Module java.base
Package java.util.stream

Interface Stream<T>

- Type Parameters:
  `T` - the type of the stream elements

All Superinterfaces:
`AutoCloseable`, `BaseStream<T,Stream<T>>`

```
public interface Stream<T>
extends BaseStream<T,Stream<T>>
```

A sequence of elements supporting sequential and parallel aggregate operations. The following example illustrates an aggregate operation using `Stream` and `IntStream`:

```
int sum = widgets.stream()
                 .filter(w -> w.getColor() == RED)
                 .mapToInt(w -> w.getWeight())
                 .sum();
```

In this example, `widgets` is a `Collection<Widget>`. We create a stream of `Widget` objects via `Collection.stream()`, filter it to produce a stream containing only the red widgets, and then transform it into a stream of `int` values representing the weight of each red widget. Then this stream is summed to produce a total weight.

In addition to `Stream`, which is a stream of object references, there are primitive specializations for `IntStream`, `LongStream`, and `DoubleStream`, all of which are referred to as "streams" and conform to the characteristics and restrictions described here.

To perform a computation, stream operations are composed into a stream pipeline. A stream pipeline consists of a source (which might be an array, a collection, a generator function, an I/O channel, etc), zero or more intermediate operations (which transform a stream into another stream, such as `filter(Predicate)`), and a terminal operation (which produces a result or side-effect, such as `count()` or `forEach(Consumer)`). Streams are lazy; computation on the source data is only performed when the terminal operation is initiated, and source elements are consumed only as needed.

A stream implementation is permitted significant latitude in optimizing the computation of the result. For example, a stream implementation is free to elide operations (or entire stages) from a stream pipeline -- and therefore elide invocation of behavioral parameters -- if it can prove that it would not affect the result of the computation. This means that side-effects of behavioral parameters may not always be executed and should not be relied upon, unless otherwise specified (such as by the terminal operations `forEach` and `forEachOrdered`). (For a specific example of such an optimization, see the API note documented on the `count()` operation. For more detail, see the side-effects section of the stream package documentation.)

Collections and streams, while bearing some superficial similarities, have different goals. Collections are primarily concerned with the efficient management of, and access to, their elements. By contrast, streams do not provide a means to directly access or manipulate their elements, and are instead concerned with declaratively describing their source and the computational operations which will be performed in aggregate on that source. However, if the provided stream operations do not offer the desired functionality, the `BaseStream.iterator()` and `BaseStream.spliterator()` operations can be used to perform a controlled traversal.

A stream pipeline, like the "widgets" example above, can be viewed as a query on the stream source. Unless the source was explicitly designed for concurrent modification (such as a `ConcurrentHashMap`),

unpredictable or erroneous behavior may result from modifying the stream source while it is being queried.

Most stream operations accept parameters that describe user-specified behavior, such as the lambda expression `w -> w.getWeight()` passed to `mapToInt` in the example above. To preserve correct behavior, these behavioral parameters:

- must be non-interfering (they do not modify the stream source); and
- in most cases must be stateless (their result should not depend on any state that might change during execution of the stream pipeline).

Such parameters are always instances of a functional interface such as `Function`, and are often lambda expressions or method references. Unless otherwise specified these parameters must be non-null.

A stream should be operated on (invoking an intermediate or terminal stream operation) only once. This rules out, for example, "forked" streams, where the same source feeds two or more pipelines, or multiple traversals of the same stream. A stream implementation may throw `IllegalStateException` if it detects that the stream is being reused. However, since some stream operations may return their receiver rather than a new stream object, it may not be possible to detect reuse in all cases.

Streams have a `BaseStream.close()` method and implement `AutoCloseable`. Operating on a stream after it has been closed will throw `IllegalStateException`. Most stream instances do not actually need to be closed after use, as they are backed by collections, arrays, or generating functions, which require no special resource management. Generally, only streams whose source is an IO channel, such as those returned by `Files.lines(Path)`, will require closing. If a stream does require closing, it must be opened as a resource within a try-with-resources statement or similar control structure to ensure that it is closed promptly after its operations have completed.

Stream pipelines may execute either sequentially or in parallel. This execution mode is a property of the stream. Streams are created with an initial choice of sequential or parallel execution. (For example, `Collection.stream()` creates a sequential stream, and `Collection.parallelStream()` creates a parallel one.) This choice of execution mode may be modified by the `BaseStream.sequential()` or `BaseStream.parallel()` methods, and may be queried with the `BaseStream.isParallel()` method.

Since:
1.8
See Also:
`IntStream`, `LongStream`, `DoubleStream`, java.util.stream

Nested Class Summary

Nested Classes

Modifier and Type	Interface	Description
static interface	`Stream.Builder<T>`	A mutable builder for a `Stream`.

Method Summary

All Methods Static Methods Instance Methods Abstract Methods Default Methods

Modifier and Type	Method	Description
boolean	`allMatch(Predicate<? super T> predicate)`	Returns whether all elements of this stream match the provided predicate.
boolean	`anyMatch(Predicate<? super T> predicate)`	Returns whether any elements of this stream match the provided predicate.
static <T> `Stream.Builder<T>`	`builder()`	Returns a builder for a `Stream`.
<R> R	`collect(Supplier<R> supplier, BiConsumer<R,? super`	Performs a mutable reduction operation on the elements of this stream.

	T> accumulator, BiConsumer<R,R> combiner)	
<R,A> R	collect(Collector<? super T,A,R> collector)	Performs a mutable reduction operation on the elements of this stream using a Collector.
static <T> Stream<T>	concat(Stream<? extends T> a, Stream<? extends T> b)	Creates a lazily concatenated stream whose elements are all the elements of the first stream followed by all the elements of the second stream.
long	count()	Returns the count of elements in this stream.
Stream<T>	distinct()	Returns a stream consisting of the distinct elements (according to Object.equals(Object)) of this stream.
default Stream<T>	dropWhile(Predicate<? super T> predicate)	Returns, if this stream is ordered, a stream consisting of the remaining elements of this stream after dropping the longest prefix of elements that match the given predicate.
static <T> Stream<T>	empty()	Returns an empty sequential Stream.
Stream<T>	filter(Predicate<? super T> predicate)	Returns a stream consisting of the elements of this stream that match the given predicate.
Optional<T>	findAny()	Returns an Optional describing some element of the stream, or an empty Optional if the stream is empty.
Optional<T>	findFirst()	Returns an Optional describing the first element of this stream, or an empty Optional if the stream is empty.
<R> Stream<R>	flatMap(Function<? super T,? extends Stream<? extends R>> mapper)	Returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.
DoubleStream	flatMapToDouble(Function<? super T,? extends DoubleStream> mapper)	Returns an DoubleStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.
IntStream	flatMapToInt(Function<? super T,? extends IntStream> mapper)	Returns an IntStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.
LongStream	flatMapToLong(Function<? super T,? extends LongStream> mapper)	Returns an LongStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.
void	forEach(Consumer<? super T> action)	Performs an action for each element of this stream.
void	forEachOrdered(Consumer<? super T> action)	Performs an action for each element of this stream, in the encounter order of the stream if the stream has a defined encounter order.
static <T> Stream<T>	generate(Supplier<? extends	Returns an infinite sequential unordered

	T> s)	stream where each element is generated by the provided Supplier.
static <T> Stream<T>	iterate(T seed, Predicate<? super T> hasNext, UnaryOperator<T> next)	Returns a sequential ordered Stream produced by iterative application of the given next function to an initial element, conditioned on satisfying the given hasNext predicate.
static <T> Stream<T>	iterate(T seed, UnaryOperator<T> f)	Returns an infinite sequential ordered Stream produced by iterative application of a function f to an initial element seed, producing a Stream consisting of seed, f(seed), f(f(seed)), etc.
Stream<T>	limit(long maxSize)	Returns a stream consisting of the elements of this stream, truncated to be no longer than maxSize in length.
<R> Stream<R>	map(Function<? super T,? extends R> mapper)	Returns a stream consisting of the results of applying the given function to the elements of this stream.
DoubleStream	mapToDouble(ToDoubleFunction<? super T> mapper)	Returns a DoubleStream consisting of the results of applying the given function to the elements of this stream.
IntStream	mapToInt(ToIntFunction<? super T> mapper)	Returns an IntStream consisting of the results of applying the given function to the elements of this stream.
LongStream	mapToLong(ToLongFunction<? super T> mapper)	Returns a LongStream consisting of the results of applying the given function to the elements of this stream.
Optional<T>	max(Comparator<? super T> comparator)	Returns the maximum element of this stream according to the provided Comparator.
Optional<T>	min(Comparator<? super T> comparator)	Returns the minimum element of this stream according to the provided Comparator.
boolean	noneMatch(Predicate<? super T> predicate)	Returns whether no elements of this stream match the provided predicate.
static <T> Stream<T>	of(T t)	Returns a sequential Stream containing a single element.
static <T> Stream<T>	of(T... values)	Returns a sequential ordered stream whose elements are the specified values.
static <T> Stream<T>	ofNullable(T t)	Returns a sequential Stream containing a single element, if non-null, otherwise returns an empty Stream.
Stream<T>	peek(Consumer<? super T> action)	Returns a stream consisting of the elements of this stream, additionally performing the provided action on each element as elements are consumed from the resulting stream.
Optional<T>	reduce (BinaryOperator<T> accumulator)	Performs a reduction on the elements of this stream, using an associative accumulation function, and returns an Optional describing the reduced value, if any.
T	reduce(T identity, BinaryOperator<T> accumulator)	Performs a reduction on the elements of this stream, using the provided identity value and an associative accumulation function, and returns the reduced value.
<U> U	reduce(U identity,	Performs a reduction on the elements of

	BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)	this stream, using the provided identity, accumulation and combining functions.
Stream<T>	skip(long n)	Returns a stream consisting of the remaining elements of this stream after discarding the first n elements of the stream.
Stream<T>	sorted()	Returns a stream consisting of the elements of this stream, sorted according to natural order.
Stream<T>	sorted(Comparator<? super T> comparator)	Returns a stream consisting of the elements of this stream, sorted according to the provided Comparator.
default Stream<T>	takeWhile(Predicate<? super T> predicate)	Returns, if this stream is ordered, a stream consisting of the longest prefix of elements taken from this stream that match the given predicate.
Object[]	toArray()	Returns an array containing the elements of this stream.
<A> A[]	toArray (IntFunction<A[]> generator)	Returns an array containing the elements of this stream, using the provided generator function to allocate the returned array, as well as any additional arrays that might be required for a partitioned execution or for resizing.

- **Methods inherited from interface java.util.stream.BaseStream**

  close, isParallel, iterator, onClose, parallel, sequential, spliterator, unordered

Method Detail

filter

```
Stream<T> filter(Predicate<? super T> predicate)
```

Returns a stream consisting of the elements of this stream that match the given predicate.

This is an intermediate operation.

Parameters:
predicate - a non-interfering, stateless predicate to apply to each element to determine if it should be included
Returns:
the new stream

map

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

Returns a stream consisting of the results of applying the given function to the elements of this stream.

This is an intermediate operation.

Type Parameters:
R - The element type of the new stream
Parameters:
mapper - a non-interfering, stateless function to apply to each element
Returns:

the new stream

- **mapToInt**

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

  Returns an `IntStream` consisting of the results of applying the given function to the elements of this stream.

  This is an intermediate operation.

  Parameters:
  `mapper` - a non-interfering, stateless function to apply to each element
  Returns:
  the new stream

- **mapToLong**

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

  Returns a `LongStream` consisting of the results of applying the given function to the elements of this stream.

  This is an intermediate operation.

  Parameters:
  `mapper` - a non-interfering, stateless function to apply to each element
  Returns:
  the new stream

- **mapToDouble**

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

  Returns a `DoubleStream` consisting of the results of applying the given function to the elements of this stream.

  This is an intermediate operation.

  Parameters:
  `mapper` - a non-interfering, stateless function to apply to each element
  Returns:
  the new stream

- **flatMap**

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

  Returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is `closed` after its contents have been placed into this stream. (If a mapped stream is `null` an empty stream is used, instead.)

  This is an intermediate operation.

  API Note:
  The `flatMap()` operation has the effect of applying a one-to-many transformation to the elements of the stream, and then flattening the resulting elements into a new stream.

  Examples.

If `orders` is a stream of purchase orders, and each purchase order contains a collection of line items, then the following produces a stream containing all the line items in all the orders:

```
orders.flatMap(order -> order.getLineItems().stream())...
```

If `path` is the path to a file, then the following produces a stream of the `words` contained in that file:

```
Stream<String> lines = Files.lines(path, StandardCharsets.UTF_8);
Stream<String> words = lines.flatMap(line -> Stream.of(line.split(" +")));
```

The `mapper` function passed to `flatMap` splits a line, using a simple regular expression, into an array of words, and then creates a stream of words from that array.

Type Parameters:

`R` - The element type of the new stream

Parameters:

`mapper` - a non-interfering, stateless function to apply to each element which produces a stream of new values

Returns:

the new stream

- **flatMapToInt**

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

Returns an `IntStream` consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is `closed` after its contents have been placed into this stream. (If a mapped stream is `null` an empty stream is used, instead.)

This is an intermediate operation.

Parameters:

`mapper` - a non-interfering, stateless function to apply to each element which produces a stream of new values

Returns:

the new stream

See Also:

`flatMap(Function)`

- **flatMapToLong**

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

Returns an `LongStream` consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is `closed` after its contents have been placed into this stream. (If a mapped stream is `null` an empty stream is used, instead.)

This is an intermediate operation.

Parameters:

`mapper` - a non-interfering, stateless function to apply to each element which produces a stream of new values

Returns:

the new stream

See Also:

`flatMap(Function)`

- **flatMapToDouble**

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

  Returns an `DoubleStream` consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is `closed` after its contents have placed been into this stream. (If a mapped stream is `null` an empty stream is used, instead.)

  This is an intermediate operation.

  Parameters:
  `mapper` - a non-interfering, stateless function to apply to each element which produces a stream of new values
  Returns:
  the new stream
  See Also:
  `flatMap(Function)`

- **distinct**

  `Stream<T> distinct()`

  Returns a stream consisting of the distinct elements (according to `Object.equals(Object)`) of this stream.

  For ordered streams, the selection of distinct elements is stable (for duplicated elements, the element appearing first in the encounter order is preserved.) For unordered streams, no stability guarantees are made.

  This is a stateful intermediate operation.

  API Note:
  Preserving stability for `distinct()` in parallel pipelines is relatively expensive (requires that the operation act as a full barrier, with substantial buffering overhead), and stability is often not needed. Using an unordered stream source (such as `generate(Supplier)`) or removing the ordering constraint with `BaseStream.unordered()` may result in significantly more efficient execution for `distinct()` in parallel pipelines, if the semantics of your situation permit. If consistency with encounter order is required, and you are experiencing poor performance or memory utilization with `distinct()` in parallel pipelines, switching to sequential execution with `BaseStream.sequential()` may improve performance.
  Returns:
  the new stream

- **sorted**

  `Stream<T> sorted()`

  Returns a stream consisting of the elements of this stream, sorted according to natural order. If the elements of this stream are not `Comparable`, a `java.lang.ClassCastException` may be thrown when the terminal operation is executed.

  For ordered streams, the sort is stable. For unordered streams, no stability guarantees are made.

  This is a stateful intermediate operation.

  Returns:
  the new stream

- **sorted**

`Stream<T> sorted(Comparator<? super T> comparator)`

Returns a stream consisting of the elements of this stream, sorted according to the provided `Comparator`.

For ordered streams, the sort is stable. For unordered streams, no stability guarantees are made.

This is a stateful intermediate operation.

Parameters:
`comparator` - a non-interfering, stateless `Comparator` to be used to compare stream elements
Returns:
the new stream

- **peek**

`Stream<T> peek(Consumer<? super T> action)`

Returns a stream consisting of the elements of this stream, additionally performing the provided action on each element as elements are consumed from the resulting stream.

This is an intermediate operation.

For parallel stream pipelines, the action may be called at whatever time and in whatever thread the element is made available by the upstream operation. If the action modifies shared state, it is responsible for providing the required synchronization.

API Note:
This method exists mainly to support debugging, where you want to see the elements as they flow past a certain point in a pipeline:

```
Stream.of("one", "two", "three", "four")
    .filter(e -> e.length() > 3)
    .peek(e -> System.out.println("Filtered value: " + e))
    .map(String::toUpperCase)
    .peek(e -> System.out.println("Mapped value: " + e))
    .collect(Collectors.toList());
```

In cases where the stream implementation is able to optimize away the production of some or all the elements (such as with short-circuiting operations like `findFirst`, or in the example described in `count()`), the action will not be invoked for those elements.

Parameters:
`action` - a non-interfering action to perform on the elements as they are consumed from the stream
Returns:
the new stream

- **limit**

`Stream<T> limit(long maxSize)`

Returns a stream consisting of the elements of this stream, truncated to be no longer than `maxSize` in length.

This is a short-circuiting stateful intermediate operation.

API Note:
While `limit()` is generally a cheap operation on sequential stream pipelines, it can be quite expensive on ordered parallel pipelines, especially for large values of `maxSize`, since

`limit(n)` is constrained to return not just any n elements, but the first n elements in the encounter order. Using an unordered stream source (such as `generate(Supplier)`) or removing the ordering constraint with `BaseStream.unordered()` may result in significant speedups of `limit()` in parallel pipelines, if the semantics of your situation permit. If consistency with encounter order is required, and you are experiencing poor performance or memory utilization with `limit()` in parallel pipelines, switching to sequential execution with `BaseStream.sequential()` may improve performance.

Parameters:

`maxSize` - the number of elements the stream should be limited to

Returns:

the new stream

Throws:

`IllegalArgumentException` - if `maxSize` is negative

- **skip**

`Stream<T> skip(long n)`

Returns a stream consisting of the remaining elements of this stream after discarding the first `n` elements of the stream. If this stream contains fewer than `n` elements then an empty stream will be returned.

This is a stateful intermediate operation.

API Note:

While `skip()` is generally a cheap operation on sequential stream pipelines, it can be quite expensive on ordered parallel pipelines, especially for large values of `n`, since `skip(n)` is constrained to skip not just any n elements, but the first n elements in the encounter order. Using an unordered stream source (such as `generate(Supplier)`) or removing the ordering constraint with `BaseStream.unordered()` may result in significant speedups of `skip()` in parallel pipelines, if the semantics of your situation permit. If consistency with encounter order is required, and you are experiencing poor performance or memory utilization with `skip()` in parallel pipelines, switching to sequential execution with `BaseStream.sequential()` may improve performance.

Parameters:

`n` - the number of leading elements to skip

Returns:

the new stream

Throws:

`IllegalArgumentException` - if `n` is negative

- **takeWhile**

`default Stream<T> takeWhile(Predicate<? super T> predicate)`

Returns, if this stream is ordered, a stream consisting of the longest prefix of elements taken from this stream that match the given predicate. Otherwise returns, if this stream is unordered, a stream consisting of a subset of elements taken from this stream that match the given predicate.

If this stream is ordered then the longest prefix is a contiguous sequence of elements of this stream that match the given predicate. The first element of the sequence is the first element of this stream, and the element immediately following the last element of the sequence does not match the given predicate.

If this stream is unordered, and some (but not all) elements of this stream match the given predicate, then the behavior of this operation is nondeterministic; it is free to take any subset of matching elements (which includes the empty set).

Independent of whether this stream is ordered or unordered if all elements of this stream match the given predicate then this operation takes all elements (the result is the same as the input), or if no elements of the stream match the given predicate then no elements are taken (the result is an empty stream).

This is a short-circuiting stateful intermediate operation.

API Note:

While `takeWhile()` is generally a cheap operation on sequential stream pipelines, it can be quite expensive on ordered parallel pipelines, since the operation is constrained to return not just any valid prefix, but the longest prefix of elements in the encounter order. Using an unordered stream source (such as `generate(Supplier)`) or removing the ordering constraint with `BaseStream.unordered()` may result in significant speedups of `takeWhile()` in parallel pipelines, if the semantics of your situation permit. If consistency with encounter order is required, and you are experiencing poor performance or memory utilization with `takeWhile()` in parallel pipelines, switching to sequential execution with `BaseStream.sequential()` may improve performance.

Implementation Requirements:

The default implementation obtains the `spliterator` of this stream, wraps that spliterator so as to support the semantics of this operation on traversal, and returns a new stream associated with the wrapped spliterator. The returned stream preserves the execution characteristics of this stream (namely parallel or sequential execution as per `BaseStream.isParallel()`) but the wrapped spliterator may choose to not support splitting. When the returned stream is closed, the close handlers for both the returned and this stream are invoked.

Parameters:

`predicate` - a non-interfering, stateless predicate to apply to elements to determine the longest prefix of elements.

Returns:

the new stream

Since:

9

- **dropWhile**

```
default Stream<T> dropWhile(Predicate<? super T> predicate)
```

Returns, if this stream is ordered, a stream consisting of the remaining elements of this stream after dropping the longest prefix of elements that match the given predicate. Otherwise returns, if this stream is unordered, a stream consisting of the remaining elements of this stream after dropping a subset of elements that match the given predicate.

If this stream is ordered then the longest prefix is a contiguous sequence of elements of this stream that match the given predicate. The first element of the sequence is the first element of this stream, and the element immediately following the last element of the sequence does not match the given predicate.

If this stream is unordered, and some (but not all) elements of this stream match the given predicate, then the behavior of this operation is nondeterministic; it is free to drop any subset of matching elements (which includes the empty set).

Independent of whether this stream is ordered or unordered if all elements of this stream match the given predicate then this operation drops all elements (the result is an empty stream), or if no elements of the stream match the given predicate then no elements are dropped (the result is the same as the input).

This is a stateful intermediate operation.

API Note:

While `dropWhile()` is generally a cheap operation on sequential stream pipelines, it can be quite expensive on ordered parallel pipelines, since the operation is constrained to return not just any valid prefix, but the longest prefix of elements in the encounter order. Using an unordered stream source (such as `generate(Supplier)`) or removing the ordering constraint with `BaseStream.unordered()` may result in significant speedups of `dropWhile()` in parallel pipelines, if the semantics of your situation permit. If consistency with encounter order is required, and you are experiencing poor performance or memory utilization with `dropWhile()` in parallel pipelines, switching to sequential execution with `BaseStream.sequential()` may improve performance.

Implementation Requirements:

The default implementation obtains the `spliterator` of this stream, wraps that spliterator so as to support the semantics of this operation on traversal, and returns a new stream associated with the wrapped spliterator. The returned stream preserves the execution characteristics of this stream (namely parallel or sequential execution as per `BaseStream.isParallel()`) but the wrapped spliterator may choose to not support splitting. When the returned stream is closed, the close handlers for both the returned and this stream are invoked.

Parameters:

`predicate` - a non-interfering, stateless predicate to apply to elements to determine the longest prefix of elements.

Returns:

the new stream

Since:

9

- **forEach**

```
void forEach(Consumer<? super T> action)
```

Performs an action for each element of this stream.

This is a terminal operation.

The behavior of this operation is explicitly nondeterministic. For parallel stream pipelines, this operation does not guarantee to respect the encounter order of the stream, as doing so would sacrifice the benefit of parallelism. For any given element, the action may be performed at whatever time and in whatever thread the library chooses. If the action accesses shared state, it is responsible for providing the required synchronization.

Parameters:

`action` - a non-interfering action to perform on the elements

- **forEachOrdered**

```
void forEachOrdered(Consumer<? super T> action)
```

Performs an action for each element of this stream, in the encounter order of the stream if the stream has a defined encounter order.

This is a terminal operation.

This operation processes the elements one at a time, in encounter order if one exists. Performing the action for one element happens-before performing the action for subsequent elements, but for any given element, the action may be performed in whatever thread the library chooses.

Parameters:

`action` - a non-interfering action to perform on the elements

See Also:

`forEach(Consumer)`

- **toArray**

```
Object[] toArray()
```

Returns an array containing the elements of this stream.

This is a terminal operation.

Returns:
an array containing the elements of this stream

- **toArray**

```
<A> A[] toArray(IntFunction<A[]> generator)
```

Returns an array containing the elements of this stream, using the provided `generator` function to allocate the returned array, as well as any additional arrays that might be required for a partitioned execution or for resizing.

This is a terminal operation.

API Note:
The generator function takes an integer, which is the size of the desired array, and produces an array of the desired size. This can be concisely expressed with an array constructor reference:

```
Person[] men = people.stream()
                     .filter(p -> p.getGender() == MALE)
                     .toArray(Person[]::new);
```

Type Parameters:
`A` - the element type of the resulting array
Parameters:
`generator` - a function which produces a new array of the desired type and the provided length
Returns:
an array containing the elements in this stream
Throws:
`ArrayStoreException` - if the runtime type of the array returned from the array generator is not a supertype of the runtime type of every element in this stream

- **reduce**

```
T reduce(T identity,
         BinaryOperator<T> accumulator)
```

Performs a reduction on the elements of this stream, using the provided identity value and an associative accumulation function, and returns the reduced value. This is equivalent to:

```
T result = identity;
for (T element : this stream)
    result = accumulator.apply(result, element)
return result;
```

but is not constrained to execute sequentially.

The `identity` value must be an identity for the accumulator function. This means that for all `t`, `accumulator.apply(identity, t)` is equal to `t`. The `accumulator` function must be an associative function.

This is a terminal operation.

API Note:

Sum, min, max, average, and string concatenation are all special cases of reduction. Summing a stream of numbers can be expressed as:

```
Integer sum = integers.reduce(0, (a, b) -> a+b);
```

or:

```
Integer sum = integers.reduce(0, Integer::sum);
```

While this may seem a more roundabout way to perform an aggregation compared to simply mutating a running total in a loop, reduction operations parallelize more gracefully, without needing additional synchronization and with greatly reduced risk of data races.

Parameters:

`identity` - the identity value for the accumulating function

`accumulator` - an associative, non-interfering, stateless function for combining two values

Returns:

the result of the reduction

- **reduce**

```
Optional<T> reduce(BinaryOperator<T> accumulator)
```

Performs a reduction on the elements of this stream, using an associative accumulation function, and returns an `Optional` describing the reduced value, if any. This is equivalent to:

```
boolean foundAny = false;
T result = null;
for (T element : this stream) {
    if (!foundAny) {
        foundAny = true;
        result = element;
    }
    else
        result = accumulator.apply(result, element);
}
return foundAny ? Optional.of(result) : Optional.empty();
```

but is not constrained to execute sequentially.

The `accumulator` function must be an associative function.

This is a terminal operation.

Parameters:

`accumulator` - an associative, non-interfering, stateless function for combining two values

Returns:

an `Optional` describing the result of the reduction

Throws:

`NullPointerException` - if the result of the reduction is null

See Also:

`reduce(Object, BinaryOperator)`, `min(Comparator)`, `max(Comparator)`

- **reduce**

```
<U> U reduce(U identity,
             BiFunction<U,? super T,U> accumulator,
             BinaryOperator<U> combiner)
```

Performs a reduction on the elements of this stream, using the provided identity, accumulation and combining functions. This is equivalent to:

```
U result = identity;
for (T element : this stream)
    result = accumulator.apply(result, element)
return result;
```

but is not constrained to execute sequentially.

The `identity` value must be an identity for the combiner function. This means that for all `u`, `combiner(identity, u)` is equal to `u`. Additionally, the `combiner` function must be compatible with the `accumulator` function; for all `u` and `t`, the following must hold:

```
combiner.apply(u, accumulator.apply(identity, t)) == accumulator.apply(u, t)
```

This is a terminal operation.

API Note:

Many reductions using this form can be represented more simply by an explicit combination of `map` and `reduce` operations. The `accumulator` function acts as a fused mapper and accumulator, which can sometimes be more efficient than separate mapping and reduction, such as when knowing the previously reduced value allows you to avoid some computation.

Type Parameters:

`U` - The type of the result

Parameters:

`identity` - the identity value for the combiner function

`accumulator` - an associative, non-interfering, stateless function for incorporating an additional element into a result

`combiner` - an associative, non-interfering, stateless function for combining two values, which must be compatible with the accumulator function

Returns:

the result of the reduction

See Also:

`reduce(BinaryOperator)`, `reduce(Object, BinaryOperator)`

- **collect**

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

Performs a mutable reduction operation on the elements of this stream. A mutable reduction is one in which the reduced value is a mutable result container, such as an `ArrayList`, and elements are incorporated by updating the state of the result rather than by replacing the result. This produces a result equivalent to:

```
R result = supplier.get();
for (T element : this stream)
    accumulator.accept(result, element);
return result;
```

Like `reduce(Object, BinaryOperator)`, `collect` operations can be parallelized without requiring additional synchronization.

This is a terminal operation.

API Note:

There are many existing classes in the JDK whose signatures are well-suited for use with method references as arguments to `collect()`. For example, the following will accumulate strings into an `ArrayList`:

```
List<String> asList = stringStream.collect(ArrayList::new, ArrayList::add,
                                           ArrayList::addAll);
```

The following will take a stream of strings and concatenates them into a single string:

```
String concat = stringStream.collect(StringBuilder::new, StringBuilder::append,
                                     StringBuilder::append)
                            .toString();
```

Type Parameters:

`R` - the type of the mutable result container

Parameters:

`supplier` - a function that creates a new mutable result container. For a parallel execution, this function may be called multiple times and must return a fresh value each time.

`accumulator` - an associative, non-interfering, stateless function that must fold an element into a result container.

`combiner` - an associative, non-interfering, stateless function that accepts two partial result containers and merges them, which must be compatible with the accumulator function. The combiner function must fold the elements from the second result container into the first result container.

Returns:

the result of the reduction

- **collect**

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

Performs a mutable reduction operation on the elements of this stream using a `Collector`. A `Collector` encapsulates the functions used as arguments to `collect(Supplier, BiConsumer, BiConsumer)`, allowing for reuse of collection strategies and composition of collect operations such as multiple-level grouping or partitioning.

If the stream is parallel, and the `Collector` is `concurrent`, and either the stream is unordered or the collector is `unordered`, then a concurrent reduction will be performed (see `Collector` for details on concurrent reduction.)

This is a terminal operation.

When executed in parallel, multiple intermediate results may be instantiated, populated, and merged so as to maintain isolation of mutable data structures. Therefore, even when executed in parallel with non-thread-safe data structures (such as `ArrayList`), no additional synchronization is needed for a parallel reduction.

API Note:

The following will accumulate strings into an ArrayList:

```
List<String> asList = stringStream.collect(Collectors.toList());
```

The following will classify `Person` objects by city:

```
Map<String, List<Person>> peopleByCity
    = personStream.collect(Collectors.groupingBy(Person::getCity));
```

The following will classify `Person` objects by state and city, cascading two `Collector`s together:

```
Map<String, Map<String, List<Person>>> peopleByStateAndCity
    = personStream.collect(Collectors.groupingBy(Person::getState,
                                                 Collectors.groupingBy(Person::getCity)));
```

Type Parameters:
`R` - the type of the result
`A` - the intermediate accumulation type of the `Collector`
Parameters:
`collector` - the `Collector` describing the reduction
Returns:
the result of the reduction
See Also:
`collect(Supplier, BiConsumer, BiConsumer)`, `Collectors`

- **min**

`Optional<T> min(Comparator<? super T> comparator)`

Returns the minimum element of this stream according to the provided `Comparator`. This is a special case of a reduction.

This is a terminal operation.

Parameters:
`comparator` - a non-interfering, stateless `Comparator` to compare elements of this stream
Returns:
an `Optional` describing the minimum element of this stream, or an empty `Optional` if the stream is empty
Throws:
`NullPointerException` - if the minimum element is null

- **max**

`Optional<T> max(Comparator<? super T> comparator)`

Returns the maximum element of this stream according to the provided `Comparator`. This is a special case of a reduction.

This is a terminal operation.

Parameters:
`comparator` - a non-interfering, stateless `Comparator` to compare elements of this stream
Returns:
an `Optional` describing the maximum element of this stream, or an empty `Optional` if the stream is empty
Throws:
`NullPointerException` - if the maximum element is null

- **count**

`long count()`

Returns the count of elements in this stream. This is a special case of a reduction and is equivalent to:

```
return mapToLong(e -> 1L).sum();
```

This is a terminal operation.

API Note:

An implementation may choose to not execute the stream pipeline (either sequentially or in parallel) if it is capable of computing the count directly from the stream source. In such cases no source elements will be traversed and no intermediate operations will be evaluated. Behavioral parameters with side-effects, which are strongly discouraged except for harmless cases such as debugging, may be affected. For example, consider the following stream:

```
List<String> l = Arrays.asList("A", "B", "C", "D");
long count = l.stream().peek(System.out::println).count();
```

The number of elements covered by the stream source, a `List`, is known and the intermediate operation, `peek`, does not inject into or remove elements from the stream (as may be the case for `flatMap` or `filter` operations). Thus the count is the size of the `List` and there is no need to execute the pipeline and, as a side-effect, print out the list elements.

Returns:

the count of elements in this stream

- **anyMatch**

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

Returns whether any elements of this stream match the provided predicate. May not evaluate the predicate on all elements if not necessary for determining the result. If the stream is empty then `false` is returned and the predicate is not evaluated.

This is a short-circuiting terminal operation.

API Note:

This method evaluates the existential quantification of the predicate over the elements of the stream (for some x P(x)).

Parameters:

`predicate` - a non-interfering, stateless predicate to apply to elements of this stream

Returns:

`true` if any elements of the stream match the provided predicate, otherwise `false`

- **allMatch**

```
boolean allMatch(Predicate<? super T> predicate)
```

Returns whether all elements of this stream match the provided predicate. May not evaluate the predicate on all elements if not necessary for determining the result. If the stream is empty then `true` is returned and the predicate is not evaluated.

This is a short-circuiting terminal operation.

API Note:

This method evaluates the universal quantification of the predicate over the elements of the stream (for all x P(x)). If the stream is empty, the quantification is said to be vacuously satisfied and is always `true` (regardless of P(x)).

Parameters:

`predicate` - a non-interfering, stateless predicate to apply to elements of this stream

Returns:

`true` if either all elements of the stream match the provided predicate or the stream is empty, otherwise `false`

- **noneMatch**

`boolean noneMatch(Predicate<? super T> predicate)`

Returns whether no elements of this stream match the provided predicate. May not evaluate the predicate on all elements if not necessary for determining the result. If the stream is empty then `true` is returned and the predicate is not evaluated.

This is a short-circuiting terminal operation.

API Note:

This method evaluates the universal quantification of the negated predicate over the elements of the stream (for all x ~P(x)). If the stream is empty, the quantification is said to be vacuously satisfied and is always `true`, regardless of P(x).

Parameters:

`predicate` - a non-interfering, stateless predicate to apply to elements of this stream

Returns:

`true` if either no elements of the stream match the provided predicate or the stream is empty, otherwise `false`

- **findFirst**

`Optional<T> findFirst()`

Returns an `Optional` describing the first element of this stream, or an empty `Optional` if the stream is empty. If the stream has no encounter order, then any element may be returned.

This is a short-circuiting terminal operation.

Returns:

an `Optional` describing the first element of this stream, or an empty `Optional` if the stream is empty

Throws:

`NullPointerException` - if the element selected is null

- **findAny**

`Optional<T> findAny()`

Returns an `Optional` describing some element of the stream, or an empty `Optional` if the stream is empty.

This is a short-circuiting terminal operation.

The behavior of this operation is explicitly nondeterministic; it is free to select any element in the stream. This is to allow for maximal performance in parallel operations; the cost is that multiple invocations on the same source may not return the same result. (If a stable result is desired, use `findFirst()` instead.)

Returns:

an `Optional` describing some element of this stream, or an empty `Optional` if the stream is empty

Throws:

`NullPointerException` - if the element selected is null

See Also:

`findFirst()`

- **builder**

`static <T> Stream.Builder<T> builder()`

Returns a builder for a `Stream`.

Type Parameters:

`T` - type of elements
Returns:
a stream builder

- **empty**

```
static <T> Stream<T> empty()
```

Returns an empty sequential `Stream`.

Type Parameters:
`T` - the type of stream elements
Returns:
an empty sequential stream

- **of**

```
static <T> Stream<T> of(T t)
```

Returns a sequential `Stream` containing a single element.

Type Parameters:
`T` - the type of stream elements
Parameters:
`t` - the single element
Returns:
a singleton sequential stream

- **ofNullable**

```
static <T> Stream<T> ofNullable(T t)
```

Returns a sequential `Stream` containing a single element, if non-null, otherwise returns an empty `Stream`.

Type Parameters:
`T` - the type of stream elements
Parameters:
`t` - the single element
Returns:
a stream with a single element if the specified element is non-null, otherwise an empty stream
Since:
9

- **of**

```
@SafeVarargs
static <T> Stream<T> of(T... values)
```

Returns a sequential ordered stream whose elements are the specified values.

Type Parameters:
`T` - the type of stream elements
Parameters:
`values` - the elements of the new stream
Returns:
the new stream

- **iterate**

```
static <T> Stream<T> iterate(T seed,
                             UnaryOperator<T> f)
```

Returns an infinite sequential ordered `Stream` produced by iterative application of a function `f` to an initial element `seed`, producing a `Stream` consisting of `seed`, `f(seed)`, `f(f(seed))`, etc.

The first element (position `0`) in the `Stream` will be the provided `seed`. For `n > 0`, the element at position `n`, will be the result of applying the function `f` to the element at position `n - 1`.

The action of applying `f` for one element happens-before the action of applying `f` for subsequent elements. For any given element the action may be performed in whatever thread the library chooses.

Type Parameters:
`T` - the type of stream elements
Parameters:
`seed` - the initial element
`f` - a function to be applied to the previous element to produce a new element
Returns:
a new sequential `Stream`

- **iterate**

```
static <T> Stream<T> iterate(T seed,
                             Predicate<? super T> hasNext,
                             UnaryOperator<T> next)
```

Returns a sequential ordered `Stream` produced by iterative application of the given `next` function to an initial element, conditioned on satisfying the given `hasNext` predicate. The stream terminates as soon as the `hasNext` predicate returns false.

`Stream.iterate` should produce the same sequence of elements as produced by the corresponding for-loop:

```
for (T index=seed; hasNext.test(index); index = next.apply(index)) {
    ...
}
```

The resulting sequence may be empty if the `hasNext` predicate does not hold on the seed value. Otherwise the first element will be the supplied `seed` value, the next element (if present) will be the result of applying the `next` function to the `seed` value, and so on iteratively until the `hasNext` predicate indicates that the stream should terminate.

The action of applying the `hasNext` predicate to an element happens-before the action of applying the `next` function to that element. The action of applying the `next` function for one element happens-before the action of applying the `hasNext` predicate for subsequent elements. For any given element an action may be performed in whatever thread the library chooses.

Type Parameters:
`T` - the type of stream elements
Parameters:
`seed` - the initial element
`hasNext` - a predicate to apply to elements to determine when the stream must terminate.
`next` - a function to be applied to the previous element to produce a new element
Returns:
a new sequential `Stream`
Since:
9

- **generate**

```
static <T> Stream<T> generate(Supplier<? extends T> s)
```

Returns an infinite sequential unordered stream where each element is generated by the provided `Supplier`. This is suitable for generating constant streams, streams of random elements, etc.

Type Parameters:
`T` - the type of stream elements
Parameters:
`s` - the `Supplier` of generated elements
Returns:
a new infinite sequential unordered `Stream`

- **concat**

```
static <T> Stream<T> concat(Stream<? extends T> a,
                            Stream<? extends T> b)
```

Creates a lazily concatenated stream whose elements are all the elements of the first stream followed by all the elements of the second stream. The resulting stream is ordered if both of the input streams are ordered, and parallel if either of the input streams is parallel. When the resulting stream is closed, the close handlers for both input streams are invoked.

Implementation Note:
Use caution when constructing streams from repeated concatenation. Accessing an element of a deeply concatenated stream can result in deep call chains, or even `StackOverflowError`.

Subsequent changes to the sequential/parallel execution mode of the returned stream are not guaranteed to be propagated to the input streams.

Type Parameters:
`T` - The type of stream elements
Parameters:
`a` - the first stream
`b` - the second stream
Returns:
the concatenation of the two input streams