JEP 461: Stream Gatherers (Preview)

Owner Viktor Klang
Type Feature
Scope SE
Status Closed / Delivered
Release 22
Component core-libs / java.util.stream
Discussion core dash libs dash dev at openjdk dot org
Relates to JEP 473: Stream Gatherers (Second Preview)
Reviewed by Alan Bateman, Alex Buckley, Paul Sandoz
Endorsed by Paul Sandoz
Created 2023/10/11 13:08
Updated 2024/04/17 14:09
Issue 8317955

Summary

Enhance the Stream API to support custom intermediate operations. This will allow stream pipelines to transform data in ways that are not easily achievable with the existing built-in intermediate operations. This is a preview API.

Goals

- Make stream pipelines more flexible and expressive.
- Insofar as possible, allow custom intermediate operations to manipulate streams of infinite size.

Non-Goals

- It is not a goal to change the Java programming language to better facilitate stream processing.
- It is not a goal to special-case the compilation of code that uses the Stream API.

Motivation

Java 8 introduced the first API designed specifically for lambda expressions: the Stream API, `java.util.stream`. A stream is a lazily computed, potentially unbounded sequence of values. The API supports the ability to process a stream either sequentially or in parallel.

A *stream pipeline* consists of three parts: a source of elements, any number of intermediate operations, and a terminal operation. For example:

```
long numberOfWords =
    Stream.of("the", "", "fox", "jumps", "over", "the", "", "dog")  // (1)
          .filter(Predicate.not(String::isEmpty))                   // (2)
          .collect(Collectors.counting());                          // (3)
```

This programming style is both expressive and efficient. With the builder-style API, each intermediate operation returns a new stream; evaluation begins only when a terminal operation is invoked. In this example, line (1) creates a stream, but does not evaluate it, line (2) sets up an intermediate `filter` operation but still does not evaluate the stream, and finally the terminal `collect` operation on line (3) evaluates the entire stream pipeline.

The Stream API provides a reasonably rich, albeit fixed, set of intermediate and terminal operations: mapping, filtering, reduction, sorting, and so forth. It also includes an extensible terminal operation, `Stream::collect`, which enables the output of a pipeline to be summarized in a variety of ways.

The use of streams in the Java ecosystem is by now pervasive, and ideal for many tasks, but the fixed set of intermediate operations means that some complex tasks cannot easily be expressed as stream pipelines. Either a required intermediate operation does not exist, or it exists but does not directly support the task.

As an example, suppose the task is to take a stream of strings and make it distinct, but with distinctness based on string length rather than content. That is, at most one string of length 1 should be emitted, and at most one string of length 2, and at most one string of length 3, and so forth. Ideally, the code would look something like this:

```
var result = Stream.of("foo", "bar", "baz", "quux")
                   .distinctBy(String::length)      // Hypothetical
                   .toList();

// result ==> [foo, quux]
```

Unfortunately, `distinctBy` is not a built-in intermediate operation. The closest built-in operation, `distinct`, tracks the elements it has already seen by using object equality to compare them. That is, `distinct` is stateful but in this case uses the wrong state: We want it to track elements based on equality of string length, not string content. We could work around this limitation by declaring a class that defines object equality in terms of string length, wrapping each string in an instance of that class and applying `distinct` to those instances. This expression of the task is not intuitive, however, and makes for code that is difficult to maintain:

```
record DistinctByLength(String str) {

    @Override public boolean equals(Object obj) {
        return obj instanceof DistinctByLength(String other)
               && str.length() == other.length();
    }

    @Override public int hashCode() {
        return str == null ? 0 : Integer.hashCode(str.length());
    }

}

var result = Stream.of("foo", "bar", "baz", "quux")
                   .map(DistinctByLength::new)
                   .distinct()
                   .map(DistinctByLength::str)
                   .toList();

// result ==> [foo, quux]
```

As another example, suppose the task is to group elements into fixed-size groups of three, but retain only the first two groups: `[0, 1, 2, 3, 4, 5, 6, ...]` should produce `[[0, 1, 2], [3, 4, 5]]`. Ideally, the code would look like this:

```
var result = Stream.iterate(0, i -> i + 1)
                   .windowFixed(3)                  // Hypothetical
                   .limit(2)
                   .toList();

// result ==> [[0, 1, 2], [3, 4, 5]]
```

Unfortunately, no built-in intermediate operation supports this task. The best option is to place the fixed-window grouping logic in the terminal operation, by invoking `collect` with a custom `Collector`. However, we must precede the `collect` operation with a fixed-size `limit` operation, since the collector cannot signal to `collect` that it is finished while new elements are appearing — which happens forever with an infinite stream. Also, the task is inherently about ordered data, so it is not feasible to have the collector perform grouping in parallel, and it must signal this fact by throwing an exception if its combiner is invoked. The resulting code is difficult to understand:

```
var result
    = Stream.iterate(0, i -> i + 1)
            .limit(3 * 2)
            .collect(Collector.of(
                () -> new ArrayList<ArrayList<Integer>>(),
                (groups, element) -> {
                    if (groups.isEmpty() || groups.getLast().size() == 3) {
                         var current = new ArrayList<Integer>();
                         current.add(element);
                         groups.addLast(current);
                     } else {
                         groups.getLast().add(element);
                     }
                },
                (left, right) -> {
                    throw new UnsupportedOperationException("Cannot be parallelized");
                }
            ));

// result ==> [[0, 1, 2], [3, 4, 5]]
```

Over the years, many new intermediate operations have been suggested for the Stream API. Most of them make sense when considered in isolation, but adding all of them would make the (already large) Stream API more difficult to learn because its operations would be less discoverable.

The designers of the Stream API understood that it would be desirable to have an extension point so that anyone could define intermediate stream operations. At the time, however, they did not know what that extension point should look like. It eventually became clear that the extension point for terminal operations, namely `Stream::collect(Collector)`, was effective. We can now take a similar approach for intermediate operations.

In summary, more intermediate operations create more situational value, making streams a better fit for even more tasks. We should provide an API for custom intermediate operations that allows developers to transform finite and infinite streams in their preferred ways.

Description

`Stream::gather(Gatherer)` is a new intermediate stream operation that processes the elements of a stream by applying a user-defined entity called a *gatherer*. With the `gather` operation we can build efficient, parallel-ready streams that implement almost any intermediate operation. `Stream::gather(Gatherer)` is to intermediate operations what `Stream::collect(Collector)` is to terminal operations.

A *gatherer* represents a transform of the elements of a stream; it is an instance of the `java.util.stream.Gatherer` interface. Gatherers can transform elements in a one-to-one, one-to-many, many-to-one, or many-to-many fashion. They can track previously seen elements in order to influence the transformation of later elements, they can short-circuit in order to transform infinite streams to finite ones, and they can enable parallel execution. For example, a gatherer can transform one input element to one output element until some condition becomes true, at which time it starts to transform one input element to two output elements.

A gatherer is defined by four functions that work together:

- The optional *initializer* function provides an object that maintains private state while processing stream elements. For example, a gatherer can store the current element so that, the next time it is applied, it can compare the new element with the now-previous element and, say, emit only the larger of the two. In effect, such a gatherer transforms two input elements into one output element.
- The *integrator* function integrates a new element from the input stream, possibly inspecting the private state object and possibly emitting elements to the output stream. It can also terminate processing before reaching the end of the input stream; for example, a gatherer searching for the largest of a stream of integers can terminate if it detects `Integer.MAX_VALUE`.
- The optional *combiner* function can be used to evaluate the gatherer in parallel when the input stream is marked as parallel. If a gatherer is not parallel-capable then it can still be part of a parallel stream pipeline, but it is evaluated sequentially. This is useful for cases where an operation is inherently ordered in nature and thus cannot be parallelized.
- The optional *finisher* function is invoked when there are no more input elements to consume. This function can inspect the private state object and, possibly, emit additional output elements. For example, a gatherer searching for a specific element amongst its input elements can report failure, say by throwing an exception, when its finisher is invoked.

When invoked, `Stream::gather` performs the equivalent of the following steps:

- Create a `Downstream` object which, when given an element of the gatherer's output type, passes it to the next stage in the pipeline.
- Obtain the gatherer's private state object by invoking the `get()` method of its initializer.
- Obtain the gatherer's integrator by invoking its `integrator()` method.
- While there are more input elements, invoke the integrator's integrate(...) method, passing it the state object, the next element, and the downstream object. Terminate if that method returns `false`.
- Obtain the gatherer's finisher and invoke it with the state and downstream objects.

Every existing intermediate operation declared in the `Stream` interface can be implemented by invoking `gather` with a gatherer that implements that operation. For example, given a stream of T-typed elements, `Stream::map` turns each `T` element into a `U` element by applying a function and then passes the `U` element downstream; this is simply a stateless one-to-one gatherer. As another example, `Stream::filter` takes a predicate that determines whether an input element should be passed downstream; this is simply a stateless one-to-many gatherer. In fact every stream pipeline is, conceptually, equivalent to

```
source.gather(...).gather(...).gather(...).collect(...)
```

Built-in gatherers

We introduce the following built-in gatherers in the java.util.stream.Gatherers class:

- `fold` is a stateful many-to-one gatherer which constructs an aggregate incrementally and emits that aggregate when no more input elements exist.
- `mapConcurrent` is a stateful one-to-one gatherer which invokes a supplied function for each input element concurrently, up to a supplied limit.
- `scan` is a stateful one-to-one gatherer which applies a supplied function to the current state and the current element to produce the next element, which it passes downstream.
- `windowFixed` is a stateful many-to-many gatherer which groups input elements into lists of a supplied size, emitting the windows downstream when they are full.
- `windowSliding` is a stateful many-to-many gatherer which groups input elements into lists of a supplied size. After the first window, each subsequent window is created from a copy of its predecessor by dropping the first element and appending the next element from the input stream..

Parallel evaluation

Parallel evaluation of a gatherer is split into two distinct modes. When a combiner is not provided, the stream library can still extract parallelism by executing upstream and downstream operations in parallel, analogous to a short-circuitable `parallel().forEachOrdered()` operation. When a combiner is provided, parallel evaluation is analogous to a short-circuitable `parallel().reduce()` operation.

Composing gatherers

Gatherers support composition via the `andThen(Gatherer)` method, which joins two gatherers where the first produces elements that the second can consume. This enables the creation of sophisticated gatherers by composing simpler ones, just like function composition. Semantically,

```
source.gather(a).gather(b).gather(c).collect(...)
```

is equivalent to

```
source.gather(a.andThen(b).andThen(c)).collect(...)
```

Gatherers vs. collectors

The design of the `Gatherer` interface is heavily influenced by the design of `Collector`. The main differences are:

- `Gatherer` uses an `Integrator` instead of a `BiConsumer` for per-element processing because it needs an extra input parameter for the `Downstream` object, and because it needs to return a `boolean` to indicate whether processing should continue.
- `Gatherer` uses a `BiConsumer` for its finisher instead of a `Function` because it needs an extra input parameter for its `Downstream` object, and because it cannot return a result and thus is `void`.

Example: Embracing the stream

Sometimes the lack of an appropriate intermediate operation forces us to evaluate a stream into a list and run our analysis logic in a loop. Suppose, for example, that we have a stream of temporally ordered temperature readings:

```
record Reading(Instant obtainedAt, int kelvins) {

    Reading(String time, int kelvins) {
        this(Instant.parse(time), kelvins);
    }

    static Stream<Reading> loadRecentReadings() {
        // In reality these could be read from a file, a database,
        // a service, or otherwise
        return Stream.of(
                new Reading("2023-09-21T10:15:30.00Z", 310),
                new Reading("2023-09-21T10:15:31.00Z", 312),
                new Reading("2023-09-21T10:15:32.00Z", 350),
                new Reading("2023-09-21T10:15:33.00Z", 310)
        );
    }

}
```

Suppose, further, that we want to detect suspicious changes in this stream, defined as temperature changes of more than 30° Kelvin across two consecutive readings within a five-second window of time:

```
boolean isSuspicious(Reading previous, Reading next) {
    return next.obtainedAt().isBefore(previous.obtainedAt().plusSeconds(5))
           && (next.kelvins() > previous.kelvins() + 30
               || next.kelvins() < previous.kelvins() - 30);
}
```

This requires a sequential scan of the input stream, so we must eschew declarative stream processing and implement our analysis imperatively:

```
List<List<Reading>> findSuspicious(Stream<Reading> source) {
    var suspicious = new ArrayList<List<Reading>>();
    Reading previous = null;
    boolean hasPrevious = false;
    for (Reading next : source.toList()) {
        if (!hasPrevious) {
            hasPrevious = true;
            previous = next;
        } else {
            if (isSuspicious(previous, next))
                suspicious.add(List.of(previous, next));
            previous = next;
        }
    }
    return suspicious;
}

var result = findSuspicious(Reading.loadRecentReadings());

// result ==> [[Reading[obtainedAt=2023-09-21T10:15:31Z, kelvins=312],
//              Reading[obtainedAt=2023-09-21T10:15:32Z, kelvins=350]],
//             [Reading[obtainedAt=2023-09-21T10:15:32Z, kelvins=350],
//              Reading[obtainedAt=2023-09-21T10:15:33Z, kelvins=310]]]
```

With a gatherer, however, we can express this more succinctly:

```
List<List<Reading>> findSuspicious(Stream<Reading> source) {
    return source.gather(Gatherers.windowSliding(2))
                 .filter(window -> (window.size() == 2
                                    && isSuspicious(window.get(0),
                                                    window.get(1))))
                 .toList();
}
```

Example: Defining a gatherer

The `windowFixed` gatherer declared in the `Gatherers` class could be written as a direct implementation of the `Gatherer` interface:

```
record WindowFixed<TR>(int windowSize)
    implements Gatherer<TR, ArrayList<TR>, List<TR>>
{

    public WindowFixed {
        // Validate input
        if (windowSize < 1)
            throw new IllegalArgumentException("window size must be positive");
    }

    @Override
    public Supplier<ArrayList<TR>> initializer() {
        // Create an ArrayList to hold the current open window
        return () -> new ArrayList<>(windowSize);
    }

    @Override
    public Integrator<ArrayList<TR>, TR, List<TR>> integrator() {
        // The integrator is invoked for each element consumed
        return Gatherer.Integrator.ofGreedy((window, element, downstream) -> {

            // Add the element to the current open window
            window.add(element);

            // Until we reach our desired window size,
            // return true to signal that more elements are desired
            if (window.size() < windowSize)
                return true;

            // When the window is full, close it by creating a copy
            var result = new ArrayList<TR>(window);

            // Clear the window so the next can be started
            window.clear();

            // Send the closed window downstream
            return downstream.push(result);

        });
    }

    // The combiner is omitted since this operation is intrinsically sequential,
    // and thus cannot be parallelized

    @Override
    public BiConsumer<ArrayList<TR>, Downstream<? super List<TR>>> finisher() {
        // The finisher runs when there are no more elements to pass from
        // the upstream
        return (window, downstream) -> {
            // If the downstream still accepts more elements and the current
            // open window is non-empty, then send a copy of it downstream
            if(!downstream.isRejecting() && !window.isEmpty()) {
                downstream.push(new ArrayList<TR>(window));
                window.clear();
            }
        };
    }
}
```

Example usage:

```
jshell> Stream.of(1,2,3,4,5,6,7,8,9).gather(new WindowFixed(3)).toList()
$1 ==> [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Example: An ad-hoc gatherer

The `windowFixed` gatherer could, alternatively, be written in an ad-hoc manner via the `Gatherer.ofSequential(...)` factory method:

```
/**
 * Gathers elements into fixed-size groups. The last group may contain fewer
 * elements.
 * @param windowSize the maximum size of the groups
 * @return a new gatherer which groups elements into fixed-size groups
 * @param <TR> the type of elements the returned gatherer consumes and produces
 */
static <TR> Gatherer<TR, ?, List<TR>> fixedWindow(int windowSize) {

    // Validate input
    if (windowSize < 1)
      throw new IllegalArgumentException("window size must be non-zero");

    // This gatherer is inherently order-dependent,
    // so it should not be parallelized
    return Gatherer.ofSequential(

            // The initializer creates an ArrayList which holds the current
            // open window
            () -> new ArrayList<TR>(windowSize),

            // The integrator is invoked for each element consumed
            Gatherer.Integrator.ofGreedy((window, element, downstream) -> {

                // Add the element to the current open window
                window.add(element);

                // Until we reach our desired window size,
                // return true to signal that more elements are desired
                if (window.size() < windowSize)
                    return true;

                // When window is full, close it by creating a copy
                var result = new ArrayList<TR>(window);

                // Clear the window so the next can be started
                window.clear();

                // Send the closed window downstream
                return downstream.push(result);

            }),

            // The combiner is omitted since this operation is intrinsically sequential,
            // and thus cannot be parallelized

            // The finisher runs when there are no more elements to pass from the upstream
            (window, downstream) -> {
                // If the downstream still accepts more elements and the current
                // open window is non-empty then send a copy of it downstream
                if(!downstream.isRejecting() && !window.isEmpty()) {
                    downstream.push(new ArrayList<TR>(window));
                    window.clear();
                }
            }

    );
}
```

Example usage:

```
jshell> Stream.of(1,2,3,4,5,6,7,8,9).gather(fixedWindow(3)).toList()
$1 ==> [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Example: A parallelizable gatherer

When used in a parallel stream, a gatherer is only evaluated in parallel if it provides a combiner function. This parallelizable gatherer, for example, emits at most one element based upon a supplied selector function:

```
static <TR> Gatherer<TR, ?, TR> selectOne(BinaryOperator<TR> selector) {

    // Validate input
    Objects.requireNonNull(selector, "selector must not be null");

    // Private state to track information across elements
    class State {
        TR value;            // The current best value
        boolean hasValue;    // true when value holds a valid value
    }

    // Use the `of` factory method to construct a gatherer given a set
    // of functions for `initializer`, `integrator`, `combiner`, and `finisher`
    return Gatherer.of(

            // The initializer creates a new State instance
            State::new,

            // The integrator; in this case we use `ofGreedy` to signal
            // that this integerator will never short-circuit
            Gatherer.Integrator.ofGreedy((state, element, downstream) -> {
                if (!state.hasValue) {
                    // The first element, just save it
                    state.value = element;
                    state.hasValue = true;
                } else {
                    // Select which value of the two to save, and save it
                    state.value = selector.apply(state.value, element);
                }
                return true;
            }),

            // The combiner, used during parallel evaluation
            (leftState, rightState) -> {
                if (!leftState.hasValue) {
                    // If no value on the left, return the right
                    return rightState;
                } else if (!rightState.hasValue) {
                    // If no value on the right, return the left
                    return leftState;
                } else {
                    // If both sides have values, select one of them to keep
                    // and store it in the leftState, as that will be returned
                    leftState.value = selector.apply(leftState.value,
                                                     rightState.value);
                    return leftState;
                }
            },

            // The finisher
            (state, downstream) -> {
                // Emit the selected value, if there is one, downstream
                if (state.hasValue)
                    downstream.push(state.value);
            }

    );
}
```

Example usage, on a stream of random integers:

```
jshell> Stream.generate(() -> ThreadLocalRandom.current().nextInt())
              .limit(1000)                   // Take the first 1000 elements
              .gather(selectOne(Math::max))  // Select the largest value seen
              .parallel()                    // Execute in parallel
              .findFirst()                   // Extract the largest value
$1 ==> Optional[99822]
```

Alternatives

We explored alternatives in a separate design document.

Risks and Assumptions

- The use of custom gatherers, and of the built-in gatherers declared in the `Gatherers` class, will not be as succinct as the use of the built-in intermediate operations declared in the `Stream` class. The definition of custom gatherers will, however, be similar in complexity to the definition of custom collectors for terminal `collect` operations. The use of both custom and built-in gatherers will, moreover, be similar in complexity to the use of custom collectors and the built-in collectors declared in the `Collectors` class.
- We might revise the set of built-in gatherers over the course of previewing this feature, and we might revise the set of built-in gatherers in future releases.
- We will not add a new intermediate operation to the `Stream` class for each of the built-in gatherers defined in the `Gatherers` class, even though for the sake of uniformity it is tempting to do so. In order to preserve the learnability of the `Stream` class we will consider adding new intermediate operations to it only after experience suggests that they are broadly useful. We might add such methods in a later round of preview, or even after this feature is final. Exposing new built-in gatherers now does not preclude adding dedicated `Stream` methods later.