Оглавление

[Stream API](#_Toc47185107)

[Что такое и для чего нужны?](#_Toc47185108)

[С какими типами данных работает стрим?](#_Toc47185109)

[Отличия от коллекции](#_Toc47185110)

[Основные части при использовании стрима, порядок …](#_Toc47185111)

[Операции над стримами: какие бывают, разница и почему стримы ленивы?](#_Toc47185112)

[Стрим с точки зрения многопоточности](#_Toc47185113)

[Способы создания стрима](#_Toc47185114)

[Собственными методами](#_Toc47185115)

[builder() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185116)

[empty() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185117)

[of() (Stream (2), IntStream (2), LongStream (2), DoubleStream (2))](#_Toc47185118)

[iterate() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185119)

[generate() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185120)

[range() (IntStream, LongStream)](#_Toc47185121)

[rangeClosed() (IntStream, LongStream)](#_Toc47185122)

[concat() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185123)

[Создание стрима из коллекции или массива](#_Toc47185124)

[stream() (Collection или Arrays)](#_Toc47185125)

[Создание стрима из других источников](#_Toc47185126)

[lines() (BufferedReader)](#_Toc47185127)

[lines() (Files)](#_Toc47185128)

[walk() (Paths), list() (Paths)](#_Toc47185129)

[chars() (String)](#_Toc47185130)

[Создание бесконечных стримов, как сделать их конечными](#_Toc47185131)

[Промежуточные операции](#_Toc47185132)

[filter() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185133)

[map() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185134)

[mapToObj() (IntStream, LongStream, DoubleStream)](#_Toc47185135)

[mapToInt() (Stream, LongStream, DoubleStream)](#_Toc47185136)

[mapToLong() (Stream, IntStream, DoubleStream)](#_Toc47185137)

[mapToDouble() (Stream, IntStream, LongStream)](#_Toc47185138)

[flatMap() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185139)

[flatMapToInt(), flatMapToLong(), flatMapToDouble() (Stream)](#_Toc47185140)

[distinct() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185141)

[sorted() (Stream (2), IntStream, LongStream, DoubleStream)](#_Toc47185142)

[peek() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185143)

[limit() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185144)

[skip() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185145)

[Терминальные операции](#_Toc47185146)

[forEach() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185147)

[forEachOrdered() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185148)

[toArray() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185149)

[reduce() (Stream (3), IntStream (2), LongStream (2), DoubleStream (2))](#_Toc47185150)

[collect() (Stream (2), IntStream, LongStream, DoubleStream)](#_Toc47185151)

[sum() (IntStream, LongStream, DoubleStream)](#_Toc47185152)

[min() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185153)

[max() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185154)

[count() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185155)

[average() (IntStream, LongStream, DoubleStream)](#_Toc47185156)

[summaryStatistics() (IntStream, LongStream, DoubleStream)](#_Toc47185157)

[anyMatch() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185158)

[allMatch()(Stream, IntStream, LongStream, DoubleStream)](#_Toc47185159)

[noneMatch() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185160)

[findFirst() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185161)

[findAny() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185162)

[asLongStream() (IntStream)](#_Toc47185163)

[asDoubleStream() (IntStream, LongStream)](#_Toc47185164)

[boxed() (IntStream, LongStream, DoubleStream)](#_Toc47185165)

[sequential() (IntStream, LongStream, DoubleStream)](#_Toc47185166)

[parallel() (IntStream, LongStream, DoubleStream)](#_Toc47185167)

[iterator() (IntStream, DoubleStream, LongStream)](#_Toc47185168)

[spliterator() (IntStream, LongStream, DoubleStream)](#_Toc47185169)

[Реализации](#_Toc47185170)

[Stream](#_Toc47185171)

[IntStream](#_Toc47185172)

[LongStream](#_Toc47185173)

[DoubleStream](#_Toc47185174)

[BaseStream](#_Toc47185175)

[AutoCloseable](#_Toc47185176)

[builder() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185177)

[empty() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185178)

[of() (Stream (2), IntStream (2), LongStream (2), DoubleStream (2))](#_Toc47185179)

[iterate() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185180)

[generate() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185181)

[range() (IntStream, LongStream)](#_Toc47185182)

[rangeClosed() (IntStream, LongStream)](#_Toc47185183)

[concat() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185184)

[filter() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185185)

[map() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185186)

[mapToObj() (IntStream, LongStream, DoubleStream)](#_Toc47185187)

[mapToInt() (Stream, LongStream, DoubleStream)](#_Toc47185188)

[mapToLong() (Stream, IntStream, DoubleStream)](#_Toc47185189)

[mapToDouble() (Stream, IntStream, LongStream)](#_Toc47185190)

[flatMap() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185191)

[flatMapToInt(), flatMapToLong(), flatMapToDouble() (Stream)](#_Toc47185192)

[distinct() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185193)

[sorted() (Stream (2), IntStream, LongStream, DoubleStream)](#_Toc47185194)

[peek() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185195)

[limit() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185196)

[skip() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185197)

[forEach() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185198)

[forEachOrdered() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185199)

[toArray() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185200)

[reduce() (Stream (3), IntStream (2), LongStream (2), DoubleStream (2))](#_Toc47185201)

[collect() (Stream (2), IntStream, LongStream, DoubleStream)](#_Toc47185202)

[sum() (IntStream, LongStream, DoubleStream)](#_Toc47185203)

[min() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185204)

[max() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185205)

[count() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185206)

[average() (IntStream, LongStream, DoubleStream)](#_Toc47185207)

[summaryStatistics() (IntStream, LongStream, DoubleStream)](#_Toc47185208)

[anyMatch() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185209)

[allMatch()(Stream, IntStream, LongStream, DoubleStream)](#_Toc47185210)

[noneMatch() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185211)

[findFirst() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185212)

[findAny() (Stream, IntStream, LongStream, DoubleStream)](#_Toc47185213)

[asLongStream() (IntStream)](#_Toc47185214)

[asDoubleStream() (IntStream, LongStream)](#_Toc47185215)

[boxed() (IntStream, LongStream, DoubleStream)](#_Toc47185216)

[sequential() (IntStream, LongStream, DoubleStream)](#_Toc47185217)

[parallel() (IntStream, LongStream, DoubleStream)](#_Toc47185218)

[iterator() (IntStream, DoubleStream, LongStream)](#_Toc47185219)

[spliterator() (IntStream, LongStream, DoubleStream)](#_Toc47185220)

# 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](#_Stream) добавили примитивные стримы: [IntStream](#_IntStream), [LongStream](#_LongStream) и [DoubleStream](#_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

## Собственными методами

### [builder() (Stream, IntStream, LongStream, DoubleStream)](#_builder())

Позволяет добавить элементы в стрим при помощи методов вложенного интерфейса Builder с тремя методами add() (добавление элемента), build() (создание из них стрим) и accept() (помощник для add()?).

Похожий метод [of()](#_of()_(Stream_(2),) делает то же, но без помощи интерфейса Builder.

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

x

y

z

### [empty() (Stream, IntStream, LongStream, DoubleStream)](#_empty()_(Stream,_IntStream,)

Создание пустого стрима.

|  |  |
| --- | --- |
| 1.  2. | IntStream empty = IntStream.empty();  System.out.println(empty.count()); |

0

### [of() (Stream (2), IntStream (2), LongStream (2), DoubleStream (2))](#_of()_(Stream_(2),_1)

Создает стрим перечислением элементов. Похож на [builder()](#_builder()_(Stream,_IntStream,), но для создания элементов не использует методы вложенного интерейса Builder.

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

2

4

5

6

8

10

### [iterate() (Stream, IntStream, LongStream, DoubleStream)](#_iterate()_(Stream,_IntStream,_1)

Как и [generate()](#_generate()_(Stream,_IntStream,) 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()](#_limit()_(Stream,_IntStream,).

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

вывод целых чисел от 0 до остановки

### [generate() (Stream, IntStream, LongStream, DoubleStream)](#_generate()_(Stream,_IntStream,_1)

Как и [iterate()](#_iterate()_(Stream,_IntStream,) создает бесконечный стрим, но использует Supplier (T get()). Бесконечный стрим можно сделать конечным терминальной операцией [limit()](#_limit()_(Stream,_IntStream,).

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

вывод случайных чисел типа Double до остановки

### [range() (IntStream, LongStream)](#_range()_(IntStream,_LongStream)_1)

Как и [rangeClosed()](#_rangeClosed()_(IntStream,_LongStrea) создает целый числовой стрим из указанного диапазона чисел, но последний элемент не включен.

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

целые числа от 0 до 100 не включительно

### [rangeClosed() (IntStream, LongStream)](#_rangeClosed()_(IntStream,_LongStrea)

Как и [range()](#_range()_(IntStream,_LongStream)) создает целый числовой стрим из указанного диапазона чисел, но последний элемент будет включен.

|  |  |
| --- | --- |
| 1.  2.  3. | IntStream smallIntegers2 = IntStream.rangeClosed(0, 100);  smallIntegers2.forEach(System.out::println); |

целые числа от 0 до 100 включительно

### [concat() (Stream, IntStream, LongStream, DoubleStream)](#_concat()_(Stream,_IntStream,)

Объединение двух стримов.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | 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

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | 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

## Создание стрима из коллекции или массива

### stream() (Collection или Arrays)

Создать стрим из ассоциативного массива нельзя.

|  |  |
| --- | --- |
| 1.  2.  3. | List<String> list = new ArrayList<>();  Stream<String> streamOfList = list.stream();  streamOfList.forEach(System.out::println); |

элементы списка

|  |  |
| --- | --- |
| 1.  2.  3. | Set<String> set = new HashSet<>();  Stream<String> streamOfSet = set.stream();  streamOfSet.forEach(System.out::println); |

***элементы сета***

|  |  |
| --- | --- |
| 1.  2.  3.  4. | double[] array = new double[5];  DoubleStream streamFromArray =  Arrays.stream(array);  streamFromArray.forEach(System.out::println); |

элементы массива

## Создание стрима из других источников

### lines() (BufferedReader)

Создание стрима из BufferedReader.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | 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.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | 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() (Paths), list() (Paths)

Создание стрима из пути.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | 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

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | Path path = Paths.get("D:\\java\\example");  try (  Stream<Path> streamOfPathWithList =  Files.list(path)) {  streamOfPathWithList.forEach(  System.out::println);  } catch (IOException e) {  e.printStackTrace();  } |

url только содержимого папки

### chars() (String)

Создание стрима из строки.

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

символы строки в виде целого числа

# Создание бесконечных стримов, как сделать их конечными

[iterate()](#_iterate()_(Stream,_IntStream,) и [generate()](#_generate()_(Stream,_IntStream,).

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

вывод целых чисел от 0 до остановки

|  |  |
| --- | --- |
| 1.  2.  3.  4. | DoubleStream randomNumbers =  DoubleStream.generate(Math::random);  randomNumbers.forEach(  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)](#_filter()_(Stream,_IntStream,)

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

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

2

4

## [map() (Stream, IntStream, LongStream, DoubleStream)](#_map()_(Stream,_IntStream,_1)

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

2

4

14

28

### [mapToObj() (IntStream, LongStream, DoubleStream)](#_mapToObj()_(IntStream,_LongStream,)

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

### [mapToInt() (Stream, LongStream, DoubleStream)](#_mapToInt()_(Stream,_LongStream,)

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

### [mapToLong() (Stream, IntStream, DoubleStream)](#_mapToLong()_(Stream,_IntStream,)

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

### [mapToDouble() (Stream, IntStream, LongStream)](#_mapToDouble()_(Stream,_IntStream,)

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

## [flatMap() (Stream, IntStream, LongStream, DoubleStream)](#_flatMap()_(Stream,_IntStream,_1)

В отличии от [map()](#_map()_(Stream,_IntStream,) 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)).

### [flatMapToInt(), flatMapToLong(), flatMapToDouble() (Stream)](#_flatMapToInt(),_flatMapToLong(),_fl)

[flatMap()](#_flatMap()_(Stream,_IntStream,), но преобразовывает к int, long или double. Function возвращает IntStream (IntStream apply(T t)).

## [distinct() (Stream, IntStream, LongStream, DoubleStream)](#_distinct()_(Stream,_IntStream,)

Удаляет дубликаты.

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

1

7

## [sorted() (Stream (2), IntStream, LongStream, DoubleStream)](#_sorted()_(Stream_(2),)

Сортирует поток. Есть метод для сортировки при помощи Comparator.

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

1

4

7

9

## [peek() (Stream, IntStream, LongStream, DoubleStream)](#_peek()_(Stream,_IntStream,_1)

Произвести какое-либо действия над элементами. В отличии от [map()](#_map()_(Stream,_IntStream,) принимает Consumer (void accept(T t)) и не может изменить тип элемента. [forEach()](#_forEach()_(Stream,_IntStream,) — аналогичная терминальная операция.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | Stream<String> nameStream = Stream.of(  "Alice", "Bob", "Chuck");  nameStream  .peek(System.out::println)  .count(); |

Alice

Bob

Chuck

## [limit() (Stream, IntStream, LongStream, DoubleStream)](#_limit()_(Stream,_IntStream,_1)

Обрезает стрим до указанного количества элементов.

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

1

4

## [skip() (Stream, IntStream, LongStream, DoubleStream)](#_skip()_(Stream,_IntStream,)

Пропускает указывает количество элементов.

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

4

24

# Терминальные операции

## [forEach() (Stream, IntStream, LongStream, DoubleStream)](#_forEach()_(Stream,_IntStream,_1)

Произвести какое-либо действия над элементами. Использует Consumer (void accept(T t)). [peek()](#_peek()_(Stream,_IntStream,) аналогичная промежуточная операция.

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

Alice

Bob

Chuck

## [forEachOrdered() (Stream, IntStream, LongStream, DoubleStream)](#_forEachOrdered()_(Stream,_IntStream)

В отличии от [forEach()](#_forEach()_(Stream,_IntStream,) 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.

## [toArray() (Stream, IntStream, LongStream, DoubleStream)](#_toArray()_(Stream,_IntStream,)

Вернет массив.

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

## [reduce() (Stream (3), IntStream (2), LongStream (2), DoubleStream (2))](#_reduce()_(Stream_(3),_1)

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

## [collect() (Stream (2), IntStream, LongStream, DoubleStream)](#_collect()_(Stream_(2),)

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; }

);

|  |  |
| --- | --- |
| 1.  2.  3.  4. | List<String> list =  Stream.of("Alice", "Bob", "Chuck")  .collect(Collectors.toList());  System.out.println(list); |

[Alice, Bob, Chuck]

## [sum() (IntStream, LongStream, DoubleStream)](#_sum()_(IntStream,_LongStream,)

Подсчет суммы элементов. Частный случай [reduce()](#_reduce()_(Stream_(3),).

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

45

## [min() (Stream, IntStream, LongStream, DoubleStream)](#_min()_(Stream,_IntStream,)

Найти минимальный элемент. Для объектов используется Comparator. Частный случай [reduce()](#_reduce()_(Stream_(3),).

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

1

|  |  |
| --- | --- |
| 1.  2.  3. | OptionalInt optionalMin = "строка".chars().min();  int numberMin = optionalMin.getAsInt();  System.out.println(numberMin); |

1072

## [max() (Stream, IntStream, LongStream, DoubleStream)](#_max()_(Stream,_IntStream,)

Найти максимальный элемент. Для объектов используется Comparator. Частный случай [reduce()](#_reduce()_(Stream_(3),).

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

9

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

1090

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

Подсчитать количество элементов. Частный случай [reduce()](#_reduce()_(Stream_(3),).

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | long count =  Stream.of(5, 9, 3, 8, 1, 6, 4, 7, 2)  .limit(5)  .count();  System.out.println(count); |

5

## [average() (IntStream, LongStream, DoubleStream)](#_average()_(IntStream,_LongStream,)

Среднее арифметическое элементов. Частный случай [reduce()](#_reduce()_(Stream_(3),).

## [summaryStatistics() (IntStream, LongStream, DoubleStream)](#_summaryStatistics()_(IntStream,_Lon)

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

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

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

## [anyMatch() (Stream, IntStream, LongStream, DoubleStream)](#_anyMatch()_(Stream,_IntStream,)

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

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

true

## [allMatch()(Stream, IntStream, LongStream, DoubleStream)](#_allMatch()(Stream,_IntStream,_LongS)

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

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

true

## [noneMatch() (Stream, IntStream, LongStream, DoubleStream)](#_noneMatch()_(Stream,_IntStream,)

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

|  |  |
| --- | --- |
| 1.  2.  3.  4. | boolean noneMatch =  Stream.of(15, 27, 43, 29)  .noneMatch(i -> i % 2 == 0);  System.out.println(noneMatch); |

true

## [findFirst() (Stream, IntStream, LongStream, DoubleStream)](#_findFirst()_(Stream,_IntStream,)

Вернет первый элемент.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | Optional<String> optionalFirst =  Stream.of("Alice", "Bob", "Chuck")  .findFirst();  String stringFirst = optionalFirst.get();  System.out.println(stringFirst); |

Alice

## [findAny() (Stream, IntStream, LongStream, DoubleStream)](#_findAny()_(Stream,_IntStream,)

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

|  |  |
| --- | --- |
| 1.  2.  3.  4. | Optional<String> optionalAny = Stream.of("Alice", "Bob", "Chuck").findAny();  String stringAny = optionalAny.get();  System.out.println(stringAny); |

какой‑то из элементов

## [asLongStream() (IntStream)](#_asLongStream()_(IntStream))

Преобразует IntStream в LongStream.

## [asDoubleStream() (IntStream, LongStream)](#_asDoubleStream()_(IntStream,_LongSt)

Преобразует IntStream или LongStream в DoubleStream.

## [boxed() (IntStream, LongStream, DoubleStream)](#_boxed()_(IntStream,_LongStream,)

Преобразует примитивный стрим в объектный с соответствующим примитиву классом оберткой.

## [sequential() (IntStream, LongStream, DoubleStream)](#_sequential()_(IntStream,_LongStream)

Возвращает последовательный поток.

## [parallel() (IntStream, LongStream, DoubleStream)](#_parallel()_(IntStream,_LongStream,)

Возвращает параллельный поток.

## [iterator() (IntStream, DoubleStream, LongStream)](#_iterator()_(IntStream,_DoubleStream)

Возвращает итератор для элементов этого потока.

## [spliterator() (IntStream, LongStream, DoubleStream)](#_spliterator()_(IntStream,_LongStrea)

Возвращает сплитератор для элементов этого потока.

# Реализации

## Stream

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6. | package java.util.stream.Stream;  public interface Stream<T>  extends [BaseStream<T, Stream<T>>](#_BaseStream) {  ...  } |

## IntStream

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6. | package java.util.stream.Stream;  public interface IntStream  extends [BaseStream<Integer, IntStream>](#_BaseStream) {  ...  } |

## LongStream

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6. | package java.util.stream.Stream;  public interface LongStream  extends [BaseStream<Long, LongStream>](#_BaseStream) {  ...  } |

## DoubleStream

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6. | package java.util.stream.Stream;  public interface DoubleStream  extends [BaseStream<Double, DoubleStream>](#_BaseStream) {  ...  } |

## BaseStream

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

## AutoCloseable

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

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

|  |  |
| --- | --- |
| 1.  2.  3. | public static<T> Builder<T> builder() {  return new Streams.StreamBuilderImpl<>();  } |
| 1.  2.  3. | public static Builder builder() {  return new Streams.IntStreamBuilderImpl();  } |
| 1.  2.  3. | public static Builder builder() {  return new Streams.LongStreamBuilderImpl();  } |
| 1.  2.  3. | public static Builder builder() {  return new Streams.DoubleStreamBuilderImpl();  } |

Builder (Stream, IntStream, LongStream, DoubleStream)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | public interface Builder<T> extends Consumer<T> {  @Override  void accept(T t);  default Builder<T> add(T t) {  accept(t);  return this;  }  Stream<T> build();  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | public interface Builder extends IntConsumer {  @Override  void accept(int t);  default Builder add(int t) {  accept(t);  return this;  }  IntStream build();  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | public interface Builder extends LongConsumer {  @Override  void accept(long t);  default Builder add(long t) {  accept(t);  return this;  }  LongStream build();  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | public interface Builder extends DoubleConsumer {  @Override  void accept(double t);  default Builder add(double t) {  accept(t);  return this;  }  DoubleStream build();  } |

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

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | public static<T> Stream<T> empty() {  return StreamSupport.stream(  Spliterators.<T>emptySpliterator(),  false);  } |
| 1.  2.  3.  4.  5. | public static IntStream empty() {  return StreamSupport.intStream(  Spliterators.emptyIntSpliterator(),  false);  } |
| 1.  2.  3.  4.  5. | public static LongStream empty() {  return StreamSupport.longStream(  Spliterators.emptyLongSpliterator(),  false);  } |
| 1.  2.  3.  4.  5. | public static DoubleStream empty() {  return StreamSupport.doubleStream(  Spliterators.emptyDoubleSpliterator(),  false);  } |

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

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | public static<T> Stream<T> of(T t) {  return StreamSupport.stream(  new Streams.StreamBuilderImpl<>(t),  false);  } |
| 1.  2.  3.  4.  5. | @SafeVarargs  @SuppressWarnings("varargs")  public static<T> Stream<T> of(T... values) {  return Arrays.stream(values);  } |
| 1.  2.  3.  4.  5. | public static IntStream of(int t) {  return StreamSupport.intStream(  new Streams.IntStreamBuilderImpl(t),  false);  } |
| 1.  2.  3. | public static IntStream of(int... values) {  return Arrays.stream(values);  } |
| 1.  2.  3.  4.  5. | public static LongStream of(long t) {  return StreamSupport.longStream(  new Streams.LongStreamBuilderImpl(t),  false);  } |
| 1.  2.  3. | public static LongStream of(long... values) {  return Arrays.stream(values);  } |
| 1.  2.  3.  4.  5.  6. | public static DoubleStream of(double t) {  return StreamSupport.doubleStream(  new Streams.DoubleStreamBuilderImpl(  t),  false);  } |
| 1.  2.  3. | public static DoubleStream of(double... values) {  return Arrays.stream(values);  } |

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

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29. | public static<T> Stream<T> iterate(  final T seed,  final UnaryOperator<T> f) {  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);  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28. | 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);  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28. | 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);  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28. | public static DoubleStream iterate(  final double seed,  final DoubleUnaryOperator f) {  Objects.requireNonNull(f);  final PrimitiveIterator.OfDouble iterator =  new PrimitiveIterator.OfDouble() {  double t = seed;  @Override  public boolean hasNext() {  return true;  }  @Override  public double nextDouble() {  double v = t;  t = f.applyAsDouble(t);  return v;  }  };  return StreamSupport.doubleStream(  Spliterators.spliteratorUnknownSize(  iterator,  Spliterator.ORDERED  | Spliterator.IMMUTABLE  | Spliterator.NONNULL),  false);  } |

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

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | public static<T> Stream<T> generate(  Supplier<T> s) {  Objects.requireNonNull(s);  return StreamSupport.stream(  new StreamSpliterators  .InfiniteSupplyingSpliterator  .OfRef<>(Long.MAX\_VALUE, s),  false);  } |
| 1.  2.  3.  4.  5.  6.  7.  8. | public static IntStream generate(IntSupplier s) {  Objects.requireNonNull(s);  return StreamSupport.intStream(  new StreamSpliterators  .InfiniteSupplyingSpliterator  .OfInt(Long.MAX\_VALUE, s),  false);  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | public static LongStream generate(  LongSupplier s) {  Objects.requireNonNull(s);  return StreamSupport.longStream(  new StreamSpliterators  .InfiniteSupplyingSpliterator  .OfLong(Long.MAX\_VALUE, s),  false);  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | public static DoubleStream generate(  DoubleSupplier s) {  Objects.requireNonNull(s);  return StreamSupport.doubleStream(  new StreamSpliterators  .InfiniteSupplyingSpliterator  .OfDouble(Long.MAX\_VALUE, s),  false);  } |

## range() (IntStream, LongStream)

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

## rangeClosed() (IntStream, LongStream)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14. | public static IntStream rangeClosed(  int startInclusive,  int endInclusive) {  if (startInclusive > endInclusive) {  return empty();  } else {  return StreamSupport.intStream(  new Streams.RangeIntSpliterator(  startInclusive,  endInclusive,  true),  false);  }  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26. | public static LongStream rangeClosed(  long startInclusive,  final long endInclusive) {  if (startInclusive > endInclusive) {  return empty();  } else if (  endInclusive - startInclusive + 1  <= 0) {  long m =  startInclusive  + Long.divideUnsigned(  endInclusive - startInclusive,  2)  + 1;  return concat(  range(startInclusive, m),  rangeClosed(m, endInclusive));  } else {  return StreamSupport.longStream(  new Streams.RangeLongSpliterator(  startInclusive,  endInclusive,  true),  false);  }  } |

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

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17. | 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));  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16. | 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));  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16. | 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));  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18. | 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));  } |

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

|  |  |
| --- | --- |
| 1. | Stream<T> filter(Predicate<? super T> predicate); |
| 1. | IntStream filter(IntPredicate predicate); |
| 1. | LongStream filter(LongPredicate predicate); |
| 1. | DoubleStream filter(DoublePredicate predicate); |

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

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

## mapToObj() (IntStream, LongStream, DoubleStream)

|  |  |
| --- | --- |
| 1.  2. | <U> Stream<U> mapToObj(  IntFunction<? extends U> mapper); |
| 1.  2. | <U> Stream<U> mapToObj(  LongFunction<? extends U> mapper); |
| 1.  2. | <U> Stream<U> mapToObj(  DoubleFunction<? extends U> mapper); |

## mapToInt() (Stream, LongStream, DoubleStream)

|  |  |
| --- | --- |
| 1.  2. | IntStream mapToInt(  ToIntFunction<? super T> mapper); |
| 1. | IntStream mapToInt(LongToIntFunction mapper); |
| 1. | IntStream mapToInt(DoubleToIntFunction mapper); |

## mapToLong() (Stream, IntStream, DoubleStream)

|  |  |
| --- | --- |
| 1.  2. | LongStream mapToLong(  ToLongFunction<? super T> mapper); |
| 1. | LongStream mapToLong(IntToLongFunction mapper); |
| 1. | LongStream mapToLong(DoubleToLongFunction mapper); |

## mapToDouble() (Stream, IntStream, LongStream)

|  |  |
| --- | --- |
| 1.  2. | DoubleStream mapToDouble(  ToDoubleFunction<? super T> mapper); |
| 1.  2. | DoubleStream mapToDouble(  IntToDoubleFunction mapper); |
| 1.  2. | DoubleStream mapToDouble(  LongToDoubleFunction mapper); |

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

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | <R> Stream<R> flatMap(  Function<  ? super T,  ? extends Stream<? extends R>>  mapper); |
| 1.  2. | IntStream flatMap(  IntFunction<? extends IntStream> mapper); |
| 1.  2.  3. | LongStream flatMap(  LongFunction<? extends LongStream>  mapper); |
| 1.  2.  3. | DoubleStream flatMap(  DoubleFunction<? extends DoubleStream>  mapper); |

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

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | IntStream flatMapToInt(  Function<  ? super T,  ? extends IntStream>  mapper); |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | 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();  } |
| 1.  2.  3.  4.  5. | LongStream flatMapToLong(  Function<  ? super T,  ? extends LongStream>  mapper); |
| 1.  2.  3.  4.  5. | DoubleStream flatMapToDouble(  Function<  ? super T,  ? extends DoubleStream>  mapper); |

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

|  |  |
| --- | --- |
| 1. | Stream<T> distinct(); |
| 1. | IntStream distinct(); |
| 1. | LongStream distinct(); |
| 1. | DoubleStream distinct(); |

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

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

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

|  |  |
| --- | --- |
| 1. | Stream<T> peek(Consumer<? super T> action); |
| 1. | IntStream peek(IntConsumer action); |
| 1. | LongStream peek(LongConsumer action); |
| 1. | DoubleStream peek(DoubleConsumer action); |

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

|  |  |
| --- | --- |
| 1. | Stream<T> limit(long maxSize); |
| 1. | IntStream limit(long maxSize); |
| 1. | LongStream limit(long maxSize); |
| 1. | DoubleStream limit(long maxSize); |

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

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

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

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

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

|  |  |
| --- | --- |
| 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)

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

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

|  |  |
| --- | --- |
| 1.  2.  3. | T reduce(  T identity,  BinaryOperator<T> accumulator); |
| 1. | Optional<T> reduce(BinaryOperator<T> accumulator); |
| 1.  2.  3.  4. | <U> U reduce(  U identity,  BiFunction<U, ? super T, U> accumulator,  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.  2.  3.  4. | <U> U reduce(  U identity,  BiFunction<U, ? super T, U> accumulator,  BinaryOperator<U> combiner); |
| 1.  2.  3. | double reduce(  double identity,  DoubleBinaryOperator op); |
| 1. | OptionalDouble reduce(DoubleBinaryOperator op); |

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

|  |  |
| --- | --- |
| 1.  2.  3.  4. | <R> R collect(  Supplier<R> supplier,  BiConsumer<R, ? super T> accumulator,  BiConsumer<R, R> combiner); |
| 1.  2. | <R, A> R collect(  Collector<? super T, A, R> collector); |
| 1.  2.  3.  4. | <R> R collect(  Supplier<R> supplier,  ObjIntConsumer<R> accumulator,  BiConsumer<R, R> combiner); |
| 1.  2.  3.  4. | <R> R collect(  Supplier<R> supplier,  ObjLongConsumer<R> accumulator,  BiConsumer<R, R> combiner); |
| 1.  2.  3.  4. | <R> R collect(  Supplier<R> supplier,  ObjDoubleConsumer<R> accumulator,  BiConsumer<R, R> combiner); |

## sum() (IntStream, LongStream, DoubleStream)

|  |  |
| --- | --- |
| 1. | int sum(); |
| 1. | long sum(); |
| 1. | double sum(); |

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

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

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

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

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

|  |  |
| --- | --- |
| 1. | long count(); |

## average() (IntStream, LongStream, DoubleStream)

|  |  |
| --- | --- |
| 1. | OptionalDouble average(); |

## summaryStatistics() (IntStream, LongStream, DoubleStream)

|  |  |
| --- | --- |
| 1. | IntSummaryStatistics summaryStatistics(); |
| 1. | LongSummaryStatistics summaryStatistics(); |
| 1. | DoubleSummaryStatistics summaryStatistics(); |

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

|  |  |
| --- | --- |
| 1. | boolean anyMatch(Predicate<? super T> predicate); |
| 1. | boolean anyMatch(IntPredicate predicate); |
| 1. | boolean anyMatch(LongPredicate predicate); |
| 1. | boolean anyMatch(DoublePredicate predicate); |

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

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

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

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

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

|  |  |
| --- | --- |
| 1. | Optional<T> findFirst(); |
| 1. | OptionalInt findFirst(); |
| 1. | OptionalLong findFirst(); |
| 1. | OptionalDouble findFirst(); |

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

|  |  |
| --- | --- |
| 1. | Optional<T> findAny(); |
| 1. | OptionalInt findAny(); |
| 1. | OptionalLong findAny(); |
| 1. | OptionalDouble findAny(); |

## asLongStream() (IntStream)

|  |  |
| --- | --- |
| 1. | LongStream asLongStream(); |

## asDoubleStream() (IntStream, LongStream)

|  |  |
| --- | --- |
| 1. | DoubleStream asDoubleStream(); |
| 1. | DoubleStream asDoubleStream(); |

## boxed() (IntStream, LongStream, DoubleStream)

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

## sequential() (IntStream, LongStream, DoubleStream)

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

## parallel() (IntStream, LongStream, DoubleStream)

|  |  |
| --- | --- |
| 1.  2. | @Override  IntStream parallel(); |
| 1.  2. | @Override  LongStream parallel(); |
| 1.  2. | @Override  DoubleStream parallel(); |

## iterator() (IntStream, DoubleStream, LongStream)

|  |  |
| --- | --- |
| 1.  2. | @Override  PrimitiveIterator.OfInt iterator(); |
| 1.  2. | @Override  PrimitiveIterator.OfDouble iterator(); |
| 1.  2. | @Override  PrimitiveIterator.OfLong iterator(); |

## spliterator() (IntStream, LongStream, DoubleStream)

|  |  |
| --- | --- |
| 1.  2. | @Override  Spliterator.OfInt spliterator(); |
| 1.  2. | @Override  Spliterator.OfLong spliterator(); |
| 1.  2. | @Override  Spliterator.OfDouble spliterator(); |