Оглавление

- concat() (Stream, IntStream, LongStream, DoubleStream)
- stream() (Collection или Arrays)
- lines() (BufferedReader)
- lines() (Files)
- walk() (Files), list() (Files)
- chars() (String)

Создание бесконечных стримов, как сделать их конечными

Промежуточные операции

- filter() (Stream, IntStream, LongStream, DoubleStream)
- map() (Stream, IntStream, LongStream, DoubleStream)
  - mapToObj() (IntStream, LongStream, DoubleStream)
  - mapToInt() (Stream, LongStream, DoubleStream)
  - mapToLong() (Stream, IntStream, DoubleStream)
  - mapToDouble() (Stream, IntStream, LongStream)
- flatMap() (Stream, IntStream, LongStream, DoubleStream)

flatMapToInt(), flatMapToLong(), flatMapToDouble() (Stream)

distinct() (Stream, IntStream, LongStream, DoubleStream)

sorted() (Stream (2), IntStream, LongStream, DoubleStream)

peek() (Stream, IntStream, LongStream, DoubleStream)

limit() (Stream, IntStream, LongStream, DoubleStream)

skip() (Stream, IntStream, LongStream, DoubleStream)

Терминальные операции

forEach() (Stream, IntStream, LongStream, DoubleStream)

forEachOrdered() (Stream, IntStream, LongStream, DoubleStream)

toArray() (Stream, IntStream, LongStream, DoubleStream)

reduce() (Stream (3), IntStream (2), LongStream (2), DoubleStream (2))

collect() (Stream (2), IntStream, LongStream, DoubleStream)

sum() (IntStream, LongStream, DoubleStream)

`min()` (Stream, IntStream, LongStream, DoubleStream)

`max()` (Stream, IntStream, LongStream, DoubleStream)

`count()` (Stream, IntStream, LongStream, DoubleStream)

`average()` (IntStream, LongStream, DoubleStream)

`summaryStatistics()` (IntStream, LongStream, DoubleStream)

`anyMatch()` (Stream, IntStream, LongStream, DoubleStream)

`allMatch()`(Stream, IntStream, LongStream, DoubleStream)

`noneMatch()` (Stream, IntStream, LongStream, DoubleStream)

`findFirst()` (Stream, IntStream, LongStream, DoubleStream)

`findAny()` (Stream, IntStream, LongStream, DoubleStream)

`asLongStream()` (IntStream)

`asDoubleStream()` (IntStream, LongStream)

`boxed()` (IntStream, LongStream, DoubleStream)

`sequential()` (`IntStream`, `LongStream`, `DoubleStream`)

`parallel()` (`IntStream`, `LongStream`, `DoubleStream`)

`iterator()` (`IntStream`, `DoubleStream`, `LongStream`)

`spliterator()` (`IntStream`, `LongStream`, `DoubleStream`)

Stream API

Что такое и для чего нужны?

Последовательность элементов, потенциально бесконечных, над которыми можно производить различные операции в одну строку (без использования циклов и условных операторов).

С какими типами данных работает стрим?

Помимо объектов (стрим является дженериком) стрим может работать со следующими примитивными типами: `int`, `long` и `double`. Для этого к `Stream<T>` добавили примитивные стримы: `IntStream`, `LongStream` и `DoubleStream`. В отличии от стрима дженерика они используют примитивные функциональные интерфейсы и имеют дополнительные терминальные операции (`sum()`, `average()`, `mapToObj()`).

Отличия от коллекции

· коллекции — это прежде всего хранение элементов, а стрим — выполнение каких-либо действий над элементами стрима;
· коллекции позволяют работать с элементами по отдельности;
· стрим может быть потенциально бесконечный;

· некоторые коллекции могут давать индивидуальный доступ по индексу или по ключу;
· в коллекции можно менять, удалять и добавлять элементы, а применение трансформации к стриму не влияют на источник.

Основные части при использовании стрима, порядок …

Получение стрима, промежуточные операции (0 или несколько), терминальная операция (единственная) и закрытие (если стрим выделял некоторые ресурсы, можно в блоке try с ресурсами).

Операции над стримами: какие бывают, разница и почему стримы ленивы?

Операции бывают промежуточные и терминальные (конечные).

Промежуточные операции могут выполнять несколько, т. к. возвращают тот же стрим.

Терминальная операция может быть только одна, т. к. возвращает результат определенного (другого) типа и поэтому не может использоваться повторно.

Стрим называется ленивым, потому что промежуточные операции не будут выполняться до применения терминальной.

Стрим с точки зрения многопоточности

Операции стрима могут выполнять последовательно и параллельно.

Потоки не могут быть использованы повторно. Как только была вызвана какая-нибудь конечная операция, поток закрывается.

Существуют терминальная операция `parallel()` (примитивные стримы) и `parallelStream()` (`Collection`).

Способы создания стрима

builder() (Stream, IntStream, LongStream, DoubleStream)

Позволяет добавить элементы в стрим при помощи методов вложенного интерфейса `Builder`. `of()` делает то же, но без помощи интерфейса `Builder`.

```
1. public static<T> Builder<T> builder() {
2.     return new Streams.StreamBuilderImpl<>();
3. }
```

```
1. public static Builder builder() {
2.     return new Streams.IntStreamBuilderImpl();
3. }
```

```
1. public static Builder builder() {
2.     return new Streams.LongStreamBuilderImpl();
3. }
```

```
1.  public static Builder builder() {
2.      return new Streams.DoubleStreamBuilderImpl();
3.  }
```

Интерфейса `Builder` для создания стрима вместе с `builder()`. `add()` — добавляет элементы, `build()` — создает из них стрим. `accept()` скорей всего используется для метода `add()`.

```
1.  public interface Builder<T> extends Consumer<T> {
2.
3.      @Override
4.      void accept(T t);
5.
6.      default Builder<T> add(T t) {
7.          accept(t);
8.          return this;
9.      }
10.
11.     Stream<T> build();
12. }
```

```
1.  public interface Builder extends IntConsumer {
2.
3.      @Override
4.      void accept(int t);
5.
6.      default Builder add(int t) {
7.          accept(t);
8.          return this;
9.      }
10.
11.     IntStream build();
12. }
```

```
1.  public interface Builder extends LongConsumer {
2.
3.      @Override
4.      void accept(long t);
5.
6.      default Builder add(long t) {
7.          accept(t);
8.          return this;
9.      }
```

```
10.
11.     LongStream build();
12. }
```

```
1.  public interface Builder extends DoubleConsumer {
2.
3.      @Override
4.      void accept(double t);
5.
6.      default Builder add(double t) {
7.          accept(t);
8.          return this;
9.      }
10.
11.     DoubleStream build();
12. }
```

```
1.  Stream fromBuilder = Stream.builder()
2.              .add("x")
3.              .add("y")
4.              .add("z")
5.          .build();
6.  fromBuilder.forEach(System.out::println);
```

x
y
z

empty() (Stream, IntStream, LongStream, DoubleStream)

Создание пустого стрима.

```
1.  public static<T> Stream<T> empty() {
2.      return StreamSupport.stream(
3.              Spliterators.<T>emptySpliterator(),
4.              false);
5.  }
```

```
1.  public static IntStream empty() {
2.      return StreamSupport.intStream(
3.              Spliterators.emptyIntSpliterator(),
4.              false);
5.  }
```

```
1.  public static LongStream empty() {
2.      return StreamSupport.longStream(
```

```
            Spliterators.emptyLongSpliterator(),
            false);
}
```

```
public static DoubleStream empty() {
    return StreamSupport.doubleStream(
            Spliterators.emptyDoubleSpliterator(),
            false);
}
```

```
IntStream empty = IntStream.empty();
System.out.println(empty.count());
```

0

of() (Stream (2), IntStream (2), LongStream (2), DoubleStream (2))

Создает стрим перечислением элементов. Похож на `builder()`, но для создания элементов не использует методы вложенного интерейса `Builder`.

```
public static<T> Stream<T> of(T t) {
    return StreamSupport.stream(
            new Streams.StreamBuilderImpl<>(t),
            false);
}
```

```
@SafeVarargs
@SuppressWarnings("varargs")
public static<T> Stream<T> of(T... values) {
    return Arrays.stream(values);
}
```

```
public static IntStream of(int t) {
    return StreamSupport.intStream(
            new Streams.IntStreamBuilderImpl(t),
            false);
}
```

```
public static IntStream of(int... values) {
    return Arrays.stream(values);
}
```

```
public static LongStream of(long t) {
    return StreamSupport.longStream(
```

```
3.              new Streams.LongStreamBuilderImpl(t),
4.              false);
5.  }
```

```
1.  public static LongStream of(long... values) {
2.      return Arrays.stream(values);
3.  }
```

```
1.  public static DoubleStream of(double t) {
2.      return StreamSupport.doubleStream(
3.              new Streams.DoubleStreamBuilderImpl(
4.                  t),
5.              false);
6.  }
```

```
1.  public static DoubleStream of(double... values) {
2.      return Arrays.stream(values);
3.  }
```

```
1.  IntStream streamOfElements =
2.          IntStream.of(2, 4, 5, 6, 8, 10);
3.  streamOfElements.forEach(System.out::println);
```

```
2
4
5
6
8
10
```

iterate() (Stream, IntStream, LongStream, DoubleStream)

Как и `generate()` создает бесконечный стрим. Задает значение первого элемента и дальнейшую последовательность при помощи `UnaryOperator` (`T apply(T t)`). Бесконечный стрим можно сделать конечным терминальной операцией `limit()`.

```
1.  public static<T> Stream<T> iterate(
2.          final T seed,
3.          final UnaryOperator<T> f) {
4.      Objects.requireNonNull(f);
```

```
    final Iterator<T> iterator =
            new Iterator<T>() {
        @SuppressWarnings("unchecked")
        T t = (T) Streams.NONE;

        @Override
        public boolean hasNext() {
            return true;
        }

        @Override
        public T next() {
            return t =
                    (t == Streams.NONE)
                        ? seed
                        : f.apply(t);
        }
    };
    return StreamSupport.stream(
            Spliterators.spliteratorUnknownSize(
                iterator,
                Spliterator.ORDERED
                    | Spliterator.IMMUTABLE),
            false);
}
```

```
public static IntStream iterate(
        final int seed,
        final IntUnaryOperator f) {
    Objects.requireNonNull(f);
    final PrimitiveIterator.OfInt iterator =
            new PrimitiveIterator.OfInt() {
        int t = seed;

        @Override
        public boolean hasNext() {
            return true;
        }

        @Override
        public int nextInt() {
            int v = t;
            t = f.applyAsInt(t);
            return v;
        }
```

```
        };
        return StreamSupport.intStream(
                Spliterators.spliteratorUnknownSize(
                    iterator,
                    Spliterator.ORDERED
                        | Spliterator.IMMUTABLE
                        | Spliterator.NONNULL),
                false);
    }
```

```
    public static LongStream iterate(
            final long seed,
            final LongUnaryOperator f) {
        Objects.requireNonNull(f);
        final PrimitiveIterator.OfLong iterator =
                new PrimitiveIterator.OfLong() {
            long t = seed;

            @Override
            public boolean hasNext() {
                return true;
            }

            @Override
            public long nextLong() {
                long v = t;
                t = f.applyAsLong(t);
                return v;
            }
        };
        return StreamSupport.longStream(
                Spliterators.spliteratorUnknownSize(
                    iterator,
                    Spliterator.ORDERED
                        | Spliterator.IMMUTABLE
                        | Spliterator.NONNULL),
                false);
    }
```

```
    public static DoubleStream iterate(
            final double seed,
            final DoubleUnaryOperator f) {
        Objects.requireNonNull(f);
        final PrimitiveIterator.OfDouble iterator =
                new PrimitiveIterator.OfDouble() {
            double t = seed;
```

```
 8.
 9.          @Override
10.          public boolean hasNext() {
11.              return true;
12.          }
13.
14.          @Override
15.          public double nextDouble() {
16.              double v = t;
17.              t = f.applyAsDouble(t);
18.              return v;
19.          }
20.      };
21.      return StreamSupport.doubleStream(
22.              Spliterators.spliteratorUnknownSize(
23.                  iterator,
24.                  Spliterator.ORDERED
25.                      | Spliterator.IMMUTABLE
26.                      | Spliterator.NONNULL),
27.              false);
28.  }
```

```
1.  IntStream integers =
2.          IntStream.iterate(0, n -> n + 1);
3.  integers.forEach(System.out::println);
```

вывод целых чисел от 0 до остановки

generate() (Stream, IntStream, LongStream, DoubleStream)

Как и `iterate()` создает бесконечный стрим, но использует `Supplier` (`T get()`). Бесконечный стрим можно сделать конечным терминальной операцией `limit()`.

```
1.  public static<T> Stream<T> generate(
2.          Supplier<T> s) {
3.      Objects.requireNonNull(s);
4.      return StreamSupport.stream(
5.              new StreamSpliterators
6.                  .InfiniteSupplyingSpliterator
```

```
                .OfRef<>(Long.MAX_VALUE, s),
            false);
}
```

```
public static IntStream generate(IntSupplier s) {
    Objects.requireNonNull(s);
    return StreamSupport.intStream(
            new StreamSpliterators
                .InfiniteSupplyingSpliterator
                .OfInt(Long.MAX_VALUE, s),
            false);
}
```

```
public static LongStream generate(
        LongSupplier s) {
    Objects.requireNonNull(s);
    return StreamSupport.longStream(
            new StreamSpliterators
                .InfiniteSupplyingSpliterator
                .OfLong(Long.MAX_VALUE, s),
            false);
}
```

```
public static DoubleStream generate(
        DoubleSupplier s) {
    Objects.requireNonNull(s);
    return StreamSupport.doubleStream(
            new StreamSpliterators
                .InfiniteSupplyingSpliterator
                .OfDouble(Long.MAX_VALUE, s),
            false);
}
```

```
DoubleStream randomNumbers =
        DoubleStream.generate(Math::random);
randomNumbers.forEach(
        System.out::println);
```

вывод случайных чисел типа Double до остановки

range() (IntStream, LongStream)

Как и `rangeClosed()` создает целый числовой стрим из указанного диапазона чисел, но последний элемент не включен.

```
 1.  public static IntStream range(
 2.          int startInclusive,
 3.          int endExclusive) {
 4.      if (startInclusive >= endExclusive) {
 5.          return empty();
 6.      } else {
 7.          return StreamSupport.intStream(
 8.                  new Streams.RangeIntSpliterator(
 9.                      startInclusive,
10.                      endExclusive,
11.                      false),
12.                  false);
13.      }
14.  }
```

```
 1.  public static LongStream range(
 2.          long startInclusive,
 3.          final long endExclusive) {
 4.      if (startInclusive >= endExclusive) {
 5.          return empty();
 6.      } else if (
 7.              endExclusive - startInclusive < 0) {
 8.          long m =
 9.                  startInclusive
10.                  + Long.divideUnsigned(
11.                      endExclusive - startInclusive,
12.                      2)
13.                  + 1;
14.          return concat(
15.                  range(startInclusive, m),
16.                  range(m, endExclusive));
17.      } else {
18.          return StreamSupport.longStream(
19.                  new Streams.RangeLongSpliterator(
20.                      startInclusive,
21.                      endExclusive,
22.                      false),
23.                  false);
24.      }
25.  }
```

```
 1.  IntStream smallIntegers = IntStream.range(0, 100);
 2.  smallIntegers.forEach(System.out::println);
```

целые числа от 0 до 100 не включительно

rangeClosed() (IntStream, LongStream)

Как и `range()` создает целый числовой стрим из указанного диапазона чисел, но последний элемент будет включен.

```
 1. public static IntStream rangeClosed(
 2.         int startInclusive,
 3.         int endInclusive) {
 4.     if (startInclusive > endInclusive) {
 5.         return empty();
 6.     } else {
 7.         return StreamSupport.intStream(
 8.                 new Streams.RangeIntSpliterator(
 9.                     startInclusive,
10.                     endInclusive,
11.                     true),
12.                 false);
13.     }
14. }
```

```
 1. public static LongStream rangeClosed(
 2.         long startInclusive,
 3.         final long endInclusive) {
 4.     if (startInclusive > endInclusive) {
 5.         return empty();
 6.     } else if (
 7.             endInclusive - startInclusive + 1
 8.             <= 0) {
 9.         long m =
10.                 startInclusive
11.                 + Long.divideUnsigned(
12.                     endInclusive - startInclusive,
13.                     2)
14.                 + 1;
15.         return concat(
16.                 range(startInclusive, m),
17.                 rangeClosed(m, endInclusive));
18.     } else {
19.         return StreamSupport.longStream(
20.                 new Streams.RangeLongSpliterator(
21.                     startInclusive,
22.                     endInclusive,
23.                     true),
```

```
                false);
    }
}
```

```
IntStream smallIntegers2 =
IntStream.rangeClosed(0, 100);
smallIntegers2.forEach(System.out::println);
```

целые числа от 0 до 100 включительно

concat() (Stream, IntStream, LongStream, DoubleStream)

Объединение двух стримов.

```
public static <T> Stream<T> concat(
        Stream<? extends T> a,
        Stream<? extends T> b) {
    Objects.requireNonNull(a);
    Objects.requireNonNull(b);

    @SuppressWarnings("unchecked")
    Spliterator<T> split =
            new Streams.ConcatSpliterator.OfRef<>(
                (Spliterator<T>) a.spliterator(),
                (Spliterator<T>) b.spliterator());
    Stream<T> stream = StreamSupport.stream(
            split,
            a.isParallel() || b.isParallel());
    return stream.onClose(
            Streams.composedClose(a, b));
}
```

```
public static IntStream concat(
        IntStream a,
        IntStream b) {
    Objects.requireNonNull(a);
    Objects.requireNonNull(b);

    Spliterator.OfInt split =
            new Streams.ConcatSpliterator.OfInt(
                a.spliterator(),
                b.spliterator());
    IntStream stream = StreamSupport.intStream(
            split,
```

```
            a.isParallel() || b.isParallel());
    return stream.onClose(
        Streams.composedClose(a, b));
}
```

```
public static LongStream concat(
        LongStream a,
        LongStream b) {
    Objects.requireNonNull(a);
    Objects.requireNonNull(b);

    Spliterator.OfLong split =
            new Streams.ConcatSpliterator.OfLong(
                a.spliterator(),
                b.spliterator());
    LongStream stream = StreamSupport.longStream(
            split,
            a.isParallel() || b.isParallel());
    return stream.onClose(
            Streams.composedClose(a, b));
}
```

```
public static DoubleStream concat(
        DoubleStream a,
        DoubleStream b) {
    Objects.requireNonNull(a);
    Objects.requireNonNull(b);

    Spliterator.OfDouble split =
            new Streams.ConcatSpliterator
                .OfDouble(
                    a.spliterator(),
                    b.spliterator());
    DoubleStream stream =
            StreamSupport.doubleStream(
                split,
                a.isParallel() || b.isParallel());
    return stream.onClose(
            Streams.composedClose(a, b));
}
```

```
List<String> list = new ArrayList<>();
Stream<String> streamOfList = list.stream();

Set<String> set = new HashSet<>();
```

```
Stream<String> streamOfSet = set.stream();

Stream<String> combainedStream =
        Stream.concat(streamOfList, streamOfSet);
combainedStream.forEach(
        System.out::println);
```

вывод 2-х ранее созданных Stream

```
IntStream streamOfChars = "строка".chars();

IntStream smallIntegers = IntStream.range(0, 15);

IntStream combainedIntStream = IntStream.concat(
        streamOfChars,
        smallIntegers);
combainedIntStream.forEach(System.out::println);
```

вывод 2-х ранее созданных IntStream

stream() (Collection или Arrays)

Создание стрима из `Collection` или массива. Создать стрим из ассоциативного массива нельзя.

```
List<String> list = new ArrayList<>();
Stream<String> streamOfList = list.stream();
streamOfList.forEach(System.out::println);
```

элементы списка

```
Set<String> set = new HashSet<>();
Stream<String> streamOfSet = set.stream();
streamOfSet.forEach(System.out::println);
```

элементы сета

```
double[] array = new double[5];
DoubleStream streamFromArray =
        Arrays.stream(array);
streamFromArray.forEach(System.out::println);
```

элементы массива

lines() (BufferedReader)

Создание стрима из `BufferedReader`.

```
try (
        BufferedReader bufferedReader =
            new BufferedReader(new FileReader(
```

```
                    "D:\\java\\example\\input.txt")))
                        {
    Stream<String> streamOfBufferedReader =
            bufferedReader.lines();
    streamOfBufferedReader.forEach(
        System.out::println);
} catch (IOException e) {
    e.printStackTrace();
}
```

содержимое файла input.txt

lines() (Files)

Создание стрима из файла.

```
try (Stream<String> fromFile =
        Files.lines(Paths.get(
            "D:\\java\\example\\input.txt"));) {
    fromFile.forEach(System.out::println);
} catch (IOException e) {
    e.printStackTrace();
}
```

содержимое файла

walk() (Files), list() (Files)

Создание стрима из пути.

```
Path path = Paths.get("D:\\java\\example");
try (
        Stream<Path> streamOfPathWithWalk =
            Files.walk(path)) {
    streamOfPathWithWalk.forEach(
            System.out::println);
} catch (IOException e) {
    e.printStackTrace();
}
```

url папки и ее содержимого

```
Path path = Paths.get("D:\\java\\example");
try (
        Stream<Path> streamOfPathWithList =
            Files.list(path)) {
    streamOfPathWithList.forEach(
            System.out::println);
} catch (IOException e) {
```

```
8.      e.printStackTrace();
9.  }
```

url только содержимого папки

chars() (String)

Создание стрима из строки.

```
1.  String string = "строка";
2.  IntStream streamOfChars = string.chars();
3.  streamOfChars.forEach(System.out::println);
4.  streamOfChars.close();
```

символы строки в виде целого числа

Создание бесконечных стримов, как сделать их конечными

`iterate()` и `generate()`.

```
1.  IntStream integers =
2.          IntStream.iterate(0, n -> n + 1);
3.  integers.forEach(System.out::println);
```

вывод целых чисел от 0 до остановки

```
1.  DoubleStream randomNumbers =
2.          DoubleStream.generate(Math::random);
3.  randomNumbers.forEach(
4.          System.out::println);
```

вывод случайных чисел типа Double до остановки

Чтобы сделать их конечными нужно применить промежуточную операцию `limit()`.

Промежуточные операции

filter() (Stream, IntStream, LongStream, DoubleStream)

Фильтрует элементы в соответствии с условием `Predicate` (`boolean test(T t)`).

```
1.  Stream<T> filter(Predicate<? super T> predicate);
1.  IntStream filter(IntPredicate predicate);
1.  LongStream filter(LongPredicate predicate);
```

```
1.  DoubleStream filter(DoublePredicate predicate);
```

```
1.  Arrays.asList(1, 2, 7, 14).stream()
2.          .filter(i -> i % 2 == 0)
3.          .forEach(System.out::println);
```

2
4

map() (Stream, IntStream, LongStream, DoubleStream)

Произвести какое-либо действия над элементами. `Stream` (не примитивный) позволяет изменить тип элементов: `Function (R apply(T t))`. Примитивные стримы нет: `IntUnaryOperator (int applyAsInt(int operand))`

```
1.  <R> Stream<R> map(
2.          Function<? super T, ? extends R> mapper);
```
```
1.  IntStream map(IntUnaryOperator mapper);
```
```
1.  LongStream map(LongUnaryOperator mapper);
```
```
1.  DoubleStream map(DoubleUnaryOperator mapper);
```

```
1.  Arrays.asList("1", "2", "7", "14").stream()
2.          .map(num -> Integer.parseInt(num))
3.          //.mapToInt(num -> Integer.parseInt(num))
4.          .map(i -> i + 2)
5.          .forEach(System.out::println);
6.      }
7.  }
```

2
4
14
28

mapToObj() (IntStream, LongStream, DoubleStream)

Произвести какое-либо действия над элементами примитивных стримов и преобразовать их в объекты: `IntFunction (R apply(int value))`.

```
<U> Stream<U> mapToObj(
        IntFunction<? extends U> mapper);
```

```
<U> Stream<U> mapToObj(
        LongFunction<? extends U> mapper);
```

```
<U> Stream<U> mapToObj(
        DoubleFunction<? extends U> mapper);
```

mapToInt() (Stream, LongStream, DoubleStream)

Произвести какое-либо действия над элементами и преобразовать их к `int`: `ToIntFunction (int applyAsInt(T value))`.

```
IntStream mapToInt(
        ToIntFunction<? super T> mapper);
```

```
IntStream mapToInt(LongToIntFunction mapper);
```

```
IntStream mapToInt(DoubleToIntFunction mapper);
```

mapToLong() (Stream, IntStream, DoubleStream)

Произвести какое-либо действия над элементами и преобразовать их к `long`: `ToLongBiFunction (long applyAsLong(T t, U u))`.

```
LongStream mapToLong(
        ToLongFunction<? super T> mapper);
```

```
LongStream mapToLong(IntToLongFunction mapper);
```

```
LongStream mapToLong(DoubleToLongFunction mapper);
```

mapToDouble() (Stream, IntStream, LongStream)

Произвести какое-либо действия над элементами и преобразовать их к double: ToDoubleFunction (double applyAsDouble(T value)).

```
DoubleStream mapToDouble(
        ToDoubleFunction<? super T> mapper);
```

```
DoubleStream mapToDouble(
        IntToDoubleFunction mapper);
```

```
DoubleStream mapToDouble(
        LongToDoubleFunction mapper);
```

flatMap() (Stream, IntStream, LongStream, DoubleStream)

В отличии от map() может вернуть другое количество элементов. В Stream за счет того, что в Function возвращает Stream (Stream<R> apply(T t)). В примитивах IntFunction – IntStream (IntStream apply(int value)).

```
<R> Stream<R> flatMap(
        Function<
            ? super T,
            ? extends Stream<? extends R>>
        mapper);
```

```
IntStream flatMap(
        IntFunction<? extends IntStream> mapper);
```

```
LongStream flatMap(
        LongFunction<? extends LongStream>
        mapper);
```

```
DoubleStream flatMap(
        DoubleFunction<? extends DoubleStream>
        mapper);
```

flatMapToInt(), flatMapToLong(), flatMapToDouble() (Stream)

flatMap(), но преобразовывает к int, long или double. Function возвращает IntStream (IntStream apply(T t)).

```
IntStream flatMapToInt(
        Function<
            ? super T,
            ? extends IntStream>
        mapper);
```

```
try (BufferedReader bufferedReader =
        new BufferedReader(new FileReader(
            "D:\\java\\example\\input.txt"))) {
    Stream<String> streamOfBufferedReader =
            bufferedReader.lines();
    streamOfBufferedReader
            .flatMapToInt(s -> s.chars())
            .forEach(System.out::println);
} catch (IOException e) {
    e.printStackTrace();
}
```

```
LongStream flatMapToLong(
        Function<
            ? super T,
            ? extends LongStream>
        mapper);
```

```
DoubleStream flatMapToDouble(
        Function<
            ? super T,
            ? extends DoubleStream>
        mapper);
```

distinct() (Stream, IntStream, LongStream, DoubleStream)

Удаляет дубликаты.

```
Stream<T> distinct();
```

```
1.  IntStream distinct();
1.  LongStream distinct();
1.  DoubleStream distinct();
```

```
1.  Arrays.asList("1", "1", "7", "1").stream()
2.          .distinct()
3.          .forEach(System.out::println);
```

1
7

sorted() (Stream (2), IntStream, LongStream, DoubleStream)

Сортирует поток. Есть метод для сортировки при помощи `Comparator`.

```
1.  Stream<T> sorted();
1.  Stream<T> sorted(
2.          Comparator<? super T> comparator);
1.  IntStream sorted();
1.  LongStream sorted();
1.  DoubleStream sorted();
```

```
1.  Arrays.asList(1, 9, 7, 4).stream()
2.          .sorted()
3.          .forEach(System.out::println);
```

1
4
7
9

peek() (Stream, IntStream, LongStream, DoubleStream)

Произвести какое-либо действия над элементами. В отличии от `map()` принимает `Consumer` (`void accept(T t)`) и не может изменить тип элемента.

forEach() — аналогичная терминальная операция.

```
Stream<T> peek(Consumer<? super T> action);
IntStream peek(IntConsumer action);
LongStream peek(LongConsumer action);
DoubleStream peek(DoubleConsumer action);
```

```
Stream<String> nameStream = Stream.of(
        "Alice", "Bob", "Chuck");
nameStream
        .peek(System.out::println)
        .count();
```

Alice
Bob
Chuck

limit() (Stream, IntStream, LongStream, DoubleStream)

Обрезает стрим до указанного количества элементов.

```
Stream<T> limit(long maxSize);
IntStream limit(long maxSize);
LongStream limit(long maxSize);
DoubleStream limit(long maxSize);
```

```
Arrays.asList(1, 9, 7, 4).stream()
        .sorted()
        .limit(2)
        .forEach(System.out::println);
```

1
4

skip() (Stream, IntStream, LongStream, DoubleStream)

Пропускает указывает количество элементов.

```
1. Stream<T> skip(long n);
1. IntStream skip(long n);
1. LongStream skip(long n);
1. DoubleStream skip(long n);
```

```
1. Arrays.asList(1, 9, 7, 4, 24).stream()
2.         .skip(3)
3.         .forEach(System.out::println);
```

4
24

Терминальные операции

forEach() (Stream, IntStream, LongStream, DoubleStream)

Произвести какое-либо действия над элементами. Использует `Consumer` (`void accept(T t)`). `peek()` аналогичная промежуточная операция.

```
1. void forEach(Consumer<? super T> action);
1. void forEach(IntConsumer action);
1. void forEach(LongConsumer action);
1. void forEach(DoubleConsumer action);
```

```
1. Stream.of("Alice", "Bob", "Chuck")
2.         .forEach(System.out::println);
```

Alice
Bob
Chuck

forEachOrdered() (Stream, IntStream, LongStream, DoubleStream)

В отличии от forEach() предназначена для многопоточки (???) и всегда сохраняет порядок добавления.

Для чего в стримах применяются методы forEach() и forEachOrdered()?

forEach() применяет функцию к каждому объекту стрима, порядок при параллельном выполнении не гарантируется;

forEachOrdered() применяет функцию к каждому объекту стрима с сохранением порядка элементов.

```
1. void forEachOrdered(Consumer<? super T> action);
1. void forEachOrdered(IntConsumer action);
1. void forEachOrdered(LongConsumer action);
1. void forEachOrdered(DoubleConsumer action);
```

toArray() (Stream, IntStream, LongStream, DoubleStream)

Вернет массив.

```
1. Object[] toArray();
1. int[] toArray();
1. long[] toArray();
1. double[] toArray();
```

```
1.  <A> A[] toArray(IntFunction<A[]> generator);
```

reduce() (Stream (3), IntStream (2), LongStream (2), DoubleStream (2))

Вернет один элемент (конкатенацию строк, сумму, минимальное значение и т. п.).

```
1.  T reduce(
2.          T identity,
3.          BinaryOperator<T> accumulator);
1.  Optional<T> reduce(BinaryOperator<T> accumulator);
1.  <U> U reduce(
2.          U identity,
3.          BiFunction<U, ? super T, U> accumulator,
4.          BinaryOperator<U> combiner);
1.  int reduce(int identity, IntBinaryOperator op);
1.  OptionalInt reduce(IntBinaryOperator op);
1.  long reduce(long identity, LongBinaryOperator op);
1.  OptionalLong reduce(LongBinaryOperator op);
1.  <U> U reduce(
2.          U identity,
3.          BiFunction<U, ? super T, U> accumulator,
4.          BinaryOperator<U> combiner);
1.  double reduce(
2.          double identity,
3.          DoubleBinaryOperator op);
1.  OptionalDouble reduce(DoubleBinaryOperator op);
```

collect() (Stream (2), IntStream, LongStream, DoubleStream)

Преобразовать данные в какой-либо контейнер, например, коллекцию.

collect() принимает на вход Collector<Тип_источника, Тип_аккумулятора, Тип_результата>, который содержит четыре этапа: supplier - инициализация аккумулятора, accumulator - обработка

каждого элемента, combiner - соединение двух аккумуляторов при параллельном выполнении, [finisher] - необязательный метод последней обработки аккумулятора. В Java 8 в классе Collectors реализовано несколько распространённых коллекторов:

- toList(), toCollection(), toSet() - представляют стрим в виде списка, коллекции или множества;
- toConcurrentMap(), toMap() - позволяют преобразовать стрим в Map;
- averagingInt(), averagingDouble(), averagingLong() - возвращают среднее значение;
- summingInt(), summingDouble(), summingLong() - возвращает сумму;
- summarizingInt(), summarizingDouble(), summarizingLong() - возвращают SummaryStatistics с разными агрегатными значениями;
- partitioningBy() - разделяет коллекцию на две части по соответствию условию и возвращает их как Map<Boolean, List>;
- groupingBy() - разделяет коллекцию на несколько частей и возвращает Map<N, List<T>>;

- mapping() - дополнительные преобразования значений для сложных Collector-ов.

Также существует возможность создания собственного коллектора через Collector.of():

```
Collector<String, List<String>, List<String>> toList = Collector.of(
        ArrayList::new,
        List::add,
        (l1, l2) -> { l1.addAll(l2); return l1; }
);
```

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

```
<R, A> R collect(
        Collector<? super T, A, R> collector);
```

```
<R> R collect(
        Supplier<R> supplier,
        ObjIntConsumer<R> accumulator,
        BiConsumer<R, R> combiner);
```

```
<R> R collect(
        Supplier<R> supplier,
        ObjLongConsumer<R> accumulator,
        BiConsumer<R, R> combiner);
```

```
<R> R collect(
        Supplier<R> supplier,
        ObjDoubleConsumer<R> accumulator,
        BiConsumer<R, R> combiner);
```

```
List<String> list =
```

```
2.          Stream.of("Alice", "Bob", "Chuck")
3.              .collect(Collectors.toList());
4.  System.out.println(list);
```

[Alice, Bob, Chuck]

sum() (IntStream, LongStream, DoubleStream)

Подсчет суммы элементов. Частный случай `reduce()`.

```
1.  int sum();
1.  long sum();
1.  double sum();
```

```
1.  int sum = IntStream.range(1, 10).sum();
2.  System.out.println(sum);
```

45

min() (Stream, IntStream, LongStream, DoubleStream)

Найти минимальный элемент. Для объектов используется `Comparator`. Частный случай `reduce()`.

```
1.  Optional<T> min(Comparator<? super T> comparator);
1.  OptionalInt min();
1.  OptionalLong min();
1.  OptionalDouble min();
```

```
1.  Optional<Integer> optionalMin =
2.          Stream.of(5, 9, 3, 8, 1, 6, 4, 7, 2)
3.              .min(Comparator.comparing(
4.                  Integer::valueOf));
5.  int numberMin = optionalMin.get();
6.  System.out.println(numberMin);
```

1

```
1.  OptionalInt optionalMin = "строка".chars().min();
```

```
2.  int numberMin = optionalMin.getAsInt();
3.  System.out.println(numberMin);
```

1072

max() (Stream, IntStream, LongStream, DoubleStream)

Найти максимальный элемент. Для объектов используется `Comparator`. Частный случай `reduce()`.

```
1.  Optional<T> max(Comparator<? super T> comparator);
1.  OptionalInt max();
1.  OptionalLong max();
1.  OptionalDouble max();
```

```
1.  Optional<Integer> optionalMax =
2.          Stream.of(5, 9, 3, 8, 1, 6, 4, 7, 2)
3.              .max(Comparator.comparing(
4.                  Integer::valueOf));
5.  int numberMax = optionalMax.get();
6.  System.out.println(numberMax);
```

9

```
1.  OptionalInt optionalMax = "строка".chars().max();
2.  int numberMax = optionalMax.getAsInt();
3.  System.out.println(numberMax);
```

1090

count() (Stream, IntStream, LongStream, DoubleStream)

Подсчитать количество элементов. Частный случай `reduce()`.

```
1.  long count();
```

```
1.  long count =
2.          Stream.of(5, 9, 3, 8, 1, 6, 4, 7, 2)
3.              .limit(5)
```

```
4.              .count();
5.  System.out.println(count);
```

5

average() (IntStream, LongStream, DoubleStream)

Среднее арифметическое элементов. Частный случай `reduce()`.

```
1.  OptionalDouble average();
```

summaryStatistics() (IntStream, LongStream, DoubleStream)

Выдаст стрим с информацией о количестве, сумме, минимальном, максимальном и среднем значении элементов стрима.

```
1.  IntSummaryStatistics summaryStatistics();
1.  LongSummaryStatistics summaryStatistics();
1.  DoubleSummaryStatistics summaryStatistics();
```

```
1.  List<Integer> naturalNumbers =
2.          Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);
3.  IntSummaryStatistics stats =
4.          naturalNumbers.stream()
5.              .mapToInt((x) -> x)
6.              .summaryStatistics();
7.  System.out.println(stats);
```

IntSummaryStatistics{count=9, sum=45, min=1, average=5,000000, max=9}

anyMatch() (Stream, IntStream, LongStream, DoubleStream)

Хотя бы один элемент удовлетворяет условию `Predicate` (`boolean test(T t)`).

```
1.  boolean anyMatch(Predicate<? super T> predicate);
```

```
1. boolean anyMatch(IntPredicate predicate);
1. boolean anyMatch(LongPredicate predicate);
1. boolean anyMatch(DoublePredicate predicate);
```

```
1. boolean anyMatch =
2.         Stream.of(14, 22, 40, 26)
3.             .anyMatch(i -> i % 2 == 0);
4. System.out.println(anyMatch);
```

true

allMatch()(Stream, IntStream, LongStream, DoubleStream)

Все элементы удовлетворяют условию `Predicate` (`boolean test(T t)`).

```
1. boolean allMatch(Predicate<? super T> predicate);
1. boolean allMatch(IntPredicate predicate);
1. boolean allMatch(LongPredicate predicate);
1. boolean allMatch(DoublePredicate predicate);
```

```
1. boolean anyMatch =
2.         Stream.of(14, 22, 40, 26)
3.             .anyMatch(i -> i % 2 == 0);
4. System.out.println(anyMatch);
```

true

noneMatch() (Stream, IntStream, LongStream, DoubleStream)

Все элементы не удовлетворяют условию `Predicate` (`boolean test(T t)`).

```
1. boolean noneMatch(Predicate<? super T> predicate);
1. boolean noneMatch(IntPredicate predicate);
1. boolean noneMatch(LongPredicate predicate);
1. boolean noneMatch(DoublePredicate predicate);
```

```
boolean noneMatch =
        Stream.of(15, 27, 43, 29)
            .noneMatch(i -> i % 2 == 0);
System.out.println(noneMatch);
```

true

findFirst() (Stream, IntStream, LongStream, DoubleStream)

Вернет первый элемент.

```
Optional<T> findFirst();
OptionalInt findFirst();
OptionalLong findFirst();
OptionalDouble findFirst();
```

```
Optional<String> optionalFirst =
        Stream.of("Alice", "Bob", "Chuck")
            .findFirst();
String stringFirst = optionalFirst.get();
System.out.println(stringFirst);
```

Alice

findAny() (Stream, IntStream, LongStream, DoubleStream)

Вернет какой-то элемент стрима (скорей всего первый).

```
Optional<T> findAny();
OptionalInt findAny();
OptionalLong findAny();
OptionalDouble findAny();
```

```
Optional<String> optionalAny = Stream.of("Alice",
"Bob", "Chuck").findAny();
String stringAny = optionalAny.get();
System.out.println(stringAny);
```

какой-то из элементов

asLongStream() (IntStream)

Преобразует `IntStream` в `LongStream`.

```
1. LongStream asLongStream();
```

asDoubleStream() (IntStream, LongStream)

Преобразует `IntStream` или `LongStream` в `DoubleStream`.

```
1. DoubleStream asDoubleStream();
1. DoubleStream asDoubleStream();
```

boxed() (IntStream, LongStream, DoubleStream)

Преобразует примитивный стрим в объектный с соответствующим примитиву классом оберткой.

```
1. Stream<Integer> boxed();
1. Stream<Long> boxed();
1. Stream<Double> boxed();
```

sequential() (IntStream, LongStream, DoubleStream)

Возвращает последовательный поток.

```
1. @Override
2. IntStream sequential();
1. @Override
2. LongStream sequential();
1. @Override
2. DoubleStream sequential();
```

parallel() (IntStream, LongStream, DoubleStream)

Возвращает параллельный поток.

```
1.  @Override
2.  IntStream parallel();
```

```
1.  @Override
2.  LongStream parallel();
```

```
1.  @Override
2.  DoubleStream parallel();
```

iterator() (IntStream, DoubleStream, LongStream)

Возвращает итератор для элементов этого потока.

```
1.  @Override
2.  PrimitiveIterator.OfInt iterator();
```

```
1.  @Override
2.  PrimitiveIterator.OfDouble iterator();
```

```
1.  @Override
2.  PrimitiveIterator.OfLong iterator();
```

spliterator() (IntStream, LongStream, DoubleStream)

Возвращает сплитератор для элементов этого потока.

```
1.  @Override
2.  Spliterator.OfInt spliterator();
```

```
1.  @Override
2.  Spliterator.OfLong spliterator();
```

```
1.  @Override
2.  Spliterator.OfDouble spliterator();
```

Реализации

Stream

```
1.  package java.util.stream.Stream;
2.
3.  public interface Stream<T>
4.          extends BaseStream<T, Stream<T>> {
5.      ...
6.  }
```

IntStream

```
1.  public interface IntStream
2.          extends BaseStream<Integer, IntStream> {
3.      ...
4.  }
```

LongStream

```
1.  public interface LongStream
2.          extends BaseStream<Long, LongStream> {
3.      ...
4.  }
```

DoubleStream

```
1.  public interface DoubleStream
2.          extends BaseStream<Double, DoubleStream> {
3.      ...
4.  }
```

BaseStream

```
1.  package java.util.stream;
2.
3.  public interface BaseStream<
4.              T, S extends BaseStream<T, S>>
5.          extends AutoCloseable {
6.
7.      Iterator<T> iterator();
8.
9.      Spliterator<T> spliterator();
10.
11.     boolean isParallel();
12.
13.     S sequential();
14.
15.     S parallel();
16.
17.     S unordered();
18.
19.     S onClose(Runnable closeHandler);
20.
21.     @Override
22.     void close();
23. }
```

AutoCloseable

```
1.  package java.lang;
2.
3.  public interface AutoCloseable {
4.
5.      void close() throws Exception;
6.  }
```