

# Методические указания 8

# Java Stream API

Функциональное программирование в Java

[Что такое Java Stream API?](#)

[Способы создания и виды стримов](#)

[Порядок обработки](#)

[Почему порядок работы имеет значение?](#)

[Параллельные стримы](#)

[Практическое задание](#)

[Дополнительные материалы](#)

[Используемая литература](#)

## Что такое Java Stream API?

Чтобы начать изучение Java Stream API, разберём основные определения. В языке Java есть понятие потоков, но ни классы **InputStream** (поток ввода) и **OutputStream** (поток вывода), ни **Thread** (поток исполнения) не имеют ничего общего с новшеством Java 8 — Stream API. Stream API работает не с потоком в прямом смысле слова, а с цепочкой функций, вызываемых из самих себя. Он обеспечивает функциональное программирование в Java 8. Поток — это последовательность элементов и функций, которые поддерживают различные виды операций. Чтобы не путаться в понятиях, обозначим Stream API как стрим.

```

public class StreamApp {
    static class Person {
        enum Position {
            ENGINEER, DIRECTOR, MANAGER;
        }

        private String name;
    private int age;
        private Position position;

        public Person (String name, int age, Position
position) {
            this .name = name;                this
.age = age;
            this .position = position;
        }
    }

    private static void streamSimpleTask () {
        List<Person> persons = new ArrayList<>(Arrays.asList(
new Person( "Bob1" , 35, Person.Position.MANAGER),
new Person( "Bob2" , 44, Person.Position.DIRECTOR),
new Person( "Bob3" , 25, Person.Position.ENGINEER),
new Person( "Bob4" , 42, Person.Position.ENGINEER),
new Person( "Bob5" , 55, Person.Position.MANAGER),
new Person( "Bob6" , 19, Person.Position.MANAGER),
new Person( "Bob7" , 33, Person.Position.ENGINEER),
new Person( "Bob8" , 37, Person.Position.MANAGER)
));

        List<String> engineersNames = persons.stream()
            .filter(person -> person.position == Person.Position.ENGINEER)
            .sorted((o1, o2) -> o1.age - o2.age)
            .map((Function<Person, String>) person -> person.name)
            .collect(Collectors.toList());

        System.out.println(engineersNames);
    }
}

```

В примере выше с помощью Stream API мы можем в одну строку решить такую задачу: из списка сотрудников, выбрать только инженеров, затем отсортировать их в порядке увеличения возрастов, получить имена этих сотрудников в том же порядке и сохранить их в коллекцию **List<String>** .

Операции со стримами могут относиться к терминальным или промежуточным. Все промежуточные операции возвращают модифицированный стрим, так что их можно объединять. Терминальные операции возвращают либо результат, не относящийся к стриму, либо ничего (**void** ). В приведённом листинге **filter()** , **map()** и **sorted()** — промежуточные операции. А **collect()** — терминальная операция и может возвращать коллекцию. Полный список таких операций доступен в **JavaDoc** к Java 8. Большинство операций стрима могут принимать лямбда-выражения.

## Способы создания и виды стримов

Стримы создаются из различных источников данных, но в большинстве случаев — из коллекций **List** и **Set** с помощью метода **stream()** :

```
public static void main (String[] args) {
    List<String> list = new ArrayList<>(Arrays.asList( "A" ,
    "AB" ,  "B" ));    Stream<String> stream = list.stream(); }
```

Для создания стрима, состоящего из произвольного набора элементов, можно использовать статический метод **Stream.of()** :

```
public static void main (String[] args) {
    Stream<Integer> stream = Stream.of( 1, 2, 3, 4) ;
}
```

Чтобы создать стрим из существующего массива, можно воспользоваться статическим методом класса **Arrays.stream()** или передать массив в качестве аргумента методу **Stream.of()** :

```
public static void main (String[] args) {
    String[] array = { "A" ,  "B" ,  "C" };
    Stream<String> stream = Arrays.stream(array);
    Stream<String> anotherStream = Stream.of(array); }
```

Помимо обычного **Stream** в Java 8 можно создавать «специализированные» стримы (**IntStream** , **DoubleStream** , **LongStream**) :

```
public static void main (String[] args) {
    IntStream intStream = IntStream.of( 1, 2, 3, 4) ;
    LongStream longStream = LongStream.of( 1L , 2L , 3L ,
    4L );
    IntStream rangedIntStream = IntStream.rangeClosed( 1,
    100) ; }
```

В примере выше мы создаём два стрима, состоящих из набора только Integer- и только Long-объектов. Третий стрим создаётся с помощью статического метода **rangeClosed()** и будет состоять из чисел от 1 до 100 включительно.

В **IntStream** также можно преобразовать обычный стрим:

```
public static void main (String[] args) {
    IntStream intStream = Stream.of( 1, 2, 3, 4)
    .mapToInt(n -> n); }
```

## Базовые операции

Рассмотрим несколько базовых операций, с которыми придется столкнуться при работе с Stream API.

### Терминальные операции

Первая терминальная операция — это **collect()** , предназначенная для превращения элементов стрима в коллекцию. Метод принимает в качестве аргумента объект типа **Collector** («сборщик»), играющий роль правила, по которому будут собраны элементы из стрима. В классе **Collectors** подготовлен набор из готовых сборщиков. С помощью стандартных **Collectors.toList()** и **Collectors.toSet()** можно легко получать коллекции типа **List** и **Set** .

```
public static void main (String[] args) {
    Stream<String> stream = Stream.of( "A" , "B" , "C" );
    List<String> list = stream.collect(Collectors.toList());
    Set<String> set = stream.collect(Collectors.toSet()); }

```

Кроме **toList()** и **toSet()** **Collectors** позволяют создавать подгруппы объектов из стрима для общей цели. Например, можно определить среднюю длину слова в стриме строк:

```
public static void main (String[] args) {
    String[] array = { "Aaa" , "Bbbbb" , "Cc" };
    System.out.println(Arrays.stream(array)
        .collect(Collectors.averagingInt(s -> s.length())));
}
// Результат: 3.3333333333333335

```

Можно выбрать строки по определённому признаку и вывести строкой:

```
public static void main (String[] args) {
    String[] array = { "Aaa" , "Bbbbb" , "Cc" , "Aa" };
    System.out.println(Arrays.stream(array)
        .filter(str -> str.startsWith( "A" ))
        .collect(Collectors.joining( " и " ,
        "Перечисленные слова [" , "]" начинаются на букву A" )));
}
// Перечисленные слова [Aaa и Aa] начинаются на букву A

```

**joining()** принимает в качестве аргумента разделитель, префикс и суффикс.

Вторая терминальная операция — **forEach()** , её задача заключается в выполнении указанного действия для каждого элемента стрима. В примере ниже мы пройдем по каждому элементу стрима и отпечатаем его в консоль:

```
public static void main (String[] args) {
    Stream<String> stream1 = Stream.of( "A" , "B" , "C" );
    stream1.forEach(str -> System.out.println(str));

    Stream<String> stream2 = Stream.of( "A" , "B" , "C" );
    stream2.forEach(System.out::println); }

```

Оба стрима работают одинаково, только в первом случае каждый элемент стрима **str** передаётся в качестве аргумента методу **System.out.println()** . Во втором же случае просто указывается, что для каждого элемента стрима необходимо вызвать этот метод.

Операция **count()** возвращает количество элементов в стриме.

Операция **reduce()** выполняет роль сумматора по всем элементам стрима. Всего в данный момент поддерживается три частных случая такой операции, но мы рассмотрим один. Найдём наибольшее значение в стриме:

```
public static void main (String[] args) {
    Stream<Integer> stream = Stream.of( 1, 2, 3, 24, 5, 6 );
    stream.reduce((i1, i2) -> i1 > i2 ? i1 : i2)
        .ifPresent(System.out::println);
}

```

**reduce()** принимает функцию аккумулятора **BinaryOperator** . На самом деле это **BiFunction** — когда оба операнда имеют один и тот же тип (в данном случае — **Integer** ). Пример функции сравнивает по паре входящих чисел и возвращает наибольшее.

В этом подразделе мы практически не использовали промежуточные операции, поскольку необходимо было только понять принцип действия терминальных операций. Давайте теперь посмотрим как можно выполнять последовательную обработку данных.

## Промежуточные операции

Операция **filter()** позволяет отфильтровать элементы стрима по заданному правилу. В примере ниже мы пропускаем через фильтр только чётные числа:

```
public static void main (String[] args) {
    Stream<Integer> stream = Stream.of( 1, 2, 3, 4, 5,
6) ;    stream.filter(n -> n % 2 == 0)
    .forEach(System.out::print); }
// Результат: 246
```

Если из лямбда-выражения не очень понятно, как именно это работает, то ниже приведён расширенный вариант записи. В нём метод **filter()** отдаёт объект, реализующий обобщённый функциональный интерфейс **Predicate** , в котором прописан метод **test(Integer integer)** . Каждый объект стрима будет передан этому методу, и если метод возвращает **true** , то объект проходит фильтр, в противном случае — отсеивается:

```
// ...    stream.filter(        new
Predicate<Integer>() {
    @Override                public
boolean test (Integer integer) {
return integer % 2 == 0;
    }
})...
```

Операция **distinct()** позволяет преобразовать стрим в множество уникальных объектов (по аналогии с работой **Set** ):

```
public static void main (String[] args) {
    Stream.of( "A" , "A" , "A" , "B" , "B" , "B" , "B" )
        .distinct()
        .forEach(System.out::print);
}
// AB
```

Операция **map()** служит для преобразования типа объектов внутри стрима. В примере ниже мы преобразуем **Stream<String>** в **Stream<Integer>** путем вызова у каждого строкового объекта исходного стрима метода **length()** . Тем самым мы преобразуем набор слов в набор длин этих слов:

```
public static void main (String[] args) {
    Stream<String> stream = Stream.of( "Java" , "Core" ,
"ABC" );    stream.map(str ->
str.length()) .forEach(System.out::print); }
// 443
```

Операция **limit(int n)** ограничивает набор элементов в стриме, оставляет только первые n элементов. **sorted()** позволяет отсортировать стрим объектов либо в стандартном порядке, указанном в классе объекта, либо в соответствии с переданным в метод компаратором. **skip(int n)** пропускает первые n элементов стрима.

При работе со специализированными стримами типа **IntStream** появляются методы, недоступные для обычного стрима, например: **sum()** — получает сумму элементов **IntStream**, **average()** — подсчитывает среднее значение элементов, **min()** / **max()** — находит минимальный/максимальный элемент в стриме.

## Описание базовых функциональных интерфейсов

В Java представлен базовый набор функциональных интерфейсов, применяемых при работе со Stream API. Давайте посмотрим для чего они нужны и чем отличаются.

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}
```

**Function** представляет собой функция, которая принимает на вход аргумент типа T и возвращает (производит) объект типа R. (Например, в операции map данная функция позволяет привести один объект к другому).

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

**Predicate** представляет собой предикат (функцию, возвращающую boolean) для одного аргумента. (Например, для операции filter мы прописываем предикат, определяющий критерий для отфильтровывания объектов).

```
@FunctionalInterface public interface Consumer<T> {
    void accept(T t);
}
```

**Consumer** представляет операцию, которая принимает на вход один аргумент и не возвращает никакого результата. (Например, в операции forEach)

```
@FunctionalInterface
public interface Supplier<T> {
    T get();
}
// Пример
Stream.generate( new Supplier<Integer>() {
    @Override
    public Integer get() { return (int) (Math.random() * 100) ; }
})
```

**Supplier** представляет собой поставщик результатов. В примере выше Supplier будет возвращать числа в пределах от 0 до 99.

```
@FunctionalInterface
public interface BinaryOperator<T> extends BiFunction<T, T, T> {
    T apply(T t1, T t2);
}
```

**BinaryOperator** представляет собой операцию, принимающую на вход два аргумента одного и того же типа, и возвращающая результат того же типа. (Например, используется в `reduce()`).

## Порядок обработки

Узнаем, что происходит внутри операций. Рассмотрим фрагмент кода, в котором нет терминальной операции:

```
Stream.of( "dd2" , "aa2" , "bb1" , "bb3" , "cc4" )
    .filter(s -> {
        System.out.println( "Фильтр: " + s);
        return true ;
    });
```

Во время выполнения этого блока консоль будет оставаться пустой, так как все промежуточные операции будут выполняться, только если присутствует хотя бы одна терминальная. Расширим пример терминальной операцией **forEach()** , и в консоли начнёт выводиться наш блок:

```
Stream.of( "1" , "2" , "3" , "4" , "5" )
    .filter(s -> {
        System.out.println( "Фильтр: " + s);
        return true ;
    })
    .forEach(s -> System.out.println( "Результат: " + s));
```

Что появится в консоли после выполнения блока:

```
Фильтр: 1
Результат: 1
Фильтр: 2
Результат: 2
Фильтр: 3
Результат: 3
Фильтр: 4
Результат: 4
Фильтр: 5
Результат: 5
```

Порядок выполнения может удивить. На первый взгляд, все операции будут выполняться по горизонтали одна за другой по всем элементам стрима. Но в нашем примере сначала первая строка «**dd2**» полностью проходит фильтр **forEach** , потом обрабатывается вторая строка «**aa2**» и так далее.

## Почему порядок работы имеет значение?

Следующий пример состоит из двух промежуточных операций **map** и **filter** и выполнения терминала **forEach**. Посмотрим ещё раз на порядок выполнения этих операций:

```
Stream.of( "dd2" , "aa2" , "bb1" , "bb3" , "cc4" )
    .map(s -> {
        System.out.println( "map: " + s);
        return s.toUpperCase();
    })
    .filter(s -> {
        System.out.println( "filter: " + s);
        return s.startsWith( "A" );
    })
    .forEach(s -> System.out.println( "forEach: " + s));
// Результат выполнения
// map:      dd2
// filter:   DD2
// map:      aa2
// filter:   AA2
// forEach:  AA2
// map:      bb1
// filter:   BB1
// map:      bb3
// filter:   BB3
// map:      cc
// filter:   CC
```

**map()** и **filter()** вызываются 5 раз для каждой строчки в коллекции, а **forEach** — только один раз. Но можно сократить количество выполнений, изменив порядок операций. Посмотрим, что произойдёт, если **filter()** окажется в начале цепи:

```
Stream.of( "dd2" , "aa2" , "bb1" , "bb3" , "cc4" )
    .filter(s -> {
        System.out.println( "filter: " + s);
        return s.startsWith( "a" );
    })
    .map(s -> {
        System.out.println( "map: " + s);
        return s.toUpperCase();
    })
    .forEach(s -> System.out.println( "forEach: " + s));
// filter:  dd2
// filter:  aa2
// map:     aa2
// forEach: AA2
// filter:  bb1
// filter:  bb3
// filter:  cc
```

Теперь **map** вызывается только один раз и выполняется быстрее для большого количества входных элементов. Это стоит иметь в виду при составлении комплексного метода цепи.

Есть правило при работе со стримами в Java 8: их нельзя использовать дважды, после выполнения терминальной операции стрим закрывается.



```
Stream<String> stream = Stream.of( "a1" , "b2" , "a3" ,
    "c4" , "d5" )
    .filter(s -> s.startsWith( "d" ));
stream.anyMatch(s -> true );           // Пройдёт без проблем
stream.noneMatch(s -> true );          // Выдаст исключение
```

Вызов операции **noneMatch()** после выполнения терминальной операции — в данном случае **anyMatch()** — для одного стрима вызовет исключение:

```
Exception in thread "main" java.lang.IllegalStateException: stream has already
been operated upon or closed at
java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java: 229) at
java.util.stream.ReferencePipeline.noneMatch(ReferencePipeline.java: 459)
```

Чтобы избежать этого, надо создать новую цепь для каждой терминальной операции.

## Параллельные стримы

Стримы могут выполняться параллельно, чтобы увеличить производительность при огромном количестве входных элементов. Они используют общий **ForkJoinPool** , доступный через статический метод **ForkJoinPool.commonPool()** . Размер основного пула — не более пяти потоков (**Thread** ), так как количество ядер процессора — величина ограниченная.

```
ForkJoinPool commonPool = ForkJoinPool.commonPool();
System.out.println(commonPool.getParallelism()); // 3
```

На обыкновенном компьютере общий пул выставляется в значение 3 по умолчанию. Эти параметры можно изменять, как и любые другие, как описано ниже:

```
-Djava.util.concurrent.ForkJoinPool.common.parallelism= 5
```

Коллекции поддерживают метод **parallelStream()** для создания параллельного стрима элементов. Можно вызвать промежуточный метод **parallel()** , чтобы преобразовать последовательный поток к параллельной копии.

Для понимания того, как работает параллельный стрим, печатаем информацию о текущем потоке в **System.out** :

```
Arrays.asList( "a1" , "a2" , "b1" , "c2" , "c1" )
    .parallelStream()
    .filter(s -> {
        System.out.format( "filter: %s [%s]\n" ,
s, Thread.currentThread().getName());           return true ;
    })
    .map(s -> {
        System.out.format( "map: %s [%s]\n" ,
            s, Thread.currentThread().getName());
return s.toUpperCase();
    })
    .forEach(s -> System.out.format( "forEach: %s [%s]\n"
, s, Thread.currentThread().getName()));
```

После дебага становится понятнее, какие потоки на самом деле используются для выполнения операций стрима:

```
filter: b1 [main]
filter: a2 [ForkJoinPool.commonPool-worker-1] map: a2 [ForkJoinPool.commonPool-worker-1] filter: c2 [ForkJoinPool.commonPool-worker-3] map: c2 [ForkJoinPool.commonPool-worker-3] filter: c1 [ForkJoinPool.commonPool-worker-2] map: c1 [ForkJoinPool.commonPool-worker-2] forEach: C2 [ForkJoinPool.commonPool-worker-3]
forEach: A2 [ForkJoinPool.commonPool-worker-1]
map: b1 [main] forEach: B1 [main] filter: a1 [ForkJoinPool.commonPool-worker-3] map: a1 [ForkJoinPool.commonPool-worker-3] forEach: A1 [ForkJoinPool.commonPool-worker-3]
forEach: C1 [ForkJoinPool.commonPool-worker-2]
```

Параллельный поток использует все доступные ресурсы из общей **ForkJoinPool** для выполнения операций стрима. Вывод может отличаться при последовательном запуске, потому что поведение, которое использует конкретный поток, не детерминировано.

Расширим пример с помощью дополнительной операции **sort()** :

```
Arrays.asList( "a1" , "a2" , "b1" , "c2" , "c1" )
    .parallelStream()
    .filter(s -> {
        System.out.format( "filter: %s [%s]\n"
, s, Thread.currentThread().getName());
return true ;
    })
    .map(s -> {
        System.out.format( "map: %s [%s]\n" ,
s, Thread.currentThread().getName());
return s.toUpperCase();
    })
    .sorted((s1, s2) -> {
        System.out.format( "sort: %s <> %s [%s]\n" ,
s1, s2, Thread.currentThread().getName());
return s1.compareTo(s2);
    })
    .forEach(s -> System.out.format( "forEach: %s [%s]\n"
, s, Thread.currentThread().getName()));
```

Результат может на первый взгляд показаться странным:

```
filter: c2 [ForkJoinPool.commonPool-worker-3] filter: c1 [ForkJoinPool.commonPool-worker-2] map: c1 [ForkJoinPool.commonPool-worker-2] filter: a2
```

```
[ForkJoinPool.commonPool-worker- 1] map: a2
[ForkJoinPool.commonPool-worker- 1] filter:
b1 [main] map: b1 [main]
filter: a1 [ForkJoinPool.commonPool-worker-
2] map: a1 [ForkJoinPool.commonPool-worker-
2] map: c2 [ForkJoinPool.commonPool-worker-
3] sort: A2 <> A1 [main] sort: B1 <> A2
[main] sort: C2 <> B1 [main] sort: C1 <>
C2 [main] sort: C1 <> B1 [main] sort: C1
<> C2 [main] forEach: A1
[ForkJoinPool.commonPool-worker- 1] forEach:
C2 [ForkJoinPool.commonPool-worker- 3]
forEach: B1 [main]
forEach: A2 [ForkJoinPool.commonPool-worker-
2] forEach: C1 [ForkJoinPool.commonPool-
worker- 1]
```

Кажется, что **sort()** выполняется последовательно только потоком **main**, но на самом деле под капотом используется метод **Arrays.parallelSort()**, следовательно сортировка распараллелена. Отладочный же вывод относится только к исполнению переданного лямбда-выражения.

Вернемся к **reduce()**. Мы уже выяснили, что функция-комбайнер вызывается только параллельно, но не в последовательных потоках. Посмотрим, какие потоки фактически участвуют в этом:

```
static class Person {
private String name;
private int age;
    public Person (String name, int
age) {
        this .name = name;
this .age = age;
    }
}
public static void main (String[]
args) {
    List<Person> persons =
Arrays.asList(
        new Person(
"Andrew" , 20) ,
        new
Person( "Igor" , 23) ,
new Person( "Ira" , 23) ,
new Person( "Victor" , 29)
    );
}
```

```
persons
    .parallelStream()
    .reduce( 0,
        (sum, p) -> {
            System.out.format( "accum: sum=%s;
person=%s [%s]\n" ,
                                sum, p,
Thread.currentThread().getName());
return sum += p.age;
        },
        (sum1, sum2) -> {
            System.out.format( "combiner: sum1=%s;
sum2=%s [%s]\n" ,
                                sum1, sum2,
```

```
Thread.currentThread().getName());  
return sum1 + sum2;  
});  
}
```

Вывод в консоль показывает, что оба *аккумулятора* и *комбайнера* функции выполняются параллельно на всех доступных потоках:

```
accum: sum=0; person=Ira [main]  
accum: sum=0; person=Andrew [ForkJoinPool.commonPool-worker-  
3] accum: sum=0; person=Igor [ForkJoinPool.commonPool-worker-  
1] accum: sum=0; person=Victor [ForkJoinPool.commonPool-  
worker-2] combiner: sum1=20; sum2=23  
[ForkJoinPool.commonPool-worker-1] combiner: sum1=23; sum2=29  
[ForkJoinPool.commonPool-worker-2] combiner: sum1=43; sum2=52  
[ForkJoinPool.commonPool-worker-2]
```

Параллельное выполнение может дать хороший прирост производительности в потоках с большим количеством входных элементов. Но некоторые параллельные операции стрима **reduce()** и **collect()** требуют дополнительных расчётов (комбинированные операции), которые не нужны при последовательном выполнении.

## Практическое задание

1. Создайте массив с набором слов, и с помощью Stream API найдите наиболее часто встречающееся;
2. Создайте массив объектов типа Сотрудник (с полями Имя, Возраст, Зарплата) и вычислите среднюю зарплату сотрудника;
3. Напишите метод, способный найти в массиве сотрудников из п. 2 найдите N самых старших сотрудников и отпечатает в консоль сообщение вида "N самых старших сотрудников зовут: имя1, имя2, имяN;".

## Дополнительные материалы

1. Кей С. Хорстманн, Гари Корнелл Java. Библиотека профессионала. Том 1. Основы // Пер. с англ. — М.: Вильямс, 2014. — 864 с.
2. Брюс Эккель. Философия Java // 4-е изд.: Пер. с англ. — СПб.: Питер, 2016. — 1 168 с.
3. Г. Шилдт. Java 8. Полное руководство // 9-е изд.: Пер. с англ. — М.: Вильямс, 2015. — 1 376 с.
4. Г. Шилдт. Java 8: Руководство для начинающих. // 6-е изд.: Пер. с англ. — М.: Вильямс, 2015. — 720 с.

## Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Г. Шилдт. Java 8. Полное руководство // 9-е изд.: Пер. с англ. — М.: Вильямс, 2015. — 1 376 с.