



Java Levelup #7

Что мертво — умереть не может

Проекты



ВНЕДРЕНИЕ ЗАВИСИМОСТЕЙ



Dependency Injection

```
public class MyDao {  
    //в оригинале: protected DataSource dataSource =  
    private DataSource dataSource =  
        new DataSourceImpl("driver", "url", "user", "password");  
  
    //data access methods...  
    public Person readPerson(int primaryKey) {...}  
}
```

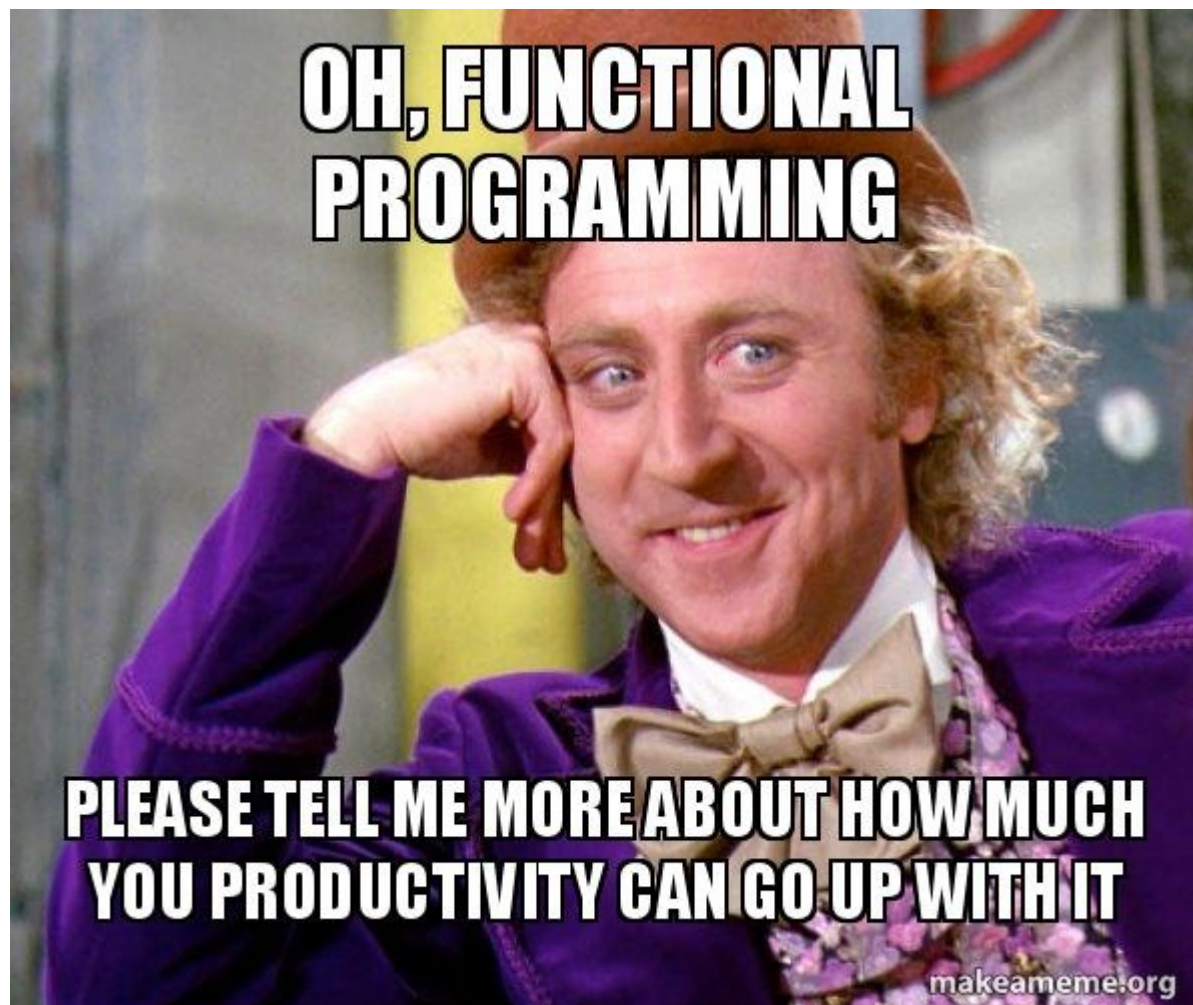
Dependency Injection

```
public class MyDao {  
  
    //в оригинале: protected DataSource dataSource = null;  
    private final DataSource dataSource;  
  
    public MyDao(String driver, String url, String user, String password){  
        this.dataSource = new DataSourceImpl(driver, url, user, password);  
    }  
  
    //data access methods...  
    public Person readPerson(int primaryKey) {...}  
  
}
```

Dependency Injection

```
public class MyDao {  
  
    //в оригинале:  protected DataSource dataSource = null;  
    private final DataSource dataSource;  
  
    public MyDao(DataSource dataSource){  
        this.dataSource = dataSource;  
    }  
  
    //data access methods...  
    public Person readPerson(int primaryKey) {...}  
  
}
```

Функциональщина :)





Приготовим фарш

Императивное программирование



```
public Forcemeat cookForcemeat(Money babki){  
  
    Meat meat = buyMeat(babki);  
    ChoppedMeat chopped = chop(meat);  
    Forcemeat result = rub(chopped);  
    return result;  
  
}
```


Приготовим фарш

функціонально

```
public Function<Money, Forcemeat> cookForcemeat(){  
    return buyMeat.andThen(chop).andThen(rub);  
}
```

```
Forcemeat result = cookForcemeat().apply(money);
```



В чем суть?

1. Immutable данные
2. Разделяем данные и функциональность (мухи и котлеты)
3. Ленивые вычисления
4. Чистые функции (тестирование)
5. Функции высшего порядка (функции принимающие другие функции как аргументы, возвращающие функции)
6. рекурсии
7. pattern matching
8. карирование



И нафига?

1. Уменьшаем количество состояний
2. Используем ленивые вычисления
3. Тестируем
4. Многопоточность
5. Параллельные вычисления
6. Non blocking IO
7. Скомпилилось - работает

99. Писать клевый код который никто не поймет



Лямбды

```
interface Function1<T, A, B> {
    T apply(A kek, B lol);
}

interface Function2 {
    Boolean apply(int kek, int lol);
}

public class Kek {
    //static void cmpArrayEls(int[] arr, Function1<Boolean, Integer, Integer> fun)
    static void cmpArrayEls(int[] arr, Function2 fun) {
        for(int i = 0; i < arr.length - 1; i++) {
            Boolean cmp = fun.apply(arr[i], arr[i+1]);
            System.out.println(arr[i] + " " + arr[i + 1] + " " + cmp);
            // DO SOME KEK...
        }
    }

    public static void main(String[] args) {
        int[] arr = new int[]{1, 2, 3, 4, 5, 6};

        cmpArrayEls(arr, new Function2() {
            public Boolean apply(int kek, int lol) {
                return kek > lol;
            }
        });

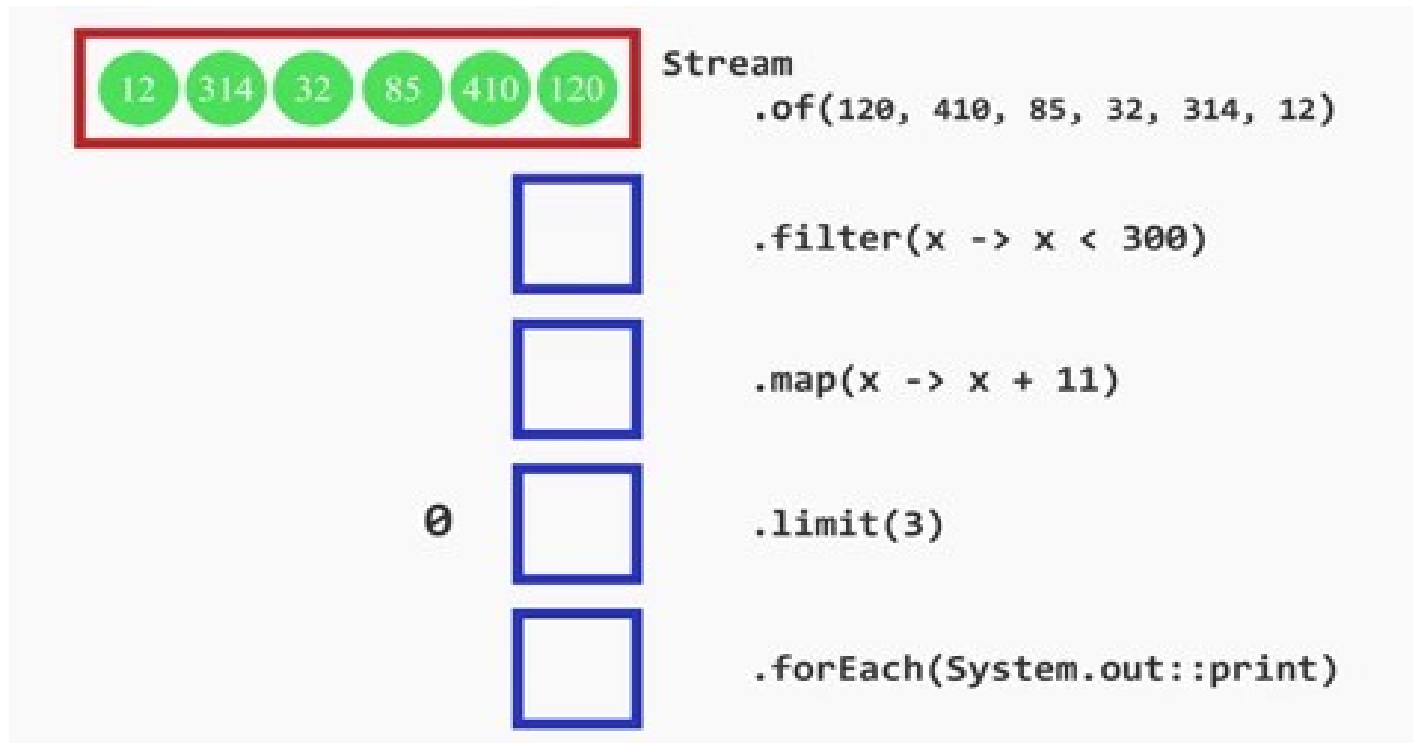
        cmpArrayEls(arr, (int kek, int lol) -> kek > lol);
        cmpArrayEls(arr, (kek, lol) -> kek > lol);
    }
}
```

Stream API

СОЗДАНИЕ:

- - Пустой стрим: `Stream.empty() // Stream<String>`
- - Стрим из List: `list.stream() // Stream<String>`
- - Стрим из Map: `map.entrySet().stream() // Stream<Map.Entry<String, String>>`
- - Стрим из массива: `Arrays.stream(array) // Stream<String>`
- - Стрим из указанных элементов: `Stream.of("a", "b", "c") // Stream<String>`

Stream API



Parallel Stream

```
list.parallelStream()  
    .filter(x -> x > 10)  
    .map(x -> x * 2)  
    .collect(Collectors.toList());
```

```
IntStream.range(0, 10)  
    .parallel()  
    .map(x -> x * 10)  
    .sum();
```



Стримы для примитивов:

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

Generate

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

```
Stream.generate(() -> 4)
    .limit(4)
    .forEach(System.out::println);
// 4, 4, 4, 4, 4, 4, ...
```

Iterate

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

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

Iterate

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

concat

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

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


concat

concat(streamA, streamB)



builder()

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

```
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)
    .add(10)
    .build()
    .forEach(System.out::println);
// 0, 1, 2, 4, 6, 8, 9, 10
```

range

- Создаёт стрим из числового промежутка [start..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
```

rangeClosed

- Создаёт стрим из числового промежутка [start..end]

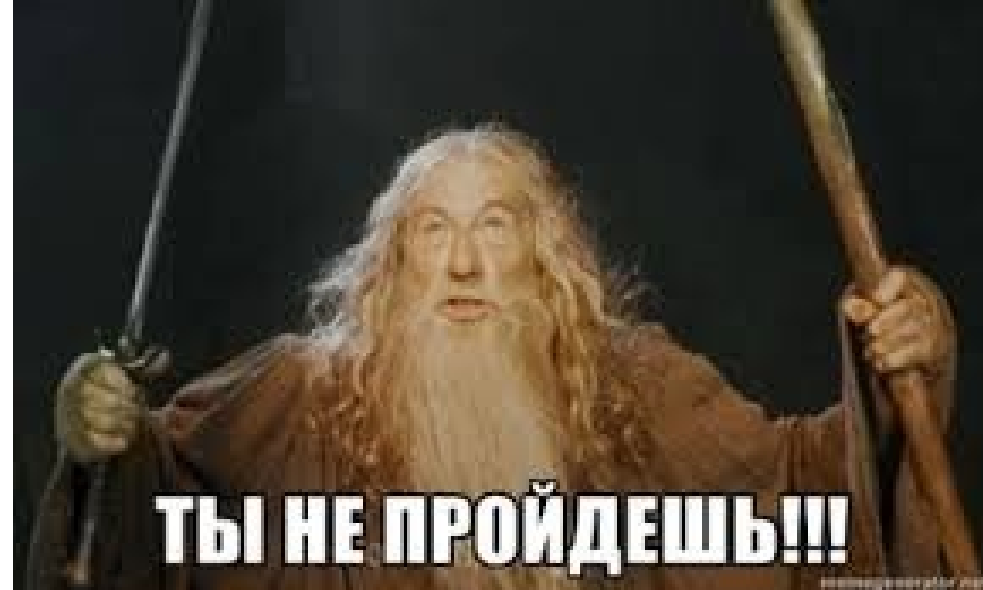
```
IntStream.rangeClosed(0, 5)
    .forEach(System.out::println);
// 0, 1, 2, 3, 4, 5
```

```
LongStream.range(-8L, -5L)
    .forEach(System.out::println);
// -8, -7, -6, -5
```



Промежуточные операторы

filter

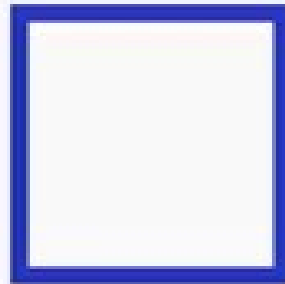


- Фильтрует стрим, принимая только те элементы, которые удовлетворяют заданному условию.

```
Stream.of(120, 410, 85, 32, 314, 12)
    .filter(x -> x > 100)
    .forEach(System.out::println);
// 120, 410, 314
```




```
filter(x -> x > 100)
```



map

- 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)`

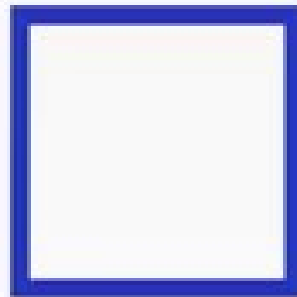
`IntStream.mapToLong(IntToLongFunction mapper)`

`IntStream.mapToDouble(IntToDoubleFunction mapper)`

- для примитивов



```
map(x -> x + 11)
```



flatMap

- Работает как map, но с одним отличием — можно преобразовать один элемент в ноль, один или множество других.

`flatMapToDouble(Function mapper)`

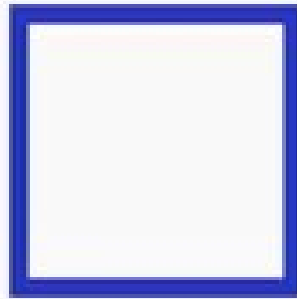
`flatMapToInt(Function mapper)`

`flatMapToLong(Function mapper)`

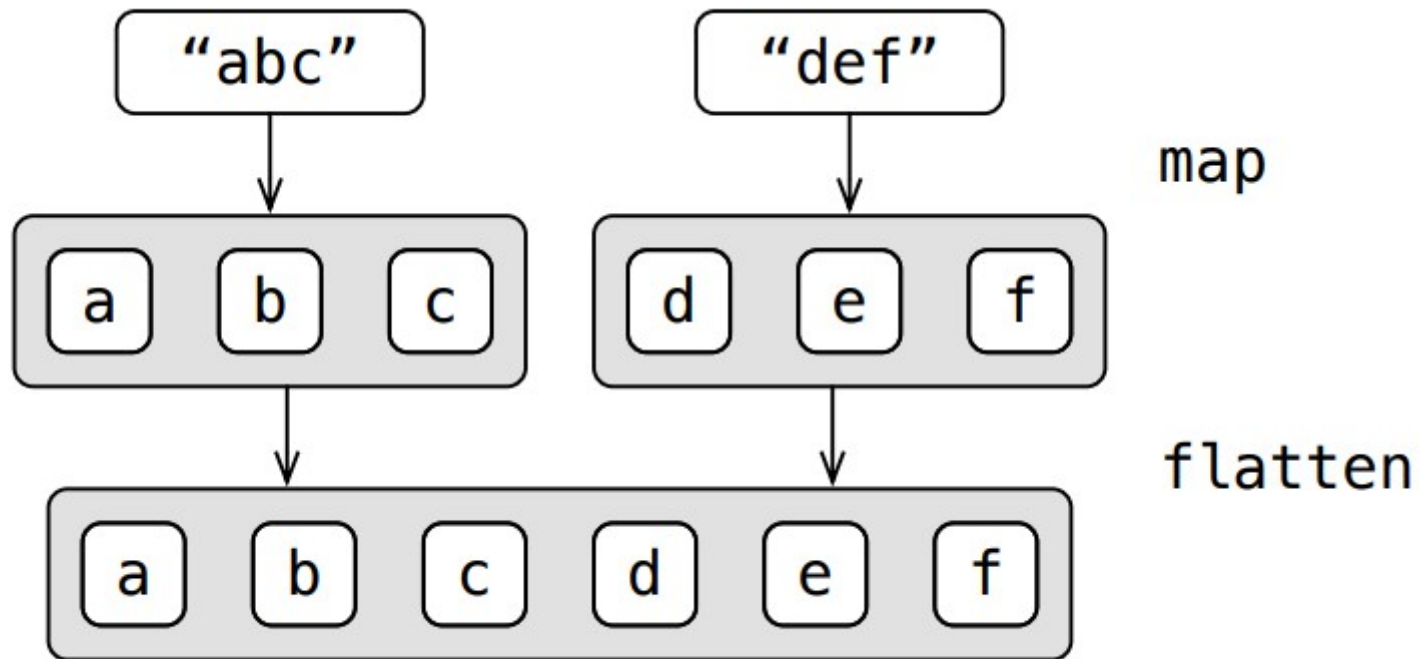
- Как и в случае с map, служат для преобразования в примитивный стрим.



```
flatMap(x -> Stream.range(0, x))
```



flatMap

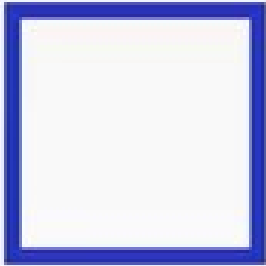


limit(long maxSize)

- Ограничивает стрим maxSize элементами.

```
Stream.of(120, 410, 85, 32, 314, 12)
    .limit(4)
    .forEach(System.out::println);
// 120, 410, 85, 32
```

`limit(4)`



skip(long n)

- Пропускает n элементов стрима.

```
Stream.of(5, 10)
```

```
.skip(40)
```

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

```
// Вывода нет
```

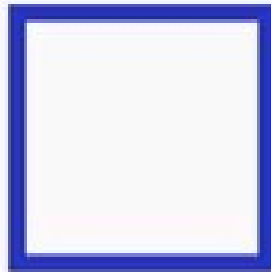
```
Stream.of(120, 410, 85, 32, 314, 12)
```

```
.skip(2)
```

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

```
// 85, 32, 314, 12
```

`skip(2)`



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.

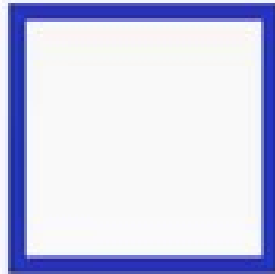
`sorted()`



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.

distinct()



context: {}



peek

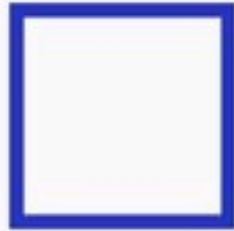
- Выполняет действие над каждым элементом стрима и при этом возвращает стрим с элементами исходного стрима.
- В отличие от `forEach`, не завершает стрим

peek

```
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));
```



```
peek(x -> System.out.format("%s, ", x))
```



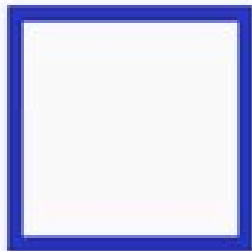


takeWhile

- 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.



```
takeWhile(x -> x < 3)
```

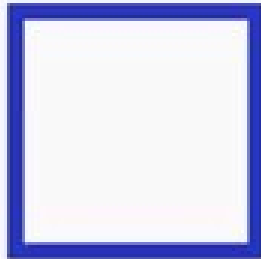


dropWhile

- 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.



```
dropWhile(x -> x < 3)
```



boxed

- Преобразует примитивный стрим в объектный.

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




Терминальные операторы

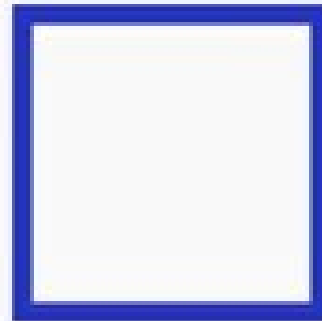


forEach

- Выполняет указанное действие для каждого элемента стрима.



```
forEach(x -> System.out.format("%s, ", x))
```





forEachOrdered

- 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
```

```
IntStream.range(0, 100000)
    .parallel()
    .filter(x -> x % 10000 == 0)
    .map(x -> x / 10000)
    .forEachOrdered(System.out::println);
// 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

count

- Возвращает количество элементов стрима

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

collect (Collector collector)

- Собирает стрим в какую-либо коллекцию или другой контейнер

```
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>"
```

R collect(Supplier supplier, BiConsumer accumulator, BiConsumer combiner)

```
List<String> list = Stream.of("a", "b", "c", "d")  
    .collect(ArrayList::new, ArrayList::add, ArrayList::addAll);  
// list: ["a", "b", "c", "d"]
```


toArray()

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

reduce

- T reduce(T identity, BinaryOperator accumulator)
- Позволяет преобразовать все элементы стрима в один объект. Например, посчитать сумму всех элементов, либо найти минимальный элемент.

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

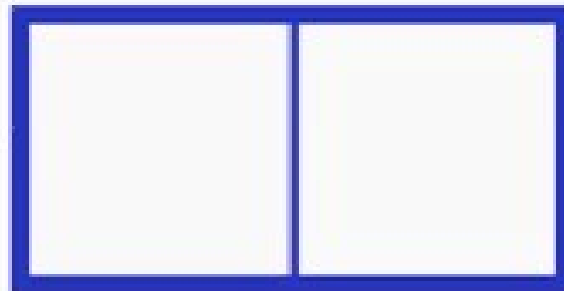
reduce

- Optional reduce(BinaryOperator accumulator)
- Identity — первый элемент
- Если стрим пустой — 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
```

reduce((acc, x) \rightarrow acc + x)



min/max

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

- Эквивалентно:

```
reduce((a, b) -> comparator.compare(a, b)  
    <= 0 ? a : b));
```

```
reduce((a, b) -> comparator.compare(a, b)  
    >= 0 ? a : b));
```


min/max

```
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
```

find

- Optional `findAny()`
- Возвращает первый попавшийся элемент стрима. В параллельных стримах это может быть действительно любой элемент, который лежал в разбитой части последовательности.
- Optional `findFirst()`
- Гарантированно возвращает первый элемент стрима, даже если стрим параллельный.



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

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

```
int anyParallel = IntStream.range(4, 65536)
    .parallel()
    .findAny()
    .getAsInt();
// anyParallel: 32770
```

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

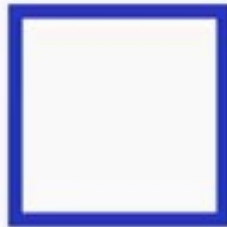



allMatch

- `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`.

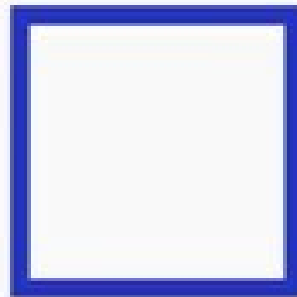


```
allMatch(x -> x <= 7)
```





```
allMatch(x -> x < 3)
```



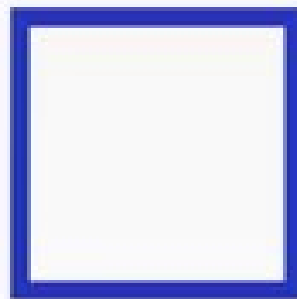


anyMatch

- Возвращает true, если хотя бы один элемент стрима удовлетворяет условию predicate.



```
anyMatch(x -> x == 3)
```



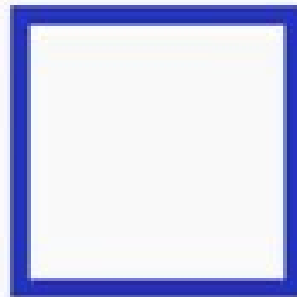


noneMatch

- `boolean noneMatch(Predicate predicate)`
- Возвращает `true`, если, пройдя все элементы стрима, ни один не удовлетворил условию `predicate`.



```
noneMatch(x -> x == 3)
```



average

- 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
```

summaryStatistics

- IntSummaryStatistics summaryStatistics()
- 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
```

Методы Collectors

- toMap(Function keyMapper, Function valueMapper)

```
Map<Character, String> map3 = Stream.of(50, 54, 55)
    .collect(Collectors.toMap(
        i -> (char) i.intValue(),
        i -> String.format("<%d>", i)
    ));
// {'2'="<50>", '6'="<54>", '7'="<55>"}
```

- toMap(Function keyMapper, Function valueMapper, BinaryOperator mergeFunction)
 - При дубликate можно объединить значения

Методы Collectors

- `toList()`
- `toSet()`
- `toCollection(Supplier collectionFactory)`

```
Deque<Integer> deque = Stream.of(1, 2, 3, 4, 5)
    .collect(Collectors.toCollection(ArrayDeque::new));

Set<Integer> set = Stream.of(1, 2, 3, 4, 5)
    .collect(Collectors.toCollection(LinkedHashSet::new));
```

Методы Collectors

- toMap(Function keyMapper, Function valueMapper, BinaryOperator mergeFunction, Supplier mapFactory)

```
Map<Character, String> map3 = Stream.of(50, 54, 55)
    .collect(Collectors.toMap(
        i -> (char) i.intValue(),
        i -> String.format("<%d>", i)
    ));
// {'2'="<50>", '6'="<54>", '7'="<55>"}
```

Методы Collectors

- `toMap(Function keyMapper, Function valueMapper, BinaryOperator mergeFunction)`

Аналогичен первой версии метода, только в случае, когда встречается два одинаковых ключа, позволяет объединить значения.

Методы Collectors

- toMap(Function keyMapper, Function valueMapper, BinaryOperator mergeFunction, Supplier mapFactory)

```
Map<Integer, String> map5 = Stream.of(50, 55, 69, 20, 19, 52)
    .collect(Collectors.toMap(
        i -> i % 5,
        i -> String.format("<%d>", i),
        (a, b) -> String.join(", ", a, b),
        LinkedHashMap::new
    ));
// {0=<50>, <55>, <20>, 4=<69>, <19>, 2=<52>}
```

Методы Collectors

- `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`.

Методы Collectors

- `collectingAndThen(Collector downstream, Function finisher)`

Собирает элементы с помощью указанного коллектора, а потом применяет к полученному результату функцию.

```
List<Integer> list = Stream.of(1, 2, 3, 4, 5)
    .collect(Collectors.collectingAndThen(
        Collectors.toList(),
        Collections::unmodifiableList));
System.out.println(list.getClass());
// class java.util.Collections$UnmodifiableRandomAccessList
```

Методы Collectors

- joining()
- joining(CharSequence delimiter)
- joining(CharSequence delimiter, CharSequence prefix, CharSequence suffix)

```
String s1 = Stream.of("a", "b", "c", "d")
    .collect(Collectors.joining());
System.out.println(s1);
// abcd
```

```
String s2 = Stream.of("a", "b", "c", "d")
    .collect(Collectors.joining("-"));
System.out.println(s2);
// a-b-c-d
```

```
String s3 = Stream.of("a", "b", "c", "d")
    .collect(Collectors.joining(" -> ", "[ ", " ]"));
System.out.println(s3);
// [ a -> b -> c -> d ]
```

Методы Collectors

- `summingInt(ToIntFunction mapper)`
- `summingLong(ToLongFunction mapper)`
- `summingDouble(ToDoubleFunction mapper)`
- преобразовывает объекты в `int/long/double` и подсчитывает сумму.
- `averagingInt(ToIntFunction mapper)`
- ...
`summarizingInt(ToIntFunction mapper)`
- ...
- Аналогично, но с полной статистикой.

Методы Collectors

- `counting()` - кол-во
- `filtering(Predicate predicate, Collector downstream)`
- `mapping(Function mapper, Collector downstream)`
- `flatMap(Function downstream)`
- `reducing(BinaryOperator op)`
- `reducing(T identity, BinaryOperator op)`
- `reducing(U identity, Function mapper, BinaryOperator op)`

Методы Collectors

```
List<Integer> ints = Stream.of(1, 2, 3, 4, 5, 6)
    .collect(Collectors.filtering(
        x -> x % 2 == 0,
        Collectors.toList()));
// 2, 4, 6
```

```
String s1 = Stream.of(1, 2, 3, 4, 5, 6)
    .collect(Collectors.filtering(
        x -> x % 2 == 0,
        Collectors.mapping(
            x -> Integer.toString(x),
            Collectors.joining("-"))
    ));
// 2-4-6
```

Методы Collectors

- minBy(Comparator comparator)
- maxBy(Comparator comparator)
- Поиск минимального/максимального элемента, основываясь на заданном компараторе.

```
Optional<String> min = Stream.of("ab", "c", "defgh", "ijk", "l")  
    .collect(Collectors.minBy(Comparator.comparing(String::length)));  
min.ifPresent(System.out::println);  
// c
```

Методы Collectors

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


Методы Collectors

```
Map<Integer, List<String>> map1 = Stream.of(
    "ab", "c", "def", "gh", "ijk", "l", "mnop")
    .collect(Collectors.groupingBy(String::length));
map1.entrySet().forEach(System.out::println);
// 1=[c, l]
// 2=[ab, gh]
// 3=[def, ijk]
// 4=[mnop]
```



```
List<Student> students = Arrays.asList(
    new Student("Alex", Speciality.Physics, 1),
    new Student("Rika", Speciality.Biology, 4),
    new Student("Julia", Speciality.Biology, 2),
    new Student("Steve", Speciality.History, 4),
    new Student("Mike", Speciality.Finance, 1),
    new Student("Hinata", Speciality.Biology, 2),
    new Student("Richard", Speciality.History, 1),
    new Student("Kate", Speciality.Psychology, 2),
    new Student("Sergey", Speciality.ComputerScience, 4),
    new Student("Maximilian", Speciality.ComputerScience, 3),
    new Student("Tim", Speciality.ComputerScience, 5),
    new Student("Ann", Speciality.Psychology, 1)
);

enum Speciality {
    Biology, ComputerScience, Economics, Finance,
    History, Philosophy, Physics, Psychology
}
```



```
students.stream()  
    .collect(Collectors.groupingBy(Student::getYear))  
    .entrySet().forEach(System.out::println);  
  
// 1=[Alex: Physics 1, Mike: Finance 1, Richard: History 1, Ann: Psychology 1]  
// 2=[Julia: Biology 2, Hinata: Biology 2, Kate: Psychology 2]  
// 3=[Maximilian: ComputerScience 3]  
// 4=[Rika: Biology 4, Steve: History 4, Sergey: ComputerScience 4]  
// 5=[Tim: ComputerScience 5]
```


Методы Collectors

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

Методы Collectors

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

```
Map<Boolean, List<String>> map1 = Stream.of(
    "ab", "c", "def", "gh", "ijk", "l", "mnop")
    .collect(Collectors.partitioningBy(s -> s.length() <= 2));
map1.entrySet().forEach(System.out::println);
// false=[def, ijk, mnop]
// true=[ab, c, gh, l]
```



**Можно записать свой
Collector :)**



I'M A SIMPLE MAN

**I SEE A FUNCTIONAL PROGRAMMING
COURSE, I TAKE IT**

imgflip.com