Module java.base

Package java.util.stream

The stream API defines several stream interfaces, which are packaged in java.util.stream and contained in the java.base module.

Describe Stream API?

Answer: - The key aspect of the stream API is its ability to perform very sophisticated operations that search, filter, map, or otherwise manipulate data.

For example, using the stream API, you can construct sequences of actions that resemble, in concept, the type of database queries for which you might use SQL.

Furthermore, in many cases, such actions can be performed in parallel, thus providing a high level of efficiency, especially when large data sets are involved. Put simply, the stream API provides a powerful means of handling data in an efficient, yet easy to use way.

What is stream in Stream API?

Answer: - A stream is a conduit (channel/tube) for data. Thus, a stream represents a sequence of objects. A stream operates on a data source, such as an array or a collection. A stream, itself, never provides storage for the data. It simply moves data, possibly filtering, sorting, or otherwise operating on that data in the process.

As a general rule, however, a stream operation by itself does not modify the data source. For example, sorting a stream does not change the order of the source. Rather, sorting a stream results in the creation of a new stream that produces the sorted result.

Stream operations and pipelines

Stream operations are divided into intermediate and terminal operations, and are combined to form stream pipelines.

A stream pipeline consists of a source (such as a Collection, an array, a generator function, or an I/O channel); followed by zero or more intermediate operations such as Stream.filter or Stream.map; and a terminal operation such as Stream.forEach or Stream.reduce.

List down the Interfaces of Stream API?

Answer: - Following are the list of Interfaces presents into Stream API: -

1. BaseStream<T, S extends BaseStream<T, S>>
2. Collector<T, A, R>
3. DoubleStream
4. IntStream
5. LongStream
6. Stream<T>

List down the Classes of stream API?

Answer: - Following are the list of Classes presents into stream API:-

1. Collectors
2. StreamSupport

Stream operates on object references; it can't operate directly on primitive types. To handle primitive type streams, the stream API defines the following interfaces:

- DoubleStream
- IntStream
- LongStream

How a Streams differ from collections?

Answer: - Stream differ form collection in following ways : -

1. **No storage.** A stream is not a data structure that stores elements; instead, it conveys elements from a source such as a data structure, an array, a generator function, or an I/O channel, through a pipeline of computational operations.
2. **Functional in nature.** An operation on a stream produces a result, but does not modify its source. For example, filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection.
3. **Laziness-seeking.** Many stream operations, such as filtering, mapping, or duplicate removal, can be implemented lazily, exposing opportunities for optimization. For example, "find the first String with three consecutive vowels" need not examine all the input strings. Stream operations are divided into intermediate (Stream-producing) operations and terminal (value- or side-effect-producing) operations. Intermediate operations are always lazy.
4. **Possibly unbounded.** While collections have a finite size, streams need not. Short-circuiting operations such as limit(n) or findFirst() can allow computations on infinite streams to complete in finite time.
5. **Consumable.** The elements of a stream are only visited once during the life of a stream. Like an Iterator, a new stream must be generated to revisit the same elements of the source.

How Streams can be obtained?

Answer: - Streams can be obtained in a number of ways. Some examples include:

1. From a **Collection** via the **stream()** and **parallelStream()** methods;
2. From an **Array** via **Arrays.stream(Object[])**;
3. From **static factory methods** on the stream classes, such as **Stream.of(Object[]), IntStream.range(int, int) or Stream.iterate(Object, UnaryOperator);**
4. The lines of a file can be obtained from **BufferedReader.lines();**
5. Streams of file paths can be obtained from methods in Files;
6. Streams of random numbers can be obtained from **Random.ints();**
7. Numerous other stream-bearing methods in the JDK, including **BitSet.stream(), Pattern.splitAsStream(java.lang.CharSequence), and JarFile.stream().**

What is a Terminal Operation?

Answer: - A Terminal operation **consumes the stream**. It is used to produce a result, such as finding the minimum value in the stream, or to execute some action, as is the case with the forEach( ) method. Once a stream has been consumed, it cannot be reused.

What is Intermediate Operation?

Answer: - Intermediate operations produce another stream. Thus, intermediate operations can be used to create a pipeline that performs a sequence of actions.

What is Short-Circuiting Terminal operation?

Answer: - Some operations are deemed short-circuiting operations. An intermediate operation is short-circuiting if, when presented with infinite input, it may produce a finite stream as a result.

A terminal operation is short-circuiting if, when presented with infinite input, it may terminate in finite time. Having a short-circuiting operation in the pipeline is a necessary, but not sufficient, condition for the processing of an infinite stream to terminate normally in finite time.

What is difference between Terminal Operation and Intermediate Operation?

Answer: -

Terminal Operation	Intermediate Operation
Terminal operations, such as Stream.forEach or IntStream.sum, may traverse the stream to produce a result or a side-effect. After the terminal operation is performed, the stream pipeline is considered consumed, and can no	Intermediate operations **return a new stream.**

longer be used; if you need to traverse the same data source again, you must return to the data source to get a new stream. In almost all cases, terminal operations are eager, completing their traversal of the data source and processing of the pipeline before returning. Only the terminal operations iterator() and spliterator() are not; these are provided as an "escape hatch" to enable arbitrary client-controlled pipeline traversals in the event that the existing operations are not sufficient to the task.	
	Intermediate operations can be used to create a pipeline that performs a sequence of actions.
	Intermediate operations do not take place immediately. Instead, the specified action is performed when a terminal operation is executed on the new stream created by an intermediate operation. **This mechanism is referred to as lazy behavior**, and the intermediate operations are referred to as lazy. The use of lazy behavior enables the stream API to perform more efficiently.
	some intermediate operations are **stateless** and some are **stateful**. **In a stateless operation, each element is processed independently of the others.**

	In a stateful operation, the processing of an element may depend on aspects of the other elements. For example, **sorting** is a **stateful operation** because an element's order depends on the values of the other elements. Thus, the sorted( ) method is stateful. However, **filtering** elements based on a stateless predicate is **stateless** because each element is handled individually. Thus, filter( ) can (and should be) stateless. The difference between stateless and stateful operations is especially important when parallel processing of a stream is desired because a stateful operation may require more than one pass to complete.

What is Stateless operation and Stateful operation?

Answer: -

- In a **stateless operation**, each element is processed independently of the others. **Filtering** elements based on a stateless predicate is stateless because **each element is handled individually**. Thus, filter () can (and should be) stateless.
- In a **stateful operation**, the processing of an element **may depend on aspects of the other elements**. For example, sorting is a stateful operation because an element's order depends on the values of the other elements. Thus, the sorted () method is stateful.
- **Stateless operations**, such as filter and map, **retain no state from previously seen element when processing a new element** -- **each element can be processed independently of operations on other elements.**

- **Stateful operations**, such as distinct and sorted, **may incorporate state from previously seen elements when processing new elements.**
- **Stateful operations** may need to process the entire input before producing a result. For example, one cannot produce any results from sorting a stream until one has seen all elements of the stream. As a result, under parallel computation, some pipelines containing stateful intermediate operations may require multiple passes on the data or may need to buffer significant data.
- Pipelines containing exclusively **stateless intermediate operations** can be processed in a single pass, whether sequential or parallel, with minimal data buffering.

Note: - The difference between stateless and stateful operations is especially important when parallel processing of a stream is desired because a stateful operation may require more than one pass to complete.

What are Reduction Operations?

Answer: - Reduction operations **reduce a stream to a single value**. For example, min (), max () and count () support reduction operation. The stream API refers to these as special case reductions because they perform a specific function.

However, the stream API generalizes this concept by providing the reduce () method. By using reduce (), you can return a value from a stream based on any arbitrary criteria. By definition, all reduction operations are **terminal operations.**

What is an accumulator?

Answer: - Accumulator is a function that operates on two values and produces a result. It is important to understand that the accumulator operation must satisfy three constraints. It must be

- Stateless
- Non-interfering
- Associative

As explained earlier, stateless means that the operation does not rely on any state information. Thus, each element is processed independently. Non-interfering means

that the data source is not modified by the operation. Finally, the operation must be associative. Here, the term associative is used in its normal, arithmetic sense, which means that, given an associative operator used in a sequence of operations, it does not matter which pair of operands are processed first. For example, (10 * 2) * 7 yields the same result as 10 * (2 * 7)

What are a Parallel Streams?

Answer: - The parallel execution of code via multicore processors can result in a substantial increase in performance. Because of this, parallel programming has become an important part of the modern programmer's job. However, parallel programming can be complex and error-prone. One of the benefits that the stream library offers is the ability to easily—and reliably—parallel process certain operations.

Parallel processing of a stream is quite simple to request: just use a parallel stream.

Once a parallel stream has been obtained, operations on the stream can occur in parallel, assuming that parallelism is supported by the environment. **As a general rule, any operation applied to a parallel stream must be stateless. It should also be non-interfering and associative.** This ensures that the results obtained by executing operations on a parallel stream are the same as those obtained from executing the same operations on a sequential stream.

How to obtain Parallel Streams?

Answer: - One way to obtain a parallel stream is to use the **parallelStream () method defined by Collection**. Another way to obtain a parallel stream is to **call the parallel () method on a sequential stream**. The **parallel ()** method is **defined** by **BaseStream.**

Mapping

Often it is useful to map the elements of one stream to another.

For example, a stream that contains a database of name, telephone, and e-mail address information might map only the name and e-mail address portions to another stream.

As another example, you might want to apply some transformation to the elements in a stream. To do this, you could map the transformed elements to a new stream.

Because mapping operations are quite common, the stream API provides built-in support for them.

The most general mapping method is **map ()**.

Collecting

To obtain a collection from a stream. To perform such an action, the stream API provides the **collect** () method.

BaseStream

Module java.base

Package java.util.stream

Since: 1.8

Interface BaseStream<T, S extends BaseStream<T, S>>

---

public interface BaseStream<T, S extends BaseStream<T, S>> extends AutoCloseable

Here, **T** specifies the **type of the elements** in the stream, and **S** specifies the **type of stream** that extends BaseStream.

BaseStream, which defines the basic functionality available in all streams.

BaseStream extends the AutoCloseable interface; thus, a stream can be managed in a try-with-resources statement. In general, however, only those streams whose data source requires closing (such as those connected to a file) will need to be closed. In most cases, such as those in which the data source is a collection, there is no need to close the stream.

Methods of BaseStream Interface: -

1. **void close(): -** Closes the invoking stream, Calling any resistered close handlers.
2. **boolean isParallel(): -** Returns true if the invoking stream is parallel. Returns false if the stream is sequential.
3. **Iterator<T> iterator()**: - Obtains an iterator to the stream and returns a reference to it. This is a **terminal operation.**
4. **Spliterator<T> spliterator()**: - Obtains a spliterator to the stream and returns a reference to it. This is a **terminal operation**.
5. **S parallel()**: - Returns a parallel stream based on the invoking stream. if the invoking stream is already parallel, then that stream is returned. This is an **intermediate operation.**
6. **S sequential()**: - Returns a sequential stream based on the invoking stream. If the invoking stream is already sequential, then that stream is returned . This is an **intermediate operation**.
7. **S unordered()**: - Returns an unordered stream based on the invoking stream. If the invoking stream is already unordered, then that stream is returned .This is an **intermediate operation.**
8. **S onClose (Runnable closeHandler)**: - Returns a new stream with the close handler specified by handler. This handler will be called when the stream is closed.This is an **intermediate operation**.

Stream Interface

Module java.base

Package java.util.stream

Since: 1.8

Interface Stream<T>

---

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

The Stream interface extends BaseStream interface for supporting sequential and parallel aggregate operations.

Methods of Stream Interface

void forEach (Consumer<? super T> action)

Performs an action for each element of this stream. This is a **terminal operation.**

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.**

boolean anyMatch (Predicate<? super T> predicate)

Returns **True** if any elements of the stream match the provided predicate, otherwise **False.**

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.**

boolean allMatch (Predicate<? super T> predicate)

Returns **True** if either all elements of the stream match the provided predicate or the stream is empty, otherwise **False**.

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.**

boolean noneMatch (Predicate<? super T> predicate)

Returns **True** if either no elements of the stream match the provided predicate or the stream is empty, otherwise **False**.

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**.

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**.

Throws: **NullPointerException** - if the element selected is null

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.**

Throws: NullPointerException - if the element selected is null

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.**

Throws: NullPointerException - if the maximum element is null

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.**

Throws: NullPointerException - if the minimum element is null

long count()

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

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

Returns a builder for a Stream.

static <T> Stream<T> empty()

Returns an empty sequential Stream.

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

Returns a sequential Stream containing a single element, if non-null, otherwise returns an empty Stream. **Since: 9**

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

Returns a sequential Stream containing a single element.

@SafeVarargs

static <T> Stream<T> of (T... values)

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

Returns: the new stream

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.**

Returns: the new stream

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

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**.

Returns: the new 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.

This is an intermediate operation. Returns: the new stream

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.

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**.

Returns: the new stream

Throws: IllegalArgumentException - if maxSize is negative

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.**

Returns: the new stream

Throws: IllegalArgumentException - if n is negative

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

Returns a builder for a Stream.

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**.

Returns: the new stream

Since: 9

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 s**tateful intermediate operation.**

Returns: the new stream

Since: 9

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 a **terminal operation**.

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 a **terminal operation.**

Throws: NullPointerException - if the result of the reduction is null

<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 a **terminal operation.**

<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 is a **terminal operation**.

<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.

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.

This method operates on the two input streams and binds each stream to its source. As a result subsequent modifications to an input stream source may not be reflected in the concatenated stream result.

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**.

<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.**

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.**

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.**

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.**

<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.**

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.**

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.**

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.**

default <R> Stream<R> mapMulti (BiConsumer<? super T, ? super Consumer<R>> mapper)

Returns a stream consisting of the results of replacing each element of this stream with multiple elements, specifically zero or more elements. Replacement is performed by applying the provided mapping function to each element in conjunction with a consumer argument that accepts replacement elements. The mapping function calls the consumer zero or more times to provide the replacement elements. This is an **intermediate operation.**

default IntStream mapMultiToInt (BiConsumer<? super T, ? super IntConsumer> mapper)

Returns an IntStream consisting of the results of replacing each element of this stream with multiple elements, specifically zero or more elements. Replacement is performed by applying the provided mapping function to each element in conjunction with a consumer argument that accepts replacement elements. The mapping function calls the consumer zero or more times to provide the replacement elements. This is an **intermediate operation**. If the consumer argument is used outside the scope of its application to the mapping function, the results are undefined.

default LongStream mapMultiToLong (BiConsumer<? super T, ? super LongConsumer> mapper)

Returns a LongStream consisting of the results of replacing each element of this stream with multiple elements, specifically zero or more elements. Replacement is performed by applying the provided mapping function to each element in conjunction with a consumer argument that accepts replacement elements. The mapping function calls the consumer zero or more times to provide the replacement elements. This is an **intermediate operation**. If the consumer argument is used

outside the scope of its application to the mapping function, the results are undefined.

default DoubleStream mapMultiToDouble (BiConsumer<? super T, ? super DoubleConsumer> mapper)

Returns a DoubleStream consisting of the results of replacing each element of this stream with multiple elements, specifically zero or more elements. Replacement is performed by applying the provided mapping function to each element in conjunction with a consumer argument that accepts replacement elements. The mapping function calls the consumer zero or more times to provide the replacement elements. **This is an intermediate operation**. If the consumer argument is used outside the scope of its application to the mapping function, the results are undefined.

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.

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.

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.

Object[] toArray()

Returns an array containing the elements of this stream. This is a **terminal operation**.

<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.**

default List<T> toList()

Accumulates the elements of this stream into a List. The elements in the list will be in this stream's encounter order, if one exists. The returned List is unmodifiable; calls to any mutator method will always cause UnsupportedOperationException to be thrown. There are no guarantees on the implementation type or serializability of the returned List.

The returned instance may be value-based. Callers should make no assumptions about the identity of the returned instances. Identity-sensitive operations on these instances (reference equality (==), identity hash code, and synchronization) are unreliable and should be avoided. **This is a terminal operation**.

Module java.base

Package java.util.stream

Since: 1.8

Interface DoubleStream

---

public interface DoubleStream extends BaseStream<Double, DoubleStream>

A sequence of primitive double-valued elements supporting sequential and parallel aggregate operations. This is the double primitive specialization of Stream.

Module java.base

Package java.util.stream

Since: 1.8

Interface LongStream

---

public interface LongStream extends BaseStream<Long, LongStream>

A sequence of primitive long-valued elements supporting sequential and parallel aggregate operations. This is the long primitive specialization of Stream.

Module java.base

Package java.util.stream

Since: 1.8

Interface IntStream

---

public interface IntStream extends BaseStream<Integer, IntStream>

A sequence of primitive int-valued elements supporting sequential and parallel aggregate operations. This is the int primitive specialization of Stream.