java.util.stream (Java Platform SE 8)

Package java.util.stream Description

Classes to support functional-style operations on streams of elements, such as map-reduce transformations on collections. For example:

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

Here we use `widgets`, a `Collection<Widget>`, as a source for a stream, and then perform a filter-map-reduce on the stream to obtain the sum of the weights of the red widgets. (Summation is an example of a reduction operation.)

The key abstraction introduced in this package is *stream*. The classes `Stream`, `IntStream`, `LongStream`, and `DoubleStream` are streams over objects and the primitive `int`, `long` and `double` types. Streams differ from collections in several ways:

- 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.
- 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.
- 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.
- 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.
- 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.

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

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

Additional stream sources can be provided by third-party libraries using these techniques.

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`.

Intermediate operations return a new stream. They are always *lazy*; executing an intermediate operation such as `filter()` does not actually perform any filtering, but instead creates a new stream that, when traversed, contains the elements of the initial stream that match the given predicate. Traversal of the pipeline source does not begin until the terminal operation of the pipeline is executed.

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 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.

Processing streams lazily allows for significant efficiencies; in a pipeline such as the filter-map-sum example above, filtering, mapping, and summing can be fused into a single pass on the data, with minimal intermediate state. Laziness also allows avoiding examining all the data when it is not necessary; for operations such as "find the first string longer than 1000 characters", it is only necessary to examine just enough strings to find one that has the desired characteristics without examining all of the strings available from the source. (This behavior becomes even more important when the input stream is infinite and not merely large.)

Intermediate operations are further divided into *stateless* and *stateful* operations. 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.

Further, 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.

Parallelism

Processing elements with an explicit `for-`loop is inherently serial. Streams facilitate parallel execution by reframing the computation as a pipeline of aggregate operations, rather than as imperative operations on each individual element. All streams operations can execute either in serial or in parallel. The stream implementations in the JDK create serial streams unless parallelism is explicitly requested. For example, `Collection` has methods `Collection.stream()` and `Collection.parallelStream()`, which produce sequential and parallel streams respectively; other stream-bearing methods such as `IntStream.range(int, int)` produce sequential streams but these streams can be efficiently parallelized by invoking their `BaseStream.parallel()` method. To execute the prior "sum of weights of widgets" query in parallel, we would do:

```
int sumOfWeights = widgets.parallelStream()
                          .filter(b -> b.getColor() == RED)
                          .mapToInt(b -> b.getWeight())
                          .sum();
```

The only difference between the serial and parallel versions of this example is the creation of the initial stream, using "`parallelStream()`" instead of "`stream()`". When the terminal operation is initiated, the

stream pipeline is executed sequentially or in parallel depending on the orientation of the stream on which it is invoked. Whether a stream will execute in serial or parallel can be determined with the `isParallel()` method, and the orientation of a stream can be modified with the `BaseStream.sequential()` and `BaseStream.parallel()` operations. When the terminal operation is initiated, the stream pipeline is executed sequentially or in parallel depending on the mode of the stream on which it is invoked.

Except for operations identified as explicitly nondeterministic, such as `findAny()`, whether a stream executes sequentially or in parallel should not change the result of the computation.

Most stream operations accept parameters that describe user-specified behavior, which are often lambda expressions. To preserve correct behavior, these *behavioral parameters* must be *non-interfering*, and in most cases must be *stateless*. Such parameters are always instances of a functional interface such as `Function`, and are often lambda expressions or method references.

Non-interference

Streams enable you to execute possibly-parallel aggregate operations over a variety of data sources, including even non-thread-safe collections such as `ArrayList`. This is possible only if we can prevent *interference* with the data source during the execution of a stream pipeline. Except for the escape-hatch operations `iterator()` and `spliterator()`, execution begins when the terminal operation is invoked, and ends when the terminal operation completes. For most data sources, preventing interference means ensuring that the data source is *not modified at all* during the execution of the stream pipeline. The notable exception to this are streams whose sources are concurrent collections, which are specifically designed to handle concurrent modification. Concurrent stream sources are those whose `Spliterator` reports the `CONCURRENT` characteristic.

Accordingly, behavioral parameters in stream pipelines whose source might not be concurrent should never modify the stream's data source. A behavioral parameter is said to *interfere* with a non-concurrent data source if it modifies, or causes to be modified, the stream's data source. The need for non-interference applies to all pipelines, not just parallel ones. Unless the stream source is concurrent, modifying a stream's data source during execution of a stream pipeline can cause exceptions, incorrect answers, or nonconformant behavior. For well-behaved stream sources, the source can be modified before the terminal operation commences and those modifications will be reflected in the covered elements. For example, consider the following code:

```
List<String> l = new ArrayList(Arrays.asList("one", "two"));
Stream<String> sl = l.stream();
l.add("three");
String s = sl.collect(joining(" "));
```

First a list is created consisting of two strings: "one"; and "two". Then a stream is created from that list. Next the list is modified by adding a third string: "three". Finally the elements of the stream are collected and joined together. Since the list was modified before the terminal `collect` operation commenced the result will be a string of "one two three". All the streams returned from JDK collections, and most other JDK classes, are well-behaved in this manner; for streams generated by other libraries, see Low-level stream construction for requirements for building well-behaved streams.

Stateless behaviors

Stream pipeline results may be nondeterministic or incorrect if the behavioral parameters to the stream operations are *stateful*. A stateful lambda (or other object implementing the appropriate functional interface) is one whose result depends on any state which might change during the execution of the stream pipeline. An example of a stateful lambda is the parameter to `map()` in:

```
    Set<Integer> seen = Collections.synchronizedSet(new HashSet<>());
    stream.parallel().map(e -> { if (seen.add(e)) return 0; else return e;
}).
```

Here, if the mapping operation is performed in parallel, the results for the same input could vary from run to run, due to thread scheduling differences, whereas, with a stateless lambda expression the results would always be the same.

Note also that attempting to access mutable state from behavioral parameters presents you with a bad choice with respect to safety and performance; if you do not synchronize access to that state, you have a data race and therefore your code is broken, but if you do synchronize access to that state, you risk having contention undermine the parallelism you are seeking to benefit from. The best approach is to avoid stateful behavioral parameters to stream operations entirely; there is usually a way to restructure the stream pipeline to avoid statefulness.

Side-effects

Side-effects in behavioral parameters to stream operations are, in general, discouraged, as they can often lead to unwitting violations of the statelessness requirement, as well as other thread-safety hazards.

If the behavioral parameters do have side-effects, unless explicitly stated, there are no guarantees as to the *visibility* of those side-effects to other threads, nor are there any guarantees that different operations on the "same" element within the same stream pipeline are executed in the same thread. Further, the ordering of those effects may be surprising. Even when a pipeline is constrained to produce a *result* that is consistent with the encounter order of the stream source (for example, `IntStream.range(0,5).parallel().map(x -> x*2).toArray()` must produce `[0, 2, 4, 6, 8]`), no guarantees are made as to the order in which the mapper function is applied to individual elements, or in what thread any behavioral parameter is executed for a given element.

Many computations where one might be tempted to use side effects can be more safely and efficiently expressed without side-effects, such as using reduction instead of mutable accumulators. However, side-effects such as using `println()` for debugging purposes are usually harmless. A small number of stream operations, such as `forEach()` and `peek()`, can operate only via side-effects; these should be used with care.

As an example of how to transform a stream pipeline that inappropriately uses side-effects to one that does not, the following code searches a stream of strings for those matching a given regular expression, and puts the matches in a list.

```
ArrayList<String> results = new ArrayList<>();
stream.filter(s -> pattern.matcher(s).matches())
      .forEach(s -> results.add(s));  // Unnecessary use of side-effects!
```

This code unnecessarily uses side-effects. If executed in parallel, the non-thread-safety of `ArrayList` would cause incorrect results, and adding needed synchronization would cause contention, undermining the benefit of parallelism. Furthermore, using side-effects here is completely unnecessary; the `forEach()` can simply be replaced with a reduction operation that is safer, more efficient, and more amenable to parallelization:

```
List<String>results =
    stream.filter(s -> pattern.matcher(s).matches())
          .collect(Collectors.toList());  // No side-effects!
```

Ordering

Streams may or may not have a defined *encounter order*. Whether or not a stream has an encounter order depends on the source and the intermediate operations. Certain stream sources (such as `List` or arrays) are intrinsically ordered, whereas others (such as `HashSet`) are not. Some intermediate operations, such as `sorted()`, may impose an encounter order on an otherwise unordered stream, and others may render an ordered stream unordered, such as `BaseStream.unordered()`. Further, some terminal operations may ignore encounter order, such as `forEach()`.

If a stream is ordered, most operations are constrained to operate on the elements in their encounter order; if the source of a stream is a `List` containing `[1, 2, 3]`, then the result of executing `map(x -> x*2)` must be `[2, 4, 6]`. However, if the source has no defined encounter order, then any permutation of the values `[2, 4, 6]` would be a valid result.

For sequential streams, the presence or absence of an encounter order does not affect performance, only determinism. If a stream is ordered, repeated execution of identical stream pipelines on an identical source will produce an identical result; if it is not ordered, repeated execution might produce different results.

For parallel streams, relaxing the ordering constraint can sometimes enable more efficient execution. Certain aggregate operations, such as filtering duplicates (`distinct()`) or grouped reductions (`Collectors.groupingBy()`) can be implemented more efficiently if ordering of elements is not relevant. Similarly, operations that are intrinsically tied to encounter order, such as `limit()`, may require buffering to ensure proper ordering, undermining the benefit of parallelism. In cases where the stream has an encounter order, but the user does not particularly *care* about that encounter order, explicitly de-ordering the stream with `unordered()` may improve parallel performance for some stateful or terminal operations. However, most stream pipelines, such as the "sum of weight of blocks" example above, still parallelize efficiently even under ordering constraints.

Reduction operations

A *reduction* operation (also called a *fold*) takes a sequence of input elements and combines them into a single summary result by repeated application of a combining operation, such as finding the sum or maximum of a set of numbers, or accumulating elements into a list. The streams classes have multiple forms of general reduction operations, called `reduce()` and `collect()`, as well as multiple specialized reduction forms such as `sum()`, `max()`, or `count()`.

Of course, such operations can be readily implemented as simple sequential loops, as in:

```
int sum = 0;
for (int x : numbers) {
   sum += x;
}
```

However, there are good reasons to prefer a reduce operation over a mutative accumulation such as the above. Not only is a reduction "more abstract" -- it operates on the stream as a whole rather than individual elements -- but a properly constructed reduce operation is inherently parallelizable, so long as the function(s) used to process the elements are associative and stateless. For example, given a stream of numbers for which we want to find the sum, we can write:

```
int sum = numbers.stream().reduce(0, (x,y) -> x+y);
```

or:

```
int sum = numbers.stream().reduce(0, Integer::sum);
```

These reduction operations can run safely in parallel with almost no modification:

```
int sum = numbers.parallelStream().reduce(0, Integer::sum);
```

Reduction parallellizes well because the implementation can operate on subsets of the data in parallel, and then combine the intermediate results to get the final correct answer. (Even if the language had a "parallel for-each" construct, the mutative accumulation approach would still required the developer to provide thread-safe updates to the shared accumulating variable `sum`, and the required synchronization would then likely eliminate any performance gain from parallelism.) Using `reduce()` instead removes all of the burden of parallelizing the reduction operation, and the library can provide an efficient parallel implementation with no additional synchronization required.

The "widgets" examples shown earlier shows how reduction combines with other operations to replace for loops with bulk operations. If `widgets` is a collection of `Widget` objects, which have a `getWeight` method, we can find the heaviest widget with:

```
OptionalInt heaviest = widgets.parallelStream()
                              .mapToInt(Widget::getWeight)
                              .max();
```

In its more general form, a `reduce` operation on elements of type `<T>` yielding a result of type `<U>` requires three parameters:

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

Here, the *identity* element is both an initial seed value for the reduction and a default result if there are no input elements. The *accumulator* function takes a partial result and the next element, and produces a new partial result. The *combiner* function combines two partial results to produce a new partial result.

(The combiner is necessary in parallel reductions, where the input is partitioned, a partial accumulation computed for each partition, and then the partial results are combined to produce a final result.)
More formally, the `identity` value must be an *identity* for the combiner function. This means that for all `u`, `combiner.apply(identity, u)` is equal to `u`. Additionally, the `combiner` function must be associative and must be compatible with the `accumulator` function: for all `u` and `t`, `combiner.apply(u, accumulator.apply(identity, t))` must be `equals()` to `accumulator.apply(u, t)`.

The three-argument form is a generalization of the two-argument form, incorporating a mapping step into the accumulation step. We could re-cast the simple sum-of-weights example using the more general form as follows:

```
int sumOfWeights = widgets.stream()
                          .reduce(0,
                                  (sum, b) -> sum + b.getWeight())
                                  Integer::sum);
```

though the explicit map-reduce form is more readable and therefore should usually be preferred. The generalized form is provided for cases where significant work can be optimized away by combining mapping and reducing into a single function.
Mutable reduction
A *mutable reduction operation* accumulates input elements into a mutable result container, such as a `Collection` or `StringBuilder`, as it processes the elements in the stream.
If we wanted to take a stream of strings and concatenate them into a single long string, we *could* achieve this with ordinary reduction:

```
String concatenated = strings.reduce("", String::concat)
```

We would get the desired result, and it would even work in parallel. However, we might not be happy about the performance! Such an implementation would do a great deal of string copying, and the run time would be *O(n^2)* in the number of characters. A more performant approach would be to accumulate the results into a `StringBuilder`, which is a mutable container for accumulating strings. We can use the same technique to parallelize mutable reduction as we do with ordinary reduction.

The mutable reduction operation is called `collect()`, as it collects together the desired results into a result container such as a `Collection`. A `collect` operation requires three functions: a supplier

function to construct new instances of the result container, an accumulator function to incorporate an input element into a result container, and a combining function to merge the contents of one result container into another. The form of this is very similar to the general form of ordinary reduction:

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

As with `reduce()`, a benefit of expressing `collect` in this abstract way is that it is directly amenable to parallelization: we can accumulate partial results in parallel and then combine them, so long as the accumulation and combining functions satisfy the appropriate requirements. For example, to collect the String representations of the elements in a stream into an `ArrayList`, we could write the obvious sequential for-each form:

```
    ArrayList<String> strings = new ArrayList<>();
    for (T element : stream) {
        strings.add(element.toString());
    }
```

Or we could use a parallelizable collect form:

```
    ArrayList<String> strings = stream.collect(() -> new ArrayList<>(),
                                               (c, e) -> c.add(e.toString()),
                                               (c1, c2) -> c1.addAll(c2));
```

or, pulling the mapping operation out of the accumulator function, we could express it more succinctly as:

```
    List<String> strings = stream.map(Object::toString)
                                 .collect(ArrayList::new, ArrayList::add,
ArrayList::addAll);
```

Here, our supplier is just the `ArrayList constructor`, the accumulator adds the stringified element to an `ArrayList`, and the combiner simply uses `addAll` to copy the strings from one container into the

other.

The three aspects of `collect` -- supplier, accumulator, and combiner -- are tightly coupled. We can use the abstraction of a `Collector` to capture all three aspects. The above example for collecting strings into a `List` can be rewritten using a standard `Collector` as:

```
List<String> strings = stream.map(Object::toString)
                             .collect(Collectors.toList());
```

Packaging mutable reductions into a Collector has another advantage: composability. The class `Collectors` contains a number of predefined factories for collectors, including combinators that transform one collector into another. For example, suppose we have a collector that computes the sum of the salaries of a stream of employees, as follows:

```
Collector<Employee, ?, Integer> summingSalaries
    = Collectors.summingInt(Employee::getSalary);
```

(The `?` for the second type parameter merely indicates that we don't care about the intermediate representation used by this collector.) If we wanted to create a collector to tabulate the sum of salaries by department, we could reuse `summingSalaries` using `groupingBy`:

```
Map<Department, Integer> salariesByDept
    =
employees.stream().collect(Collectors.groupingBy(Employee::getDepartment,
                                                 summingSalaries));
```

As with the regular reduction operation, `collect()` operations can only be parallelized if appropriate conditions are met. For any partially accumulated result, combining it with an empty result container must produce an equivalent result. That is, for a partially accumulated result `p` that is the result of any series of accumulator and combiner invocations, `p` must be equivalent to `combiner.apply(p, supplier.get())`.

Further, however the computation is split, it must produce an equivalent result. For any input elements `t1` and `t2`, the results `r1` and `r2` in the computation below must be equivalent:

```
A a1 = supplier.get();
accumulator.accept(a1, t1);
accumulator.accept(a1, t2);
R r1 = finisher.apply(a1);  // result without splitting

A a2 = supplier.get();
accumulator.accept(a2, t1);
A a3 = supplier.get();
accumulator.accept(a3, t2);
R r2 = finisher.apply(combiner.apply(a2, a3));  // result with splitting
```

Here, equivalence generally means according to `Object.equals(Object)`. but in some cases equivalence may be relaxed to account for differences in order.

Reduction, concurrency, and ordering
With some complex reduction operations, for example a `collect()` that produces a `Map`, such as:

```
Map<Buyer, List<Transaction>> salesByBuyer
    = txns.parallelStream()
          .collect(Collectors.groupingBy(Transaction::getBuyer));
```

it may actually be counterproductive to perform the operation in parallel. This is because the combining step (merging one `Map` into another by key) can be expensive for some `Map` implementations. Suppose, however, that the result container used in this reduction was a concurrently modifiable collection -- such as a `ConcurrentHashMap`. In that case, the parallel invocations of the accumulator could actually deposit their results concurrently into the same shared result container, eliminating the need for the combiner to merge distinct result containers. This potentially provides a boost to the parallel execution performance. We call this a *concurrent* reduction.

A `Collector` that supports concurrent reduction is marked with the `Collector.Characteristics.CONCURRENT` characteristic. However, a concurrent collection also has a downside. If multiple threads are depositing results concurrently into a shared container, the order in which results are deposited is non-deterministic. Consequently, a concurrent reduction is only possible if ordering is not important for the stream being processed. The `Stream.collect(Collector)` implementation will only perform a concurrent reduction if

- The stream is parallel;

- The collector has the `Collector.Characteristics.CONCURRENT` characteristic, and;
- Either the stream is unordered, or the collector has the `Collector.Characteristics.UNORDERED` characteristic.

You can ensure the stream is unordered by using the `BaseStream.unordered()` method. For example:

```
Map<Buyer, List<Transaction>> salesByBuyer
    = txns.parallelStream()
          .unordered()
          .collect(groupingByConcurrent(Transaction::getBuyer));
```

(where `Collectors.groupingByConcurrent(java.util.function.Function<? super T, ? extends K>)` is the concurrent equivalent of `groupingBy`).
Note that if it is important that the elements for a given key appear in the order they appear in the source, then we cannot use a concurrent reduction, as ordering is one of the casualties of concurrent insertion. We would then be constrained to implement either a sequential reduction or a merge-based parallel reduction.

Associativity
An operator or function `op` is *associative* if the following holds:

```
(a op b) op c == a op (b op c)
```

The importance of this to parallel evaluation can be seen if we expand this to four terms:

```
a op b op c op d == (a op b) op (c op d)
```

So we can evaluate `(a op b)` in parallel with `(c op d)`, and then invoke `op` on the results.
Examples of associative operations include numeric addition, min, and max, and string concatenation.

Low-level stream construction
So far, all the stream examples have used methods like `Collection.stream()` or `Arrays.stream(Object[])` to obtain a stream. How are those stream-bearing methods implemented?
The class `StreamSupport` has a number of low-level methods for creating a stream, all using some form of a `Spliterator`. A spliterator is the parallel analogue of an `Iterator`; it describes a (possibly infinite) collection of elements, with support for sequentially advancing, bulk traversal, and splitting off some

portion of the input into another spliterator which can be processed in parallel. At the lowest level, all streams are driven by a spliterator.

There are a number of implementation choices in implementing a spliterator, nearly all of which are tradeoffs between simplicity of implementation and runtime performance of streams using that spliterator. The simplest, but least performant, way to create a spliterator is to create one from an iterator using `Spliterators.spliteratorUnknownSize(java.util.Iterator, int)`. While such a spliterator will work, it will likely offer poor parallel performance, since we have lost sizing information (how big is the underlying data set), as well as being constrained to a simplistic splitting algorithm.

A higher-quality spliterator will provide balanced and known-size splits, accurate sizing information, and a number of other `characteristics` of the spliterator or data that can be used by implementations to optimize execution.

Spliterators for mutable data sources have an additional challenge; timing of binding to the data, since the data could change between the time the spliterator is created and the time the stream pipeline is executed. Ideally, a spliterator for a stream would report a characteristic of `IMMUTABLE` or `CONCURRENT`; if not it should be *late-binding*. If a source cannot directly supply a recommended spliterator, it may indirectly supply a spliterator using a `Supplier`, and construct a stream via the `Supplier`-accepting versions of `stream()`. The spliterator is obtained from the supplier only after the terminal operation of the stream pipeline commences.

These requirements significantly reduce the scope of potential interference between mutations of the stream source and execution of stream pipelines. Streams based on spliterators with the desired characteristics, or those using the Supplier-based factory forms, are immune to modifications of the data source prior to commencement of the terminal operation (provided the behavioral parameters to the stream operations meet the required criteria for non-interference and statelessness). See Non-Interference for more details.

Since:
1.8