Stream API

Функциональное программирование

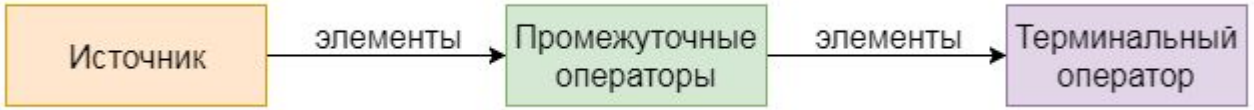
1.Stream

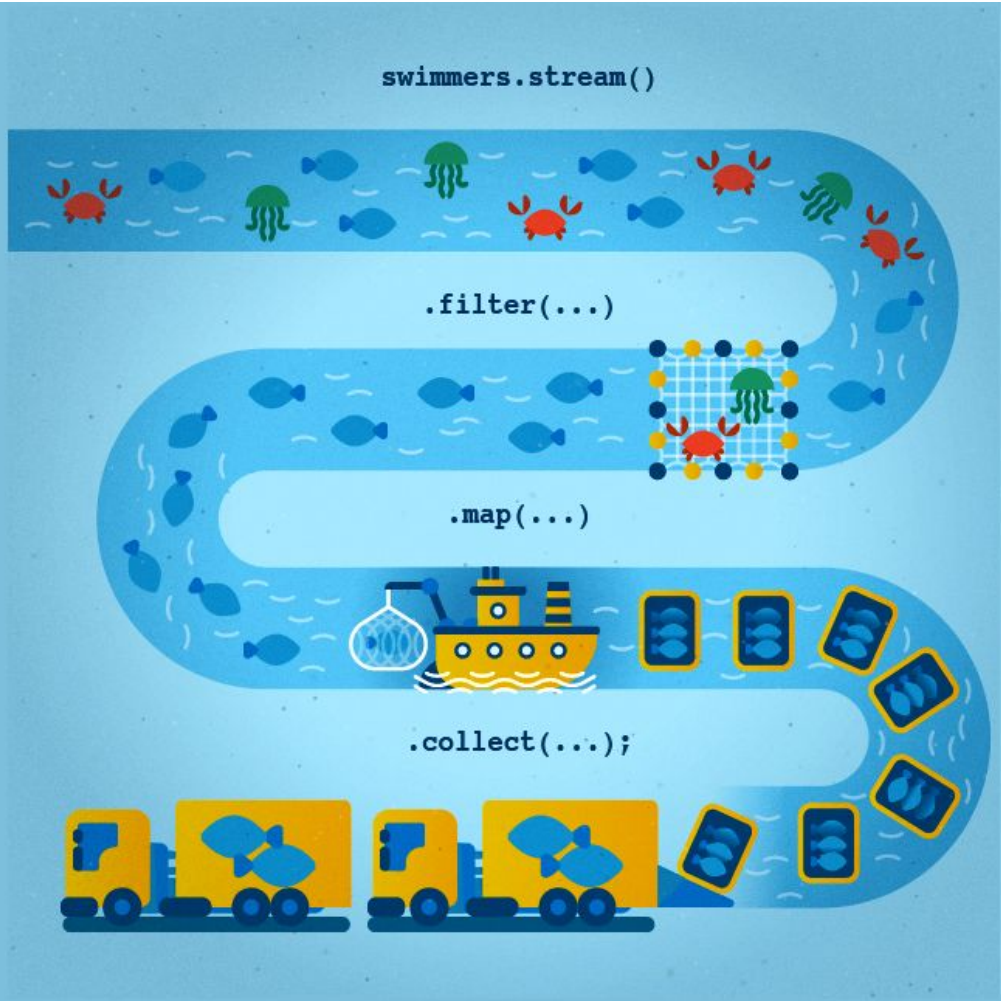
Stream — это объект для универсальной работы с данными. Мы указываем, какие операции хотим провести, при этом не заботясь о деталях реализации.

Данные могут быть получены из источников, коими являются коллекции или методы, поставляющие данные.

К данным затем применяются операторы. Операторы можно разделить на две группы:

- Промежуточные (intermediate) — обрабатывают поступающие элементы и возвращают стрим. Промежуточных операторов в цепочке обработки элементов может быть много.
- Терминальные (terminal) — обрабатывают элементы и завершают работу стрима, так что терминальный оператор в цепочке может быть только один.




swimmers.stream()
.filter(...)
.map(...)
.collect(...);

2. Получение объекта Stream

Для создания/получения объекта объект java.util.stream.Stream существует несколько способов:

- пустой стрим:
  ```
  Stream<String> streamEmpty = Stream.empty();
  ```
- на основе коллекции:
  ```
  Collection<String> collection = Arrays.asList("a", "b", "c");
  Stream<String> streamOfCollection = collection.stream();
  ```
- на основе массива:
  ```
  Stream<String> streamOfArray = Arrays.stream(new String[]{"a", "b", "c"})
  ```
- на основе списка элеменов:
  ```
  Stream<String> streamOfElements = Stream.of("a", "b", "c");
  ```
- с помощью метода Stream.**builder**():
  ```
  Stream<String> streamBuilder =
          Stream.<String>builder().add("a").add("b").add("c").build();
  ```
- с помощью метода Stream.**generate**():
  ```
  Stream<String> streamGenerated =
          Stream.generate(() -> "element").limit(10);
  ```
- с помощью метода Stream.**iterate**():
  ```
  Stream<Integer> streamIterated = Stream.iterate(40, n -> n + 2).limit(20);
  ```

2. Получение объекта Stream

- стрим примитивов:

```
IntStream intStream = IntStream.range(1, 3);
```
- в стриме 2 элемента: 1 и 2
```
LongStream longStream = LongStream.rangeClosed(1, 3);
```
- в стриме 3 элемента: 1, 2 и 3
```
DoubleStream doubleStream = new Random().doubles(3);
```
- в стриме 3 случайных элемента

- стрим из строки:

```
IntStream streamOfChars = "abc".chars();
Stream<String> streamOfString =
        Pattern.compile(", ").splitAsStream("a, b, c");
```

- стрим из файла:

```
Path path = Paths.get("C:\\file.txt");
Stream<String> streamOfStrings = Files.lines(path);
Stream<String> streamWithCharset =
        Files.lines(path, Charset.forName("UTF-8"));
```

3. Как работает стрим

У стримов есть некоторые особенности. Во-первых, обработка не начнётся до тех пор, пока не будет вызван терминальный оператор. list.stream().filter(x -> x > 100); не возьмёт ни единого элемента из списка. Во-вторых, стрим после обработки нельзя переиспользовать.

```
1. Stream<String> stream = list.stream();
2. stream.forEach(System.out::println);
3. stream.filter(s -> s.contains("Stream API"));
4. stream.forEach(System.out::println);
```

Код на второй строке выполнится, а вот на третьей выбросит исключение java.lang.IllegalStateException: stream has already been operated upon or closed.

4. Параллельные стримы

Стримы бывают последовательными (sequential) и параллельными (parallel). Последовательные выполняются только в текущем потоке, а вот параллельные используют общий пул ForkJoinPool.commonPool(). При этом элементы разбиваются (если это возможно) на несколько групп и обрабатываются в каждом потоке отдельно. Затем на нужном этапе группы объединяются в одну для предоставления конечного результата.

Чтобы получить параллельный стрим, нужно либо вызвать метод parallelStream() вместо stream(), либо превратить обычный стрим в параллельный, вызвав промежуточный оператор parallel().

```
1.  list.parallelStream()
2.      .filter(x -> x > 10)
3.      .map(x -> x * 2)
4.      .collect(Collectors.toList());
5.
6.  IntStream.range(0, 10)
7.      .parallel()
8.      .map(x -> x * 10)
9.      .sum();
```

5. Стримы для примитивов

Кроме объектных стримов Stream<T>, существуют специальные стримы для примитивных типов:

- IntStream для int,
- LongStream для long,
- DoubleStream для double.

Для boolean, byte, short и char специальных стримов не придумали, но вместо них можно использовать IntStream, а затем приводить к нужному типу. Для float тоже придётся воспользоваться DoubleStream.

6. Операторы Stream API. Источники.

- empty()
- of(T value)
- of(T... values)
- ofNullable(T t)
- generate(Supplier s)
- iterate(T seed,
  UnaryOperator f)
- iterate(T seed, Predicate hasNext, UnaryOperator f)
- concat(Stream a, Stream b)
- builder()

- IntStream.range(
  int startInclusive,
  int endExclusive)
- LongStream.range(
  long startInclusive,
  long endExclusive)
- IntStream.rangeClosed(
  int startInclusive,
  int endInclusive)
- LongStream.rangeClosed(
  long startInclusive,
  long endInclusive)

6. Операторы Stream API. Источники. of()

of(T value)

of(T... values)

Стрим для одного или нескольких перечисленных элементов.

Пример использования:

Arrays.asList(1, 2, 3).stream() .forEach(System.out::println);	Stream.of(1, 2, 3) .forEach(System.out::println);

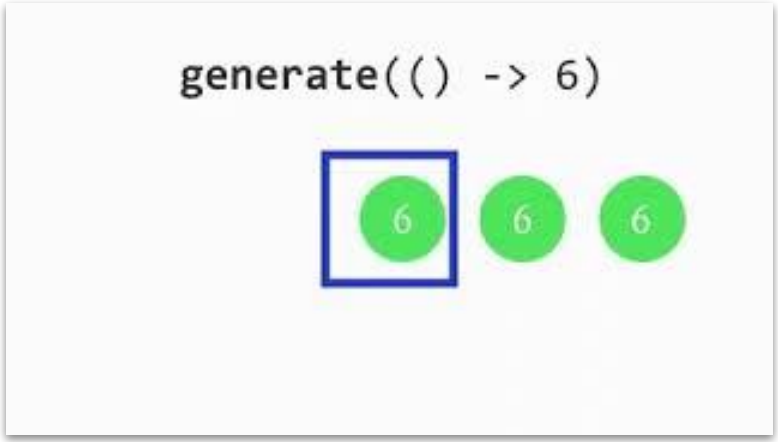
6. Операторы Stream API. Источники. generate()

generate(Supplier s)

Возвращает стрим с бесконечной последовательностью элементов, генерируемых функцией Supplier s.

Пример использования:

```
Stream.generate(() -> 6)
        .limit(6)
        .forEach(System.out::println);
```



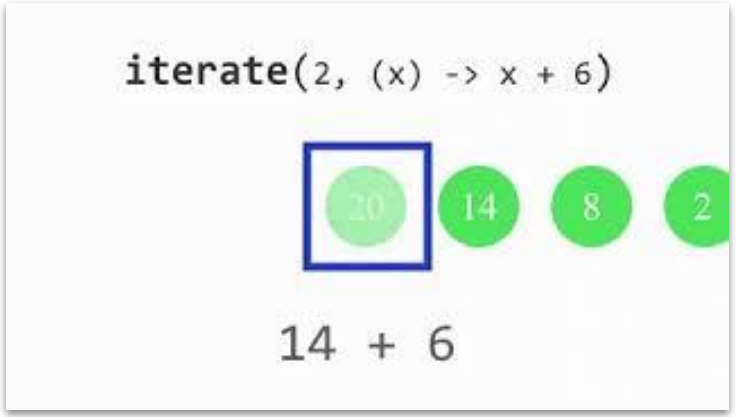
6. Операторы Stream API. Источники. iterate()

iterate(T seed, UnaryOperator f)

Возвращает бесконечный стрим с элементами, которые образуются в результате последовательного применения функции f к итерируемому значению. Первым элементом будет seed, затем f(seed), затем f(f(seed)) и так далее.

Пример использования:

```
Stream.iterate(2, x -> x + 6)
        .limit(6)
        .forEach(System.out::println);
// 2, 8, 14, 20, 26, 32
```



6. Операторы Stream API. Источники. iterate()

iterate(T seed, Predicate hasNext, UnaryOperator f)

Появился в Java 9. Всё то же самое, только добавляется ещё один аргумент hasNext: если он возвращает false, то стрим завершается. Очень похоже на цикл for.

Пример использования:

```
Stream.iterate(2, x -> x < 25, x -> x + 6)
        .forEach(System.out::println);
// 2, 8, 14, 20
```

6. Операторы Stream API. Источники. concat()

concat(Stream a, Stream b)

Объединяет два стрима так, что вначале идут элементы стрима A, а по его окончанию последуют элементы стрима B.

Пример использования:

```
Stream.concat(
        Stream.of(1, 2, 3),
        Stream.of(4, 5, 6))
        .forEach(System.out::println);
// 1, 2, 3, 4, 5, 6
```

6. Операторы Stream API. Источники. builder()

builder()

Создаёт мутабельный объект для добавления элементов в стрим без использования какого-либо контейнера для этого.

Пример использования:

```
Stream.Builder<Integer> streamBuilder = Stream.<Integer>builder()
        .add(0)
        .add(1);
for (int i = 2; i <= 8; i += 2) {
    streamBuilder.accept(i);
}
streamBuilder
    .add(9)
    .build()
    .forEach(System.out::println);
// 0, 1, 2, 4, 6, 8, 9
```

6. Операторы Stream API. Источники. Int/LongStream

- **IntStream.range(int startInclusive, int endExclusive)**
  LongStream.range(long startInclusive, long endExclusive)

  Создаёт стрим из числового промежутка [start..end), то есть от start (включительно) по end.

```
IntStream.range(0, 10).forEach(System.out::println);          // 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
LongStream.range(-10L, -5L).forEach(System.out::println);  // -10, -9, -8, -7, -6
```

- **IntStream.rangeClosed(int startInclusive, int endInclusive)**
  LongStream.rangeClosed(long startInclusive, long endInclusive)

  Создаёт стрим из числового промежутка [start..end], то есть от start (включительно) по end (включительно).

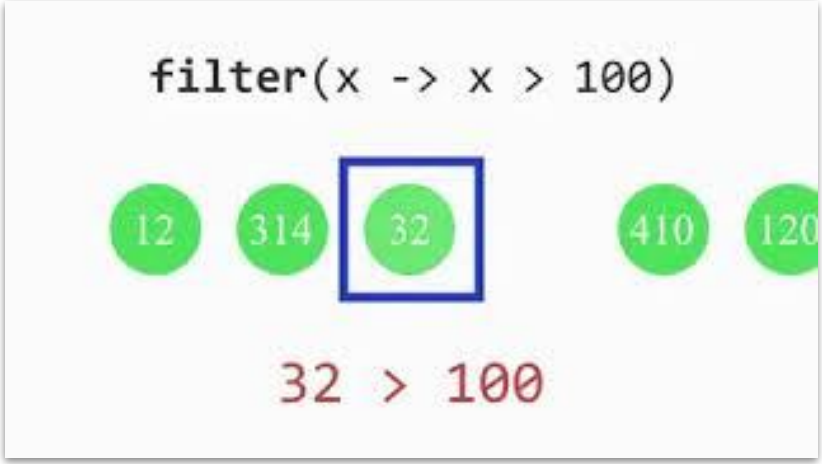
```
IntStream.rangeClosed(0, 5).forEach(System.out::println);        // 0, 1, 2, 3, 4, 5
LongStream.range(-8L, -5L) .forEach(System.out::println); // -8, -7, -6, -5
```

6. Операторы Stream API. Промежуточные операторы. filter()

filter(Predicate predicate)

Фильтрует стрим, принимая только те элементы, которые удовлетворяют заданному условию.

```
Stream.of(1, 2, 3)
    .filter(x -> x == 10)
    .forEach(System.out::print);
// Вывода нет, так как после
фильтрации стрим станет пустым
Stream.of(120, 410, 85, 32, 314, 12)
    .filter(x -> x > 100)
    .forEach(System.out::println);
// 120, 410, 314
```



6. Операторы Stream API. Промежуточные операторы. map()

- **map(Function mapper)**

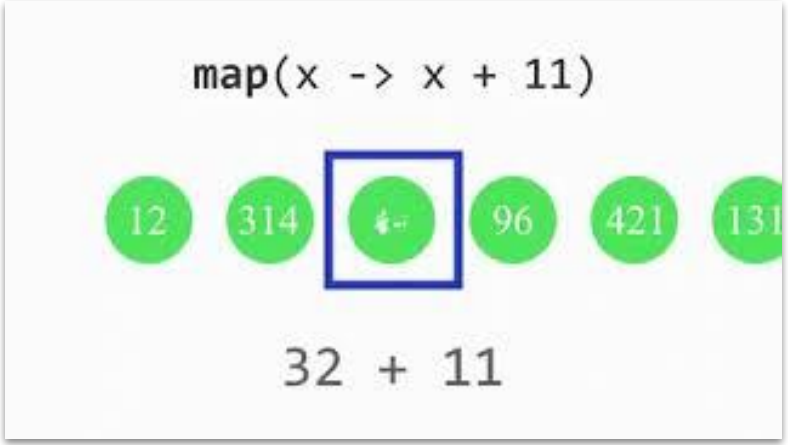
  Применяет функцию к каждому элементу и затем возвращает стрим, в котором элементами будут результаты функции. map можно применять для изменения типа элементов.

- **Stream.mapToDouble(ToDoubleFunction mapper)**
  Stream.mapToInt(ToIntFunction mapper)
  Stream.mapToLong(ToLongFunction mapper)
  IntStream.mapToObj(IntFunction mapper)
  IntStream.mapToLong(IntToLongFunction mapper)
  IntStream.mapToDouble(IntToDoubleFunction mapper)

  Специальные операторы для преобразования объектного стрима в примитивный, примитивного в объектный, либо примитивного стрима одного типа в примитивный стрим другого.

6. Операторы Stream API. Промежуточные операторы. map()

```
Stream.of("3", "4", "5")
    .map(Integer::parseInt)
    .map(x -> x + 10)
    .forEach(System.out::println);
// 13, 14, 15
 Stream.of(120, 410, 85, 32, 314, 12)
    .map(x -> x + 11)
    .forEach(System.out::println);
// 131, 421, 96, 43, 325, 23
```



6. Операторы Stream API. Промежуточные операторы. flatMap()

- **flatMap(Function<T, Stream<R>> mapper)**

  Один из самых интересных операторов. Работает как map, но с одним отличием — можно преобразовать один элемент в ноль, один или множество других.

- **flatMapToDouble(Function mapper)**

  flatMapToInt(Function mapper)

  flatMapToLong(Function mapper)

  Как и в случае с map, служат для преобразования в стрим примитивов.

```
Stream.of(2, 3, 0, 1, 3)
    .flatMap(x -> IntStream.range(0, x))
    .forEach(System.out::println);
// 0, 1, 0, 1, 2, 0, 0, 1, 2
```

6. Операторы Stream API. Промежуточные операторы. limit(), skip()

- **limit(long maxSize)**
  Ограничивает стрим maxSize элементами.
```
Stream.of(120, 410, 85, 32, 314, 12)
    .limit(4)
    .forEach(System.out::println);
// 120, 410, 85, 32
```
- **skip(long n)**
  Пропускает n элементов стрима.
```
Stream.of(5, 10)
    .skip(40)
    .forEach(System.out::println);
// Вывода нет
Stream.of(120, 410, 85, 32, 314, 12)
    .skip(2)
    .forEach(System.out::println);
// 85, 32, 314, 12
```

- **takeWhile(Predicate predicate)**
  Появился в Java 9. Возвращает элементы до тех пор, пока они удовлетворяют условию, то есть функция-предикат возвращает true. Это как limit, только не с числом, а с условием.
```
Stream.of(1, 2, 3, 4, 2, 5)
    .takeWhile(x -> x < 3)
    .forEach(System.out::println);
// 1, 2
```
- **dropWhile(Predicate predicate)**
  Появился в Java 9. Пропускает элементы до тех пор, пока они удовлетворяют условию, затем возвращает оставшуюся часть стрима. Если предикат вернул для первого элемента false, то ни единого элемента не будет пропущено. Оператор подобен skip, только работает по условию.
```
Stream.of(1, 2, 3, 4, 2, 5)
    .dropWhile(x -> x < 3)
    .forEach(System.out::println);
// 3, 4, 2, 5
```

6. Операторы Stream API. Промежуточные операторы. sorted()

- **sorted()**
- **sorted(Comparator comparator)**

Сортирует элементы стрима. Причём работает этот оператор очень хитро: если стрим уже помечен как отсортированный, то сортировка проводиться не будет, иначе соберёт все элементы, отсортирует их и вернёт новый стрим, помеченный как отсортированный.

```
Stream.of(120, 410, 85, 32, 314, 12)
    .sorted()
    .forEach(System.out::println);
// 12, 32, 85, 120, 314, 410
Stream.of(120, 410, 85, 32, 314, 12)
.sorted(Comparator.reversedOrder())
    .forEach(System.out::println);
// 410, 314, 120, 85, 32, 12
```

```
Stream.of(user1, user2, user3)
    .sorted()
    .forEach(System.out::println);
//???
```

6. Операторы Stream API. Промежуточные операторы. distinct()

distinct()

Убирает повторяющиеся элементы и возвращаем стрим с уникальными элементами. Как и в случае с sorted, смотрит, состоит ли уже стрим из уникальных элементов и если это не так, отбирает уникальные и помечает стрим как содержащий уникальные элементы.

```
Stream.of(2, 1, 8, 1, 3, 2)
    .distinct()
    .forEach(System.out::println);
// 2, 1, 8, 3
```

6. Операторы Stream API. Промежуточные операторы. peek()

peek(Consumer action)

Выполняет действие над каждым элементом стрима и при этом возвращает стрим с элементами исходного стрима. Служит для того, чтобы передать элемент куда-нибудь, не разрывая при этом цепочку операторов (вы же помните, что forEach — терминальный оператор и после него стрим завершается?), либо для отладки.

```
Stream.of(0, 3, 0, 0, 5)
    .peek(x -> System.out.format("before distinct: %d%n", x))
    .distinct()
    .peek(x -> System.out.format("after distinct: %d%n", x))
    .map(x -> x * x)
    .forEach(x -> System.out.format("after map: %d%n", x));
```

6. Операторы Stream API. Промежуточные операторы. boxed()

boxed()

Преобразует примитивный стрим в объектный.

```
DoubleStream.of(0.1, Math.PI)   //ещё примитивы
        .boxed()                //уже объекты
        .map(Object::getClass)
        .forEach(System.out::println);
// class java.lang.Double
// class java.lang.Double
```

6. Операторы Stream API. Терминальные операторы. forEach(), forEachOrdered()

void forEach(Consumer action)

Выполняет указанное действие для каждого элемента стрима.

```
Stream.of(120, 410, 85, 32, 314, 12)
    .forEach(x -> System.out.format("%s, ", x));
// 120, 410, 85, 32, 314, 12
```

void forEachOrdered(Consumer action)

Тоже выполняет указанное действие для каждого элемента стрима, но перед этим добивается правильного порядка вхождения элементов. Используется для параллельных стримов, когда нужно получить правильную последовательность элементов.

```
IntStream.range(0, 100000)
    .parallel()
    .filter(x -> x % 10000 == 0)
    .map(x -> x / 10000)
```

```
.forEach(System.out::println);
// 5, 6, 7, 3, 4, 8, 0, 9, 1, 2
```

```
.forEachOrdered(System.out::println);
// 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

6. Операторы Stream API. Терминальные операторы. count()

long count()

Возвращает количество элементов стрима.

```
long count = IntStream.range(0, 10)
    .flatMap(x -> IntStream.range(0, x))
    .count();
System.out.println(count);
// 45
```

6. Операторы Stream API. Терминальные операторы. collect()

R collect(Collector collector)

С помощью этого оператора можно собрать все элементы в список, множество или другую коллекцию, сгруппировать элементы по какому-нибудь критерию, объединить всё в строку и т.д..

```
List<Integer> list = Stream.of(1, 2, 3)
    .collect(Collectors.toList());
// list: [1, 2, 3]
String s = Stream.of(1, 2, 3)
    .map(String::valueOf)
    .collect(Collectors.joining("-", "<", ">"));
// s: "<1-2-3>"
```

6. Операторы Stream API. Терминальные операторы. toArray()

- **Object[] toArray()**

  Возвращает нетипизированный массив с элементами стрима.

- **A[] toArray(IntFunction<A[]> generator)**

  Аналогично, только возвращает типизированный массив.

```
String[] elements = Stream.of("a", "b", "c", "d")
    .toArray(String[]::new);
// elements: ["a", "b", "c", "d"]
```

6. Операторы Stream API. Терминальные операторы. toList()

- **List<T> toList()**

  Наконец-то добавлен в Java 16. Возвращает список, подобно collect(Collectors.toList()). Отличие в том, что теперь возвращаемый список гарантированно нельзя будет модифицировать. Любое добавление или удаление элементов в полученный список будет сопровождаться исключением UnsupportedOperationException.

```
List<String> elements = Stream.of("a", "b", "c", "d")
        .map(String::toUpperCase)
        .toList();
// elements: ["A", "B", "C", "D"]
```

6. Операторы Stream API. Терминальные операторы. reduce()

- **T reduce(T identity, BinaryOperator accumulator)**
- **U reduce(U identity, BiFunction accumulator, BinaryOperator combiner)**

  Ещё один полезный оператор. Позволяет преобразовать все элементы стрима в один объект. Например, посчитать сумму всех элементов, либо найти минимальный элемент.

  Сперва берётся объект identity и первый элемент стрима, применяется функция accumulator и identity становится её результатом. Затем всё продолжается для остальных элементов.

```
int sum = Stream.of(1, 2, 3, 4, 5)
    .reduce(10, (acc, x) -> acc + x);
// sum: 25
```

6. Операторы Stream API. Терминальные операторы. reduce()

- **Optional reduce(BinaryOperator accumulator)**

  Этот метод отличается тем, что у него нет начального объекта identity. В качестве него служит первый элемент стрима. Поскольку стрим может быть пустой и тогда identity объект не присвоится, то результатом функции служит Optional, позволяющий обработать и эту ситуацию, вернув Optional.empty().

```
Optional<Integer> result = Stream.<Integer>empty()
    .reduce((acc, x) -> acc + x);
System.out.println(result.isPresent());
// false
Optional<Integer> sum = Stream.of(1, 2, 3, 4, 5)
    .reduce((acc, x) -> acc + x);
System.out.println(sum.get());
// 15
```

6. Операторы Stream API. Терминальные операторы. min(),max()

- **Optional min(Comparator comparator)**
- **Optional max(Comparator comparator)**

Поиск минимального/максимального элемента, основываясь на переданном компараторе. Внутри вызывается reduce:

```
int min = Stream.of(20, 11, 45, 78, 13)
    .min(Integer::compare).get();
// min: 11

int max = Stream.of(20, 11, 45, 78, 13)
    .max(Integer::compare).get();
// max: 78
```

6. Операторы Stream API. Терминальные операторы. findAny(),findFirst()

- **Optional findAny()**

  Возвращает первый попавшийся элемент стрима. В параллельных стримах это может быть действительно любой элемент, который лежал в разбитой части последовательности..

```
int anySeq = IntStream.range(4, 65536)
    .findAny()
    .getAsInt();
// anySeq: 4
```

- **Optional findFirst()**

  Гарантированно возвращает первый элемент стрима, даже если стрим параллельный.

  Если нужен любой элемент, то для параллельных стримов быстрее будет работать findAny().

```
int firstSeq = IntStream.range(4, 65536)
    .findFirst()
    .getAsInt();
// firstSeq: 4
```

6. Операторы Stream API. Терминальные операторы. all/any/noneMatch()

- **boolean allMatch(Predicate predicate)**
  Возвращает true, если все элементы стрима удовлетворяют условию predicate. Если встречается какой-либо элемент, для которого результат вызова функции-предиката будет false, то оператор перестаёт просматривать элементы и возвращает false.
```
boolean result = Stream.of(1, 2, 3, 4, 5)
    .allMatch(x -> x <= 7);
// result: true
```
- **boolean anyMatch(Predicate predicate)**
  Возвращает true, если хотя бы один элемент стрима удовлетворяет условию predicate. Если такой элемент встретился, нет смысла продолжать перебор элементов, поэтому сразу возвращается результат.
```
boolean result = Stream.of(1, 2, 3, 4, 5)
    .anyMatch(x -> x == 3);
// result: true
```
- **boolean noneMatch(Predicate predicate)**
  Возвращает true, если, пройдя все элементы стрима, ни один не удовлетворил условию predicate. Если встречается какой-либо элемент, для которого результат вызова функции-предиката будет true, то оператор перестаёт перебирать элементы и возвращает false.
```
boolean result = Stream.of(1, 2, 3, 4, 5)
    .noneMatch(x -> x == 9);
// result: true
```

6. Операторы Stream API. Терминальные операторы. average(), sum()

- **OptionalDouble average()**
  Только для примитивных стримов. Возвращает среднее арифметическое всех элементов. Либо Optional.empty, если стрим пуст.
```
double result = IntStream.range(2, 16)
    .average()
    .getAsDouble();
// result: 8.5
```
- **sum()**
  Возвращает сумму элементов примитивного стрима. Для IntStream результат будет типа int, для LongStream — long, для DoubleStream — double.
```
long result = LongStream.range(2, 16)
    .sum();
// result: 119
```

6. Операторы Stream API. Терминальные операторы. summaryStatistics()

- **IntSummaryStatistics summaryStatistics()**
  Полезный метод примитивных стримов. Позволяет собрать статистику о числовой последовательности стрима, а именно: количество элементов, их сумму, среднее арифметическое, минимальный и максимальный элемент.

```
LongSummaryStatistics stats = LongStream.range(2, 16)
    .summaryStatistics();
System.out.format("  count: %d%n", stats.getCount());
System.out.format("    sum: %d%n", stats.getSum());
System.out.format("average: %.1f%n", stats.getAverage());
System.out.format("    min: %d%n", stats.getMin());
System.out.format("    max: %d%n", stats.getMax());
//   count: 14
//     sum: 119
// average: 8,5
//     min: 2
//     max: 15
```

7. Методы класса java.util.stream.Collectors

- **toList()**
  Самый распространённый метод. Собирает элементы в List.
- **toSet()**
  Собирает элементы в множество.
- **toCollection(Supplier collectionFactory)**
  Собирает элементы в заданную коллекцию. Если нужно конкретно указать, какой List, Set или другую коллекцию мы хотим использовать, то этот метод поможет.
- **toMap(Function keyMapper, Function valueMapper)**
  toMap(Function keyMapper, Function valueMapper, BinaryOperator mergeFunction)
  toMap(Function keyMapper, Function valueMapper, BinaryOperator mergeFunction, Supplier mapFactory)
  Собирает элементы в Map. Каждый элемент преобразовывается в ключ и в значение, основываясь на результате функций keyMapper и valueMapper соответственно. Если нужно вернуть тот же элемент, что и пришел, то можно передать Function.identity().
- **toConcurrentMap(Function keyMapper, Function valueMapper)**
  toConcurrentMap(Function keyMapper, Function valueMapper, BinaryOperator mergeFunction)
  toConcurrentMap(Function keyMapper, Function valueMapper, BinaryOperator mergeFunction, Supplier mapFactory)
  Всё то же самое, что и toMap, только работаем с ConcurrentMap.

7. Методы класса java.util.stream.Collectors

- **collectingAndThen(Collector downstream, Function finisher)**
  Собирает элементы с помощью указанного коллектора, а потом применяет к полученному результату функцию.
- **joining()**
  joining(CharSequence delimiter)
  joining(CharSequence delimiter, CharSequence prefix, CharSequence suffix)
  Собирает элементы, реализующие интерфейс CharSequence, в единую строку. Дополнительно можно указать разделитель, а также префикс и суффикс для всей последовательности.

7. Методы класса java.util.stream.Collectors

- **summingInt(ToIntFunction mapper)**
  summingLong(ToLongFunction mapper)
  summingDouble(ToDoubleFunction mapper)
  Коллектор, который преобразовывает объекты в int/long/double и подсчитывает сумму.

- **averagingInt(ToIntFunction mapper)**
  averagingLong(ToLongFunction mapper)
  averagingDouble(ToDoubleFunction mapper)
  Аналогично, но со средним значением.

- **summarizingInt(ToIntFunction mapper)**
  summarizingLong(ToLongFunction mapper)
  summarizingDouble(ToDoubleFunction mapper)
  Аналогично, но с полной статистикой.

- **counting()**
  Подсчитывает количество элементов.

7. Методы класса java.util.stream.Collectors

- **filtering(Predicate predicate, Collector downstream)**
  mapping(Function mapper, Collector downstream)
  flatMapping(Function downstream)
  reducing(BinaryOperator op)
  reducing(T identity, BinaryOperator op)
  reducing(U identity, Function mapper, BinaryOperator op)
  Специальная группа коллекторов, которая применяет операции filter, map, flatMap и reduce. filtering и flatMapping появились в Java 9.
- **minBy(Comparator comparator)**
  maxBy(Comparator comparator)
  Поиск минимального/максимального элемента, основываясь на заданном компараторе.

7. Методы класса java.util.stream.Collectors

- **groupingBy(Function classifier)**
  groupingBy(Function classifier, Collector downstream)
  groupingBy(Function classifier, Supplier mapFactory, Collector downstream)
  Группирует элементы по критерию, сохраняя результат в Map. Вместе с представленными выше агрегирующими коллекторами, позволяет гибко собирать данные.

- **groupingByConcurrent(Function classifier)**
  groupingByConcurrent(Function classifier, Collector downstream)
  groupingByConcurrent(Function classifier, Supplier mapFactory, Collector downstream)
  Аналогичный набор методов, только сохраняет в ConcurrentMap.

- **partitioningBy(Predicate predicate)**
  partitioningBy(Predicate predicate, Collector downstream)
  Ещё один интересный метод. Разбивает последовательность элементов по какому-либо критерию. В одну часть попадают все элементы, которые удовлетворяют переданному условию, во вторую — все, которые не удовлетворяют.

8. Интерфейс java.util.stream.Collector

Интерфейс java.util.stream.Collector служит для сбора элементов стрима в некоторый мутабельный контейнер. Он состоит из таких методов:
- Supplier<A> supplier() — функция, которая создаёт экземпляры контейнеров.
- BiConsumer<A,T> accumulator() — функция, которая кладёт новый элемент в контейнер.
- BinaryOperator<A> combiner() — функция, которая объединяет два контейнера в один. В параллельных стримах каждая часть может собираться в отдельный экземпляр контейнера и в итоге необходимо их объединять в один результирующий.
- Function<A,R> finisher() — функция, которая преобразовывает весь контейнер в конечный результат. Например, можно обернуть List в Collections.unmodifiableList.
- Set<Characteristics> characteristics() — возвращает характеристики коллектора, чтобы внутренняя реализация знала, с чем имеет дело. Например, можно указать, что коллектор поддерживает многопоточность.

https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.html

9. Spliterator

Элементы стримов нужно не только итерировать, но ещё и разделять на части и отдавать другим потокам. За итерацию и разбиение отвечает Spliterator. Он даже звучит как Iterator, только с приставкой Split — разделять.

Методы интерфейса:
- trySplit — как следует из названия, пытается разделить элементы на две части. Если это сделать не получается, либо элементов недостаточно для разделения, то вернёт null. В остальных случаях возвращает ещё один Spliterator с частью данных.
- tryAdvance(Consumer action) — если имеются элементы, для которых можно применить действие, то оно применяется и возвращает true, в противном случае возвращается false, но действие не выполняется.
- estimateSize() — возвращает примерное количество элементов, оставшихся для обработки, либо Long.MAX_VALUE, если стрим бесконечный или посчитать количество невозможно.
- characteristics() — возвращает характеристики сплитератора.

10. Советы и best practices

1. Если задачу не получается красиво решить стримами, не решайте её стримами.
2. Если задачу не получается красиво решить стримами, не решайте её стримами!
3. Если задача уже красиво решена НЕ стримами, всё работает и всех всё устраивает, не перерешивайте её стримами!
4. В большинстве случаев нет смысла сохранять стрим в переменную. Используйте цепочку вызовов методов (method chaining).

```
// Нечитабельно
Stream<Integer> stream = list.stream();
stream = stream.filter(x -> x > 2);
stream.forEach(System.out::println);
// Так лучше
list.stream()
        .filter(x -> x > 2)
        .forEach(System.out::println);
```

10. Советы и best practices

5. Старайтесь сперва отфильтровать стрим от ненужных элементов или ограничить его, а потом выполнять преобразования.
6. Не используйте параллельные стримы везде, где только можно. Затраты на разбиение элементов, обработку в другом потоке и последующее их слияние порой больше, чем выполнение в одном потоке.
7. При использовании параллельных стримов, убедитесь, что нигде нет блокирующих операций или чего-то, что может помешать обработке элементов.

```
list.parallelStream()
        .filter(s -> isFileExists(hash(s)))
        ...
```

10. Советы и best practices

8. Если где-то в модели вы возвращаете копию списка или другой коллекции, то подумайте о замене на стримы. Например:

```
// Было
class Model {
    private final List<String> data;
    public List<String> getData() {
        return new ArrayList<>(data);
    }
}
```

```
// Стало
class Model {
    private final List<String> data;
    public Stream<String> dataStream()
{
        return data.stream();
    }
}
```

Теперь есть возможность получить не только список **model.dataStream().collect(toList());**, но и множество, любую другую коллекцию, отфильтровать что-то, отсортировать и так далее. Оригинальный List<String> data так и останется нетронутым.

Д/З

1.Скачать и установить сборщик проектов **Maven** (Инструкция)

2.Скачать архив с программой

3.Запустить метод main.

Ссылки:

1. java.util.stream.Stream
2. java.util.stream.Collector
3. The Java 8 Stream API Tutorial (Baeldung)
4. Java 8 Stream API: шпаргалка для программиста
5. Java Stream API. Копилка рецептов (Skillbox)
6. https://struchkov.dev/blog/ru/java-stream-api/