering and mapping, are not evaluated until the terminal operation is invoked. This exposes opportunities for optimization. For example, "find the first string (findFirst() operation) with three consecutive vowels (intermediate filtering operation)", you need not filter all the source strings when the terminal operation is invoked. Another example: a filter-map-sum pipeline can be fused into a single pass on the data, with minimal intermediate state.

- **Possibly Unbounded Source**: While collections have a finite size, streams need not. Short-circuiting operations such as limit(n), findFirst() can allow computations on infinite streams.

- **Stateless/Stateful intermediate operations**: Intermediate operations are divided into stateless and stateful operations. Stateless operations, such as filter and map, retain no state from the previously seen element when processing a new element. Stateful operations, such as sorted and distinct, need to incorporate state from previously seen elements when processing a new element, and need to process the entire input before producing a result. Parallel pipeline with stateful operations may require multiple passes or may need to buffer significant data.

- **Short-circuiting operations**: A short-circuit intermediate or terminal operation can produce a new finite stream or result in a finite time, given an infinite input.

For example, the following stream filtering operation does not modified its source.

```java
import java.util.List;
import java.util.function.Predicate;
import java.util.stream.Collectors;

public class StreamImmutableTest {
  public static void main(String[] args) {
    List<Person> pList = List.of(new Person("Peter", 21),
    new Person("John", 60), new Person("Paul", 15));
    System.out.println(pList); // [Peter(21), John(60), Paul(15)]

    List<Person> adultList = pList.stream()
      .filter(p -> p.getAge() >= 21) // filter with a Predicate
      .collect(Collectors.toList()); // collect into a List
    System.out.println(adultList);   // [Peter(21), John(60)]

    // The source is not modified, i.e., immutable source
    System.out.println(pList); // [Peter(21), John(60), Paul(15)]
  }
}
```

- Java Stream API operations that returns a new Stream are called intermediate operations. Most of the times, these operations are lazy in nature, so they start producing new stream elements and send it to the next operation.

- Intermediate operations are never the final result producing operations.

- Commonly used intermediate operations are filter and map.

- Java Stream API operations that returns a result or produce a side effect. Once the terminal method is called on a stream, it consumes the stream and after that we can't use stream.

- Terminal operations are eager in nature i.e they process all the elements in the stream before returning the result.

- Commonly used terminal methods are forEach, toArray, min, max, findFirst, anyMatch, allMatch etc.

- You can identify terminal methods from the return type, they will never return a Stream.

- An intermediate operation is called short circuiting, if it may produce finite stream for an infinite stream. For example limit() and skip() are two short circuiting intermediate operations.

- A terminal operation is called short circuiting, if it may terminate in finite time for infinite stream. For example anyMatch, allMatch, noneMatch, findFirst and findAny are short circuiting terminal operations.

- JDK 8 introduces Stream API to process collections of objects. A Stream is a sequence of elements that supports sequential and parallel aggregate operations (such as filter, map and reduce), which can be pipelined to produce the desired result.

- A pipeline is a sequence of operations on a Collection. The sequence composes:

  ▶ A source: a Collection, an array, a generator function, or I/O channel of objects.

  ▶ .stream() or .parallelStream(): produce a Stream, which is a sequence of elements to be carried from the source into the pipeline.

  ▶ Some (zero or more) intermediate operations: for example, filter(Predicate) and map(Function).

  ▶ A terminal operation: such as forEach(), which produces the desired result (a value or side-effect).

  ▶ Intermediate operations are lazy. They will not be invoked until the terminal operation is executed. This improves the performance when we are processing larger data streams.

- JDK 8 introduces a new package java.util.function, which provides a number of Standard Functional Interfaces. Besides declaring one abstract method, these interfaces also heavily use default and static methods (with implementation) to enhance their functionality.
- There are four basic patterns (suppose that t is an instance of T):

Pattern	Functional Interface	Lambda Expression	Explanation
Predicate	Predicate<T> BiPredicate<T, U>	t -> boolean (t, u) -> boolean	Apply boolean test on the given element t
Function	Function<T, R> BiFunction<T, U, R> UnaryOperator<T> BinaryOperator<T>	t -> r (t, u) -> r t -> t (t, t) -> t	Transform (Map) from t to r
Consumer	Consumer<T> BiConsumer<T, U>	t -> void (t, u) -> void	Consume t with side effect such as printing
Supplier	Supplier<T> BooleanSupplier	() -> t () -> boolean	Supply an instance t

- For greater efficiency, specialized functional interfaces are defined for primitive types int, long and double, as summarized below:

Pattern	Functional Interface	Lambda Expression
Predicate	IntPredicate	int -> boolean
Function	IntFunction<R> IntToDoubleFunction IntToLongFunction ToIntFunction<T> ToIntBiFunction<T, U> IntUnaryOperator IntBinaryOperator	int -> r int -> double int -> long t -> int (t, u) -> int int -> int (int, int) -> int
Consumer	IntConsumer ObjIntConsumer<T>	int -> void (t, int) -> void
Supplier	IntSupplier	() -> int

Notes: Same Functional Interfaces as int are also defined for primitives long and double.

HUS

- Java Predicate in general meaning is a statement about something that is either true or false. In programming, predicates represent single argument functions that return a boolean value.

- In Java, *Predicate<T>* is a generic functional interface that represents a single argument function that returns a boolean value (true or false). This interface available in java.util.function package and contains a test(T t) method that evaluates the predicate of a given argument.

```java
@FunctionalInterface
public interface Predicate <T> {
    boolean test(T t);
}
```

- It represents a predicate against which elements of the stream are tested. This is used to filter elements from the java stream.

```java
import java.util.List;
import java.util.function.Predicate;

class BiggerThanFive<E> implements Predicate<Integer> {
    @Override
    public boolean test(Integer v) {
        Integer five = 5;
        return v > five;
    }
}

public class PredicateTest {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(2, 3, 1, 5, 6, 7, 8, 9, 12);
        BiggerThanFive<Integer> biggerThanfive = new BiggerThanFive<>();
        numbers.stream().filter(biggerThanfive).forEach(System.out::println);
    }
}
```

- Java Function represents a function that takes one type of argument and returns another type of argument. *Function<T, R>* is the generic form where *T* is the type of the input to the function and *R* is the type of the result of the function.

```java
@FunctionalInterface
public interface Function <T, R> {
    /**
     * Applies this function to the given argument.
     *
     * @param t the function argument
     * @return the function result
     */
    R apply (T t);
}
```

```java
import java.util.function.Function;

class SquareFunction implements Function<Integer, String> {

  @Override
  public String apply(Integer t) {
    return Integer.toString(t*t);
  }
}

public class FunctionTest {
  public static void main(String[] args) {
    Function<Integer, String> squareFunction = new SquareFunction();
    System.out.println(squareFunction.apply(2));
  }
}
```

- Java Consumer is a functional interface which represents an operation that accepts a single input argument and returns no result.

```java
@FunctionalInterface
public interface Consumer<T> {
    /**
     * Performs this operation on the given argument.
     *
     * @param t the input argument
     */
    void accept(T t);
}
```

- Unlike most other functional interfaces, Consumer is expected to operate via side-effects.
- It can be used to perform some action on all the elements of the java stream.

```java
import java.util.function.Consumer;

public class ConsumerTest {

  public static void main(String[] args) {
    // Create consumer
    Consumer<String> consumer = new Consumer<String>() {
      @Override
      public void accept(String name) {
        System.out.println("Hello, " + name);
      }
    };

    // Calling Consumer method
    consumer.accept("Java");  // Hello, Java
  }
}
```

- Java Supplier is a functional interface which represents an operation that returns a result. Supplier does not take any arguments.

```java
1  @FunctionalInterface
   public interface Supplier <T> {
3    /**
      * Gets a result.
5     *
      * @return a result
7    */
     T get();
9  }
```

Supplier Interface Example

```java
import java.util.function.Supplier;

public class SupplierTest {

  public static void main(String[] args) {

    // Create random supplier
    Supplier<Double> randomSupplier = () -> Math.random();

    System.out.println("Random value: " + randomSupplier.get());
    System.out.println("Random value: " + randomSupplier.get());
    System.out.println("Random value: " + randomSupplier.get());
  }
}
```

- Java Optional is a container object which may or may not contain a non-null value. If a value is present, *isPresent()* will return true and *get()* will return the value. Stream terminal operations return Optional object.
- Some of these methods are:

Syntax
Optional<T> reduce(BinaryOperator<T> accumulator)
Optional<T> min(Comparator<? super T> comparator)
Optional<T> max(Comparator<? super T> comparator)
Optional<T> findFirst()
Optional<T> findAny()

- For supporting parallel execution in Java 8 Stream API, Spliterator interface is used. Spliterator *trySplit* method returns a new Spliterator that manages a subset of the elements of the original Spliterator.

CREATING STREAMS

Streams are created from various sources such as collections, arrays, strings, IO resources, or generators.

1. **From a Collection<E>**: We can use Collection stream() to create sequential stream and parallelStream() to create parallel stream.

```java
List<Integer> list = new ArrayList<Integer>();
for(int i = 1; i< 10; i++){
    list.add(i);
}

// Sequential stream
Stream<Integer> stream = list.stream();
stream.forEach(p -> System.out.println(p));

// Parallel stream
Stream<Integer> parallelStream = myList.parallelStream();
```

2. **From an Object Array**: via static method Arrays.stream(Object[]).

```java
import java.util.Arrays;

public class ArrayToStreamTest {
  public static void main(String[] args) {
    // Create an array of Person objects
    Person[] personArray = {new Person("Peter", 21),
      new Person("John", 60),
      new Person("Paul", 15)};
    System.out.println(Arrays.toString(personArray));
    // [Peter(21), John(60), Paul(15)]

    Arrays.stream(personArray)
      .filter(p -> p.getAge() >= 21)
      .forEach(System.out::println);
  }
}
```

3. From static factory method of the Stream class, such as Stream.of(Object[]),
   Stream.of(Object...), IntStream.range(int, int), or Stream.iterate(Object, UnaryOperator).

   ▶ **Stream.of()**: We can use Stream.of() to create a stream from similar type of data. For example,
     we can create Java Stream of integers from a group of int or Integer objects.

```java
Stream<Integer> stream = Stream.of(1,2,3,4,5,6,7,8,9);
stream.forEach(p -> System.out.println(p));
```

■ **Stream.of(array)**: We can use Stream.of() with an array of Objects to return the stream. Note that it doesn't support autoboxing, so we can't pass primitive type array.

```java
Stream<Integer> stream = Stream.of(new Integer[]{1,2,3,4});
// Works fine

Stream<Integer> stream1 = Stream.of(new int[]{1,2,3,4});
// Compile time error, Type mismatch: cannot convert from Stream<int[]>
// to Stream<Integer>
```

- **Stream.generate() or Stream.iterate()**: Creating a stream from generated elements. This will produce a stream of 20 random numbers. We have restricted the elements count using limit() function.

```java
Stream<Integer> randomNumbers = Stream
  .generate(() -> (new Random()).nextInt(100));

randomNumbers.limit(20).forEach(System.out::println);
```

- **Stream of String chars or tokens**: Creating a stream from the characters of a given string. In the second part, we are creating the stream of tokens received from splitting from a string.

```java
IntStream stream = "12345_abcdefg".chars();
stream.forEach(p -> System.out.println(p));

// OR

Stream<String> stream = Stream.of("A,B,C".split(","));
stream.forEach(p -> System.out.println(p));
```

After performing the intermediate operations on elements in the stream, we can collect the processed elements again into a Collection using the stream Collector methods.

- **Collect Stream elements to a List**: first, we are creating a stream on integers 1 to 10. Then we are processing the stream elements to find all even numbers.
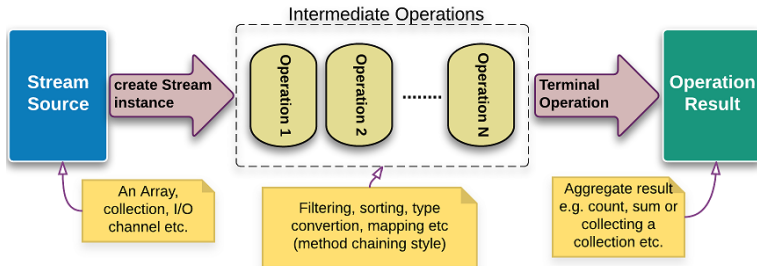
```java
List<Integer> list = new ArrayList<Integer>();

for(int i = 1; i < 10; i++){
  list.add(i);
}

Stream<Integer> stream = list.stream();
List<Integer> evenNumbersList = stream.filter(i -> i%2 == 0)
  .collect(Collectors.toList());
System.out.print(evenNumbersList);
```

- **Collect Stream elements to an Array**: first, we are creating a stream on integers 1 to 10. Then we are processing the stream elements to find all even numbers.

```java
List<Integer> list = new ArrayList<Integer>();

for(int i = 1; i< 10; i++){
    list.add(i);
}

Stream<Integer> stream = list.stream();
Integer[] evenNumbersArr = stream.filter(i -> i%2 == 0)
    .toArray(Integer[]::new);
System.out.print(evenNumbersArr);
```

- There are plenty of other ways also to collect stream into a Set, Map.

- Intermediate operations return the stream itself so you can chain multiple methods calls in a row.

- Terminal operations return a result of a certain type after processing all the stream elements. Once the terminal operation is invoked on a Stream, the iteration of the Stream and any of the chained streams will get started. Once the iteration is done, the result of the terminal operation is returned.

Java Streams

1. **Stream.filter()**: The filter() method accepts a Predicate to filter all elements of the stream. This operation is intermediate which enables us to call another stream operation (e.g. forEach()) on the result.

```java
List<Integer> myList = new ArrayList<>();
for (int i = 0; i < 100; i++) {
    myList.add(i);
}

Stream<Integer> sequentialStream = myList.stream();

// Filter numbers greater than 90
Stream<Integer> highNums = sequentialStream.filter(p -> p > 90);

System.out.print("High Nums greater than 90 = ");
highNums.forEach(p -> System.out.print(p + " "));
// Prints "High Nums greater than 90 = 91 92 93 94 95 96 97 98 99 "
```

2. **Stream.map()**: The map() intermediate operation converts each element in the stream into another object via the given function. The following example converts each string into an UPPERCASE string. But we can use map() to transform an object into another type as well.

```java
Stream<String> names = Stream.of("aBc", "d", "ef");
System.out.println(names.map(s -> {
    return s.toUpperCase();
}).collect(Collectors.toList()));
// Prints [ABC, D, EF]
```

3. **Stream.sorted()**: The sorted() method is an intermediate operation that returns a sorted view of the stream. The elements in the stream are sorted in natural order unless we pass a custom Comparator.

```java
Stream<String> names2 = Stream.of("aBc", "d", "ef", "123456");
List<String> reverseSorted = names.sorted(Comparator.reverseOrder())
    .collect(Collectors.toList());
System.out.println(reverseSorted);   // [ef, d, aBc, 123456]

Stream<String> names3 = Stream.of("aBc", "d", "ef", "123456");
List<String> naturalSorted = names2.sorted().collect(Collectors.toList());
System.out.println(naturalSorted);   // [123456, aBc, d, ef]
```

- Please note that the sorted() method only creates a sorted view of the stream without manipulating the ordering of the source Collection. In this example, the ordering of string in the names2, names3 is untouched.

4. **Stream.flatMap()**: We can use flatMap() to create a stream from the stream of list.

```java
Stream<List<String>> namesOriginalList = Stream.of(
  Arrays.asList("Pankaj"),
  Arrays.asList("David", "Lisa"),
  Arrays.asList("Amit"));

// Flat the stream from List<String> to String stream
Stream<String> flatStream = namesOriginalList
  .flatMap(strList -> strList.stream());

flatStream.forEach(System.out::println);
```

Intermediate Op	Full Syntax
filter(p)	Stream<T> filter(Predicate<Ts> predicate)
map(f)	<R> Stream<R> map(Function<Ts, Re> mapper)
mapToInt(f)	IntStream mapToInt(ToIntFunction<Ts> mapper)
mapToLong(f)	LongStream mapToLong(ToLongFunction<Ts> mapper)
mapToDouble(f)	DoubleStream mapToDouble(ToDoubleFunction<Ts> mapper)
sorted()	Stream<T> sorted()
sorted(comp)	Stream<T> sorted(Comparator<Ts> comparator)
distinct()	Stream<T> distinct()
limit(maxSize)	Stream<T> limit(long maxSize)
skip(n)	Stream<T> skip(long n)

1. **Stream.forEach()**: The forEach() method helps in iterating over all elements of a stream and perform some operation on each of them. The operation to be performed is passed as the lambda expression.

```java
Stream<Integer> numbers = Stream.of(1, 2, 3, 4, 5);
numbers.forEach(i -> System.out.print(i + ",")); // 1,2,3,4,5,
```

2. **Stream.collect()**: The collect() method is used to receive elements from a steam and store them in a collection.

```java
List<String> namesInUppercase = names.stream().sorted()
 .map(String::toUpperCase)
 .collect(Collectors.toList());

System.out.print(namesInUppercase);
```

3. **Stream.match()**: Various matching operations can be used to check whether a given predicate matches the stream elements. All of these matching operations are terminal and return a boolean result.

```java
Stream<Integer> numbers2 = Stream.of(1, 2, 3, 4, 5);
System.out.println("Stream contains 4? " + numbers2.anyMatch(i -> i == 4));
// Stream contains 4? true

Stream<Integer> numbers3 = Stream.of(1, 2, 3, 4, 5);
System.out.println("Stream contains all elements less than 10? "
 + numbers3.allMatch(i -> i < 10));
// Stream contains all elements less than 10? true

Stream<Integer> numbers4 = Stream.of(1, 2, 3, 4, 5);
System.out.println("Stream doesn't contain 10? "
 + numbers4.noneMatch(i -> i == 10));
// Stream doesn't contain 10? true
```

4. **Stream.count()**: The count() is a terminal operation returning the number of elements in the stream as a long value.

```java
Stream<Integer> numbers5 = Stream.of(1, 2, 3, 4, 5);

System.out.println("Number of elements in stream="
 + numbers5.count());   // 5
```

5. **Stream.reduce()**: The reduce() method performs a reduction on the elements of the stream with the given function. The result is an Optional holding the reduced value. In the given example, we are reducing all the strings by concatenating them using a separator #.

```java
Stream<Integer> numbers6 = Stream.of(1, 2, 3, 4, 5);

Optional<Integer> intOptional = numbers6.reduce((i, j) -> {return i*j;});
if (intOptional.isPresent()) {
  System.out.println("Multiplication = " + intOptional.get());
} // 120
```

6. **Stream.findFirst()**: This is a short circuiting terminal operation, let's see how we can use it to find the first string from a stream starting with D.

```java
Stream<String> names7 = Stream.of("Pankaj", "Amit", "David", "Lisa");
Optional<String> nameWithD = names7.filter(i -> i.startsWith("D"))
  .findFirst();

if (firstNameWithD.isPresent()) {
  System.out.println("Name starting with D = " + nameWithD.get());
} // David
```

Intermediate Op	Full Syntax
forEach(consumer)	void forEach(Consumer<Ts> action)
peek(consumer)	Stream<T> peek(Consumer<Ts> action)
reduce()	Optional<T> reduce(BinaryOperator<T> accumulator) T reduce(T identity, BinaryOperator<T> accumulator) <U> U reduce(U identity, BiFunction<U, Ts, U> accumulator, BinaryOperator<U> combiner)
collect()	<R,A> R collect(Collector<Ts, A, R> collector) <R> R collect(Supplier<R> supplier, BiConsumer<R, Ts> accumulator, BiConsumer<R, R> conbiner)
max() min() count()	Optional<T> max(Comparator<Ts> comparator) Optional<T> min(Comparator<Ts> comparator) long count()
findAny() findFirst()	Optional<T> findAny() Optional<T> findFirst()
allMatch(pred) anyMatch(pred) noneMatch(pred)	boolean allMatch(Predicate<Ts> predicate) boolean anyMatch(Predicate<Ts> predicate) boolean noneMatch(Predicate<Ts> predicate)
toArray()	Object[] toArray() <A> A[] toArray(IntFunction<A[]> generator)

- Creating Streams
  - ► concat()
  - ► empty()
  - ► generate()
  - ► iterate()
  - ► of()

- Intermediate Operations
  - ► filter()
  - ► map()
  - ► flatMap()
  - ► distinct()
  - ► sorted()
  - ► peek()
  - ► limit()
  - ► skip()

- Terminal Operations
  - ► forEach()
  - ► forEachOrdered()
  - ► toArray()
  - ► reduce()
  - ► collect()
  - ► min()
  - ► max()
  - ► count()
  - ► anyMatch()
  - ► allMatch()
  - ► noneMatch()
  - ► findFirst()
  - ► findAny()

📕 Allen B. Downey, Chris Mayfield, *Think Java* (2016).

📕 Martin Fowler, *UML Distilled - A Brief Guide To The Standard Object Modeling Language* (2004).

📕 Richard Warburton, *Object-Oriented vs. Functional Programming - Bridging the Divide Between Opposing Paradigms* (2016).

📕 James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, *The Java Language Specification - Java SE 8 Edition* (2015).

Thank You!