Overview of Java 8 Streams (Part 4)

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

Institute for Software
Integrated Systems

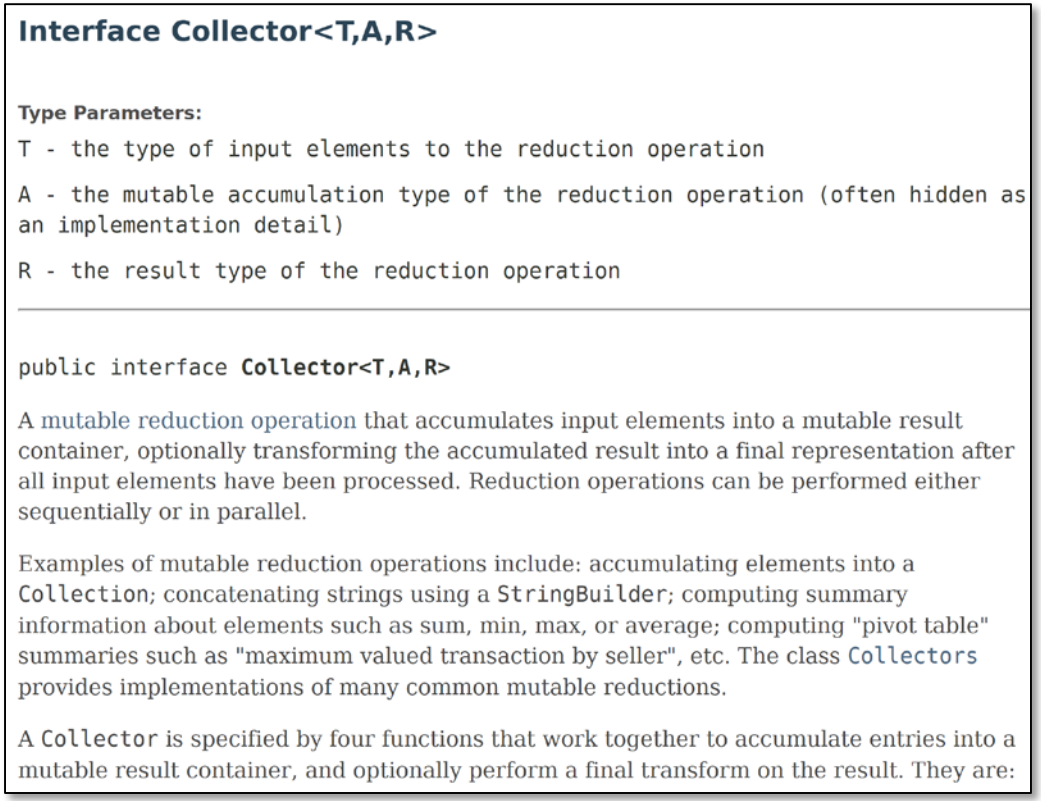
Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of Java 8 streams, e.g.,
  - Fundamentals of streams
  - Common stream aggregate operations
  - "Splittable iterators" (Spliterators)
  - Terminating a stream

Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of Java 8 streams, e.g.,
  - Fundamentals of streams
  - Common stream aggregate operations
  - "Splittable iterators" (Spliterators)
  - Terminating a stream
    - We'll show how to implement collectors



Interface Collector<T,A,R>

Type Parameters:

T - the type of input elements to the reduction operation

A - the mutable accumulation type of the reduction operation (often hidden as an implementation detail)

R - the result type of the reduction operation

---

public interface **Collector<T,A,R>**

A mutable reduction operation that accumulates input elements into a mutable result container, optionally transforming the accumulated result into a final representation after all input elements have been processed. Reduction operations can be performed either sequentially or in parallel.

Examples of mutable reduction operations include: accumulating elements into a `Collection`; concatenating strings using a `StringBuilder`; computing summary information about elements such as sum, min, max, or average; computing "pivot table" summaries such as "maximum valued transaction by seller", etc. The class `Collectors` provides implementations of many common mutable reductions.

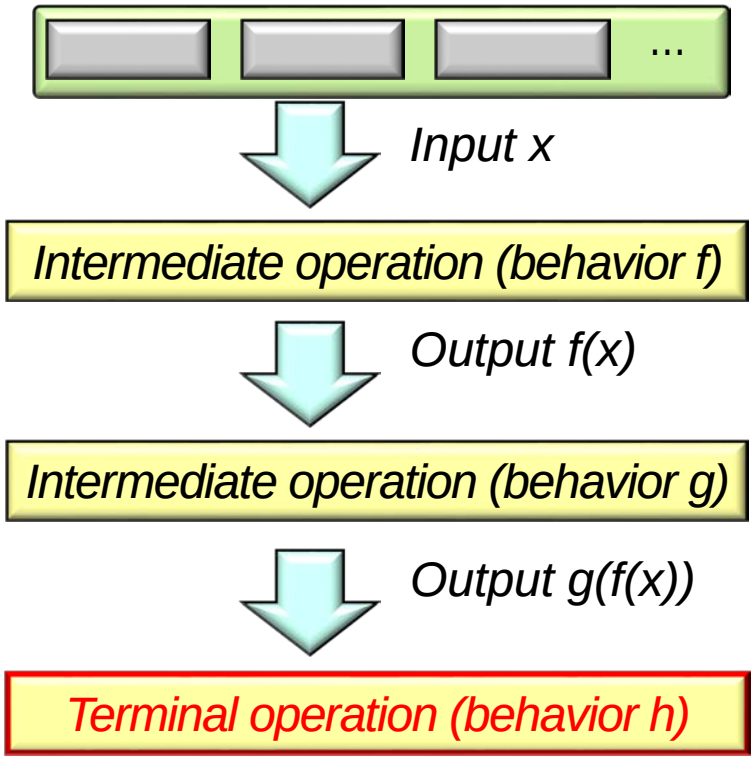
A `Collector` is specified by four functions that work together to accumulate entries into a mutable result container, and optionally perform a final transform on the result. They are:

See docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.html

Terminating a Stream

Terminating a Stream

- Every stream finishes with a terminal operation that yields a non-stream result



See docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#forEach

Terminating a Stream

- Every stream finishes with a terminal operation that yields a non-stream result, e.g.
  - no value at all

```
void runForEach() {
  Stream
    .of("horatio",
        "laertes",
        "Hamlet", ...)
    .filter(s -> toLowerCase
           (s.charAt(0)) == 'h')
    .map(this::capitalize)
    .sorted()
    .forEach
       (System.out::println);
...
```

See docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#forEach

Terminating a Stream

- Every stream finishes with a terminal operation that yields a non-stream result, e.g.
  - no value at all
  - a collection

```
void runCollect() {
  List<String> characters =
    Arrays.asList("horatio",
                  "laertes",
                  "Hamlet",
                  ...);
  List<String> results =
    characters
      .stream()
      .filter(s ->
        toLowerCase(…) =='h')
    .map(this::capitalize)
    .sorted()
    .collect(toList()); ...
```

See docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#collect

Terminating a Stream

- Every stream finishes with a terminal operation that yields a non-stream result, e.g.
  - no value at all
  - a collection

```
void runCollect() {
  List<String> characters =
    Arrays.asList("horatio",
                  "laertes",
                  "Hamlet",
                  ...);
Map<String, Long> results =
  ...
  .collect
    (groupingBy
      (identity(),
       TreeMap::new,
       summingLong
        (String::length)));
...
```

collect() can be used with a range of powerful collectors ,e.g., to group by name & length of name

See docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html

Terminating a Stream

- Every stream finishes with a terminal operation that yields a non-stream result, e.g.
  - no value at all
  - a collection

```
void runCollect() {
  List<String> characters =
    Arrays.asList("horatio",
                  "laertes",
                  "Hamlet",
                  ...);
Map<String, Long> results =
  ...
  .collect
     (groupingBy
       (identity(),
        TreeMap::new,
        summingLong
         (String::length)));
...
```

See docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html#groupingBy

Terminating a Stream

- Every stream finishes with a terminal operation that yields a non-stream result, e.g.
  - no value at all
  - a collection
  - a primitive value

```
void runCollectReduce() {
  Map<String, Long>
    matchingCharactersMap =
    Pattern.compile(",")
      .splitAsStream
        ("horatio,Hamlet,…")
    ...
  long countOfNameLengths =
    matchingCharactersMap
      .values()
      .stream()
      .reduce(0L,
              (x, y) -> x + y);
      // Could use .sum()
```

See docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#reduce

Terminating a Stream

- Every stream finishes with a terminal operation that yields a non-stream result, e.g.
  - no value at all
  - a collection
  - a primitive value

```
void runCollectReduce() {
  Map<String, Long>
    matchingCharactersMap =
    Pattern.compile(",")
      .splitAsStream
        ("horatio,Hamlet,…")
    ...
  long countOfNameLengths =
    matchingCharactersMap
      .values()
      .stream()
      .reduce(0L,
              (x, y) -> x + y);
      // Could use .sum()
```

0 is the "identity," i.e., the initial value of the reduction & the default result if there are no elements in the stream

Terminating a Stream

- Every stream finishes with a terminal operation that yields a non-stream result, e.g.
  - no value at all
  - a collection
  - a primitive value

```
void runCollectReduce() {
  Map<String, Long>
    matchingCharactersMap =
    Pattern.compile(",")
      .splitAsStream
        ("horatio,Hamlet,…")
    ...
  long countOfNameLengths =
    matchingCharactersMap
      .values()
      .stream()
      .reduce(0L,
              (x, y) -> x + y);
      // Could use .sum()
```

This lambda is the "accumulator," which is a stateless function that combines two values

Terminating a Stream

- Every stream finishes with a terminal operation that yields a non-stream result, e.g.
  - no value at all
  - a collection
  - a primitive value

```
void runCollectReduce() {
  Map<String, Long>
    matchingCharactersMap =
    Pattern.compile(",")
      .splitAsStream
        ("horatio,Hamlet,…")
    ...
  long countOfNameLengths =
    matchingCharactersMap
      .values()
      .stream()
      .reduce(0L,
              (x, y) -> x + y,
              (x, y) -> x + y);
```

There's a 3 parameter "map/reduce" version of reduce() that's used in parallel streams

See www.youtube.com/watch?v=oWlWEKNM5Aw

Implementing a Collector

Implementing a Collector

- Collector defines an interface whose implementations can accumulate input elements in a mutable result container

Interface Collector<T,A,R>

Type Parameters:

`T` - the type of input elements to the reduction operation

`A` - the mutable accumulation type of the reduction operation (often hidden as an implementation detail)

`R` - the result type of the reduction operation

`public interface Collector<T,A,R>`

A mutable reduction operation that accumulates input elements into a mutable result container, optionally transforming the accumulated result into a final representation after all input elements have been processed. Reduction operations can be performed either sequentially or in parallel.

Examples of mutable reduction operations include: accumulating elements into a `Collection`; concatenating strings using a `StringBuilder`; computing summary information about elements such as sum, min, max, or average; computing "pivot table" summaries such as "maximum valued transaction by seller", etc. The class `Collectors` provides implementations of many common mutable reductions.

A `Collector` is specified by four functions that work together to accumulate entries into a mutable result container, and optionally perform a final transform on the result. They are:

See docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.html

Implementing a Collector

- The Collector interface defines three generic types

<<Java Interface>>

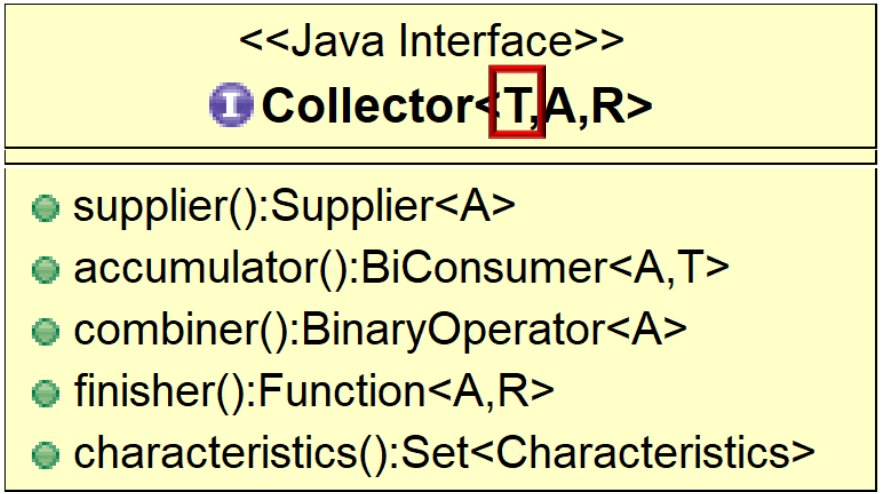
Collector<T,A,R>

- supplier():Supplier<A>
- accumulator():BiConsumer<A,T>
- combiner():BinaryOperator<A>
- finisher():Function<A,R>
- characteristics():Set<Characteristics>

See www.baeldung.com/java-8-collectors

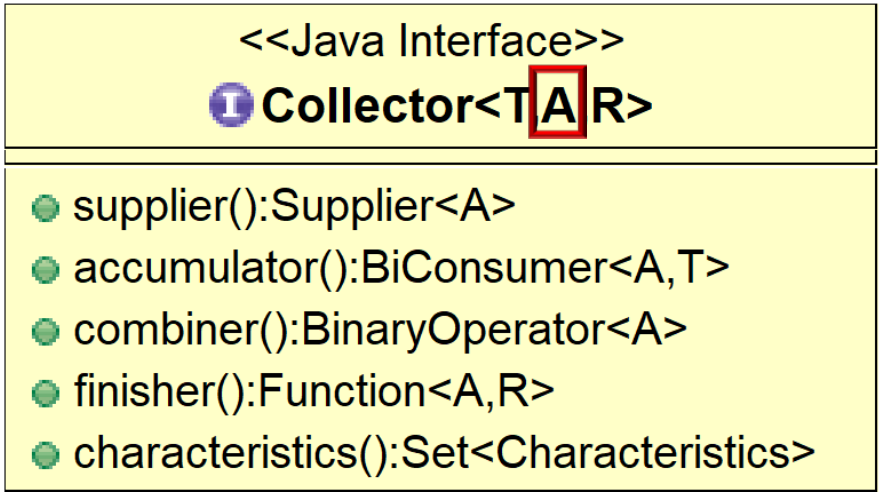
Implementing a Collector

- The Collector interface defines three generic types
  - **T** – The type of objects available
    - e.g., Integer or Long



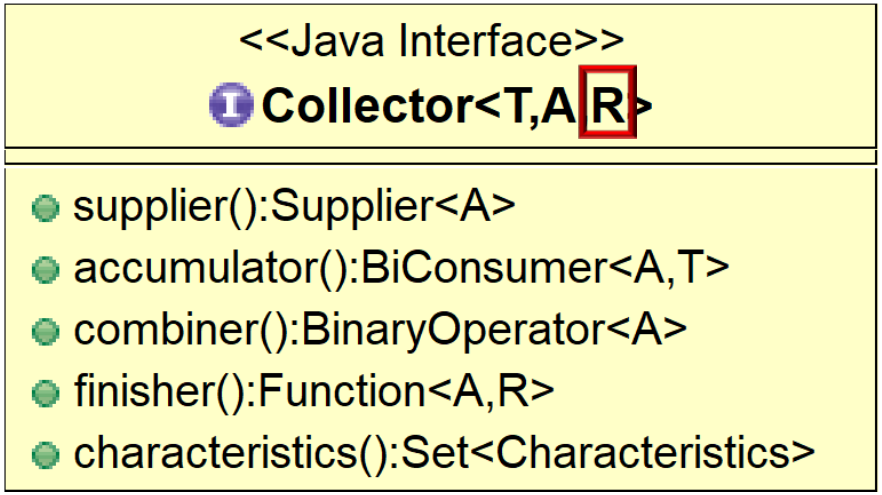
Implementing a Collector

- The Collector interface defines three generic types
  - T
  - **A** – The type of a mutable accumulator object for collection
    - e.g., ArrayList of T



Implementing a Collector

- The Collector interface defines three generic types
  - T
  - A
  - **R** – The type of a final result
    - e.g., ArrayList of T



See www.baeldung.com/java-8-collectors

Implementing a Collector

- Five methods are defined in the Collector interface

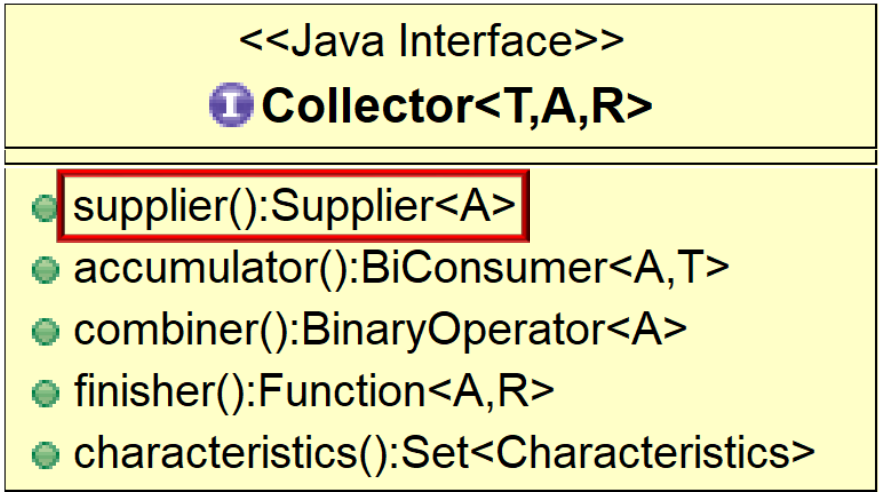
<<Java Interface>>

Collector<T,A,R>

- supplier():Supplier<A>
- accumulator():BiConsumer<A,T>
- combiner():BinaryOperator<A>
- finisher():Function<A,R>
- characteristics():Set<Characteristics>

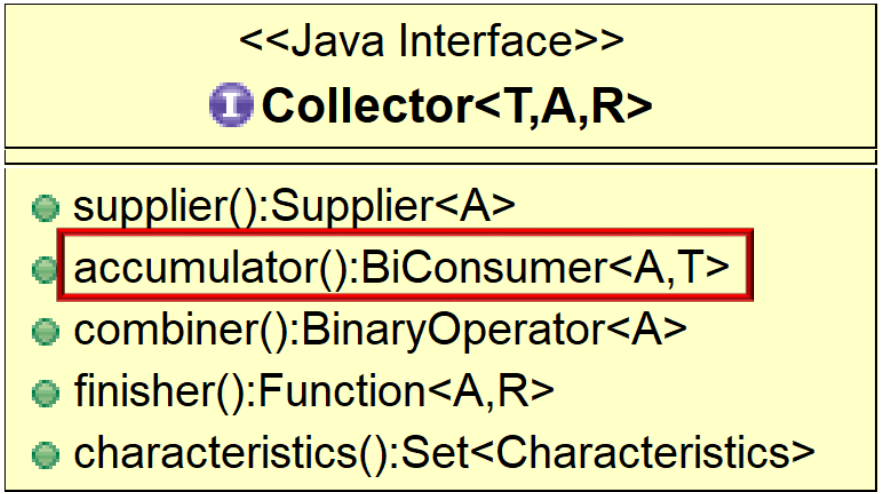
Implementing a Collector

- Five methods are defined in the Collector interface
  - **supplier()** – returns a Supplier instance that generates an empty accumulator
    - e.g., `return ArrayList::new`



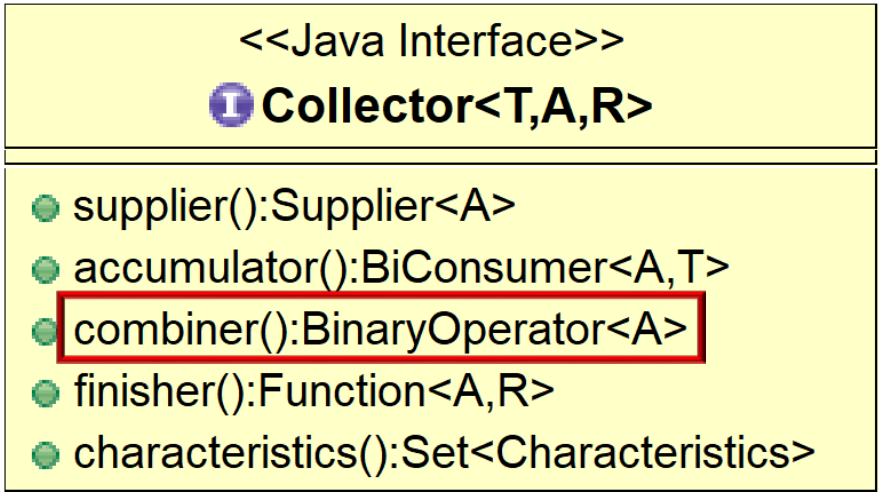
Implementing a Collector

- Five methods are defined in the Collector interface
  - supplier()
  - **accumulator()** – returns a function that adds a new element to an existing accumulator
    - e.g., `return ArrayList::add`



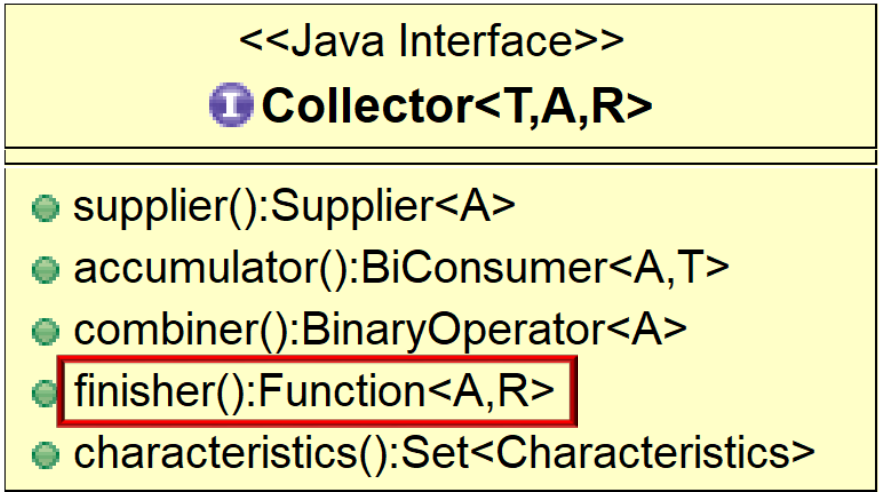
Implementing a Collector

- Five methods are defined in the Collector interface
  - supplier()
  - accumulator()
  - **combiner()** – returns a function that merges two accumulators together
    - e.g., `return(List<T> one, List<T> another) -> { one.addAll(another); return one; };`



Implementing a Collector

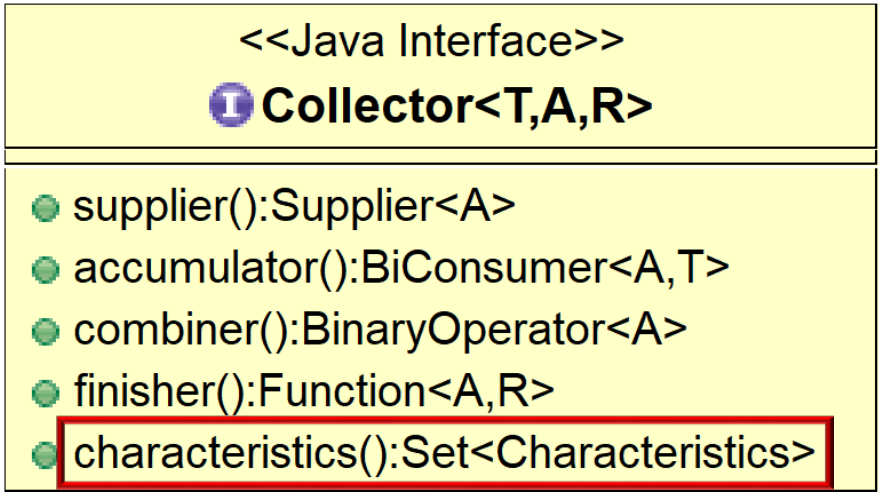
- Five methods are defined in the Collector interface
  - supplier()
  - accumulator()
  - combiner()
  - **finisher()** – returns a function that converts an accumulator to final result type
    - e.g., **`return Function.identity()`**



May be a no-op, depending on characteristics

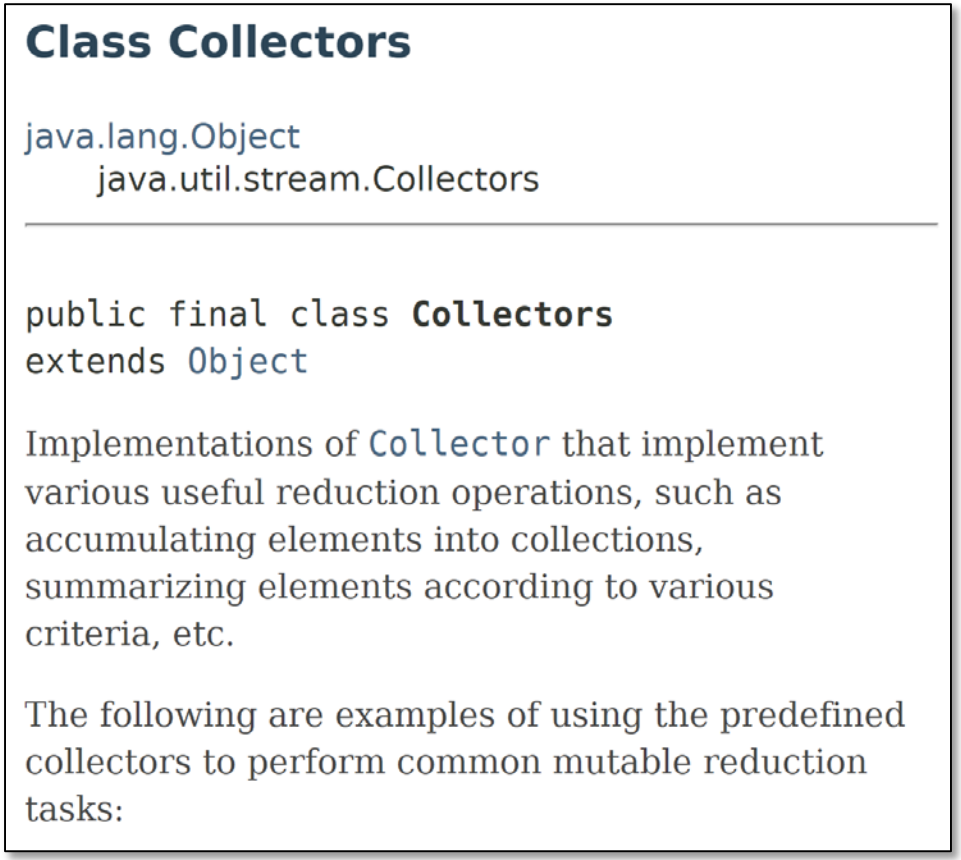
Implementing a Collector

- Five methods are defined in the Collector interface
  - **supplier()**
  - **accumulator()**
  - **combiner()**
  - **finisher()**
  - **characteristics()** – provides a stream with additional information used for internal optimizations
    - e.g., UNORDERED, IDENTIFY_FINISH, CONCURRENT



Implementing a Collector

- The Java class library defines collectors for common types



Class Collectors

java.lang.Object
    java.util.stream.Collectors

```
public final class Collectors
extends Object
```

Implementations of `Collector` that implement various useful reduction operations, such as accumulating elements into collections, summarizing elements according to various criteria, etc.

The following are examples of using the predefined collectors to perform common mutable reduction tasks:

See docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html

Implementing a Collector

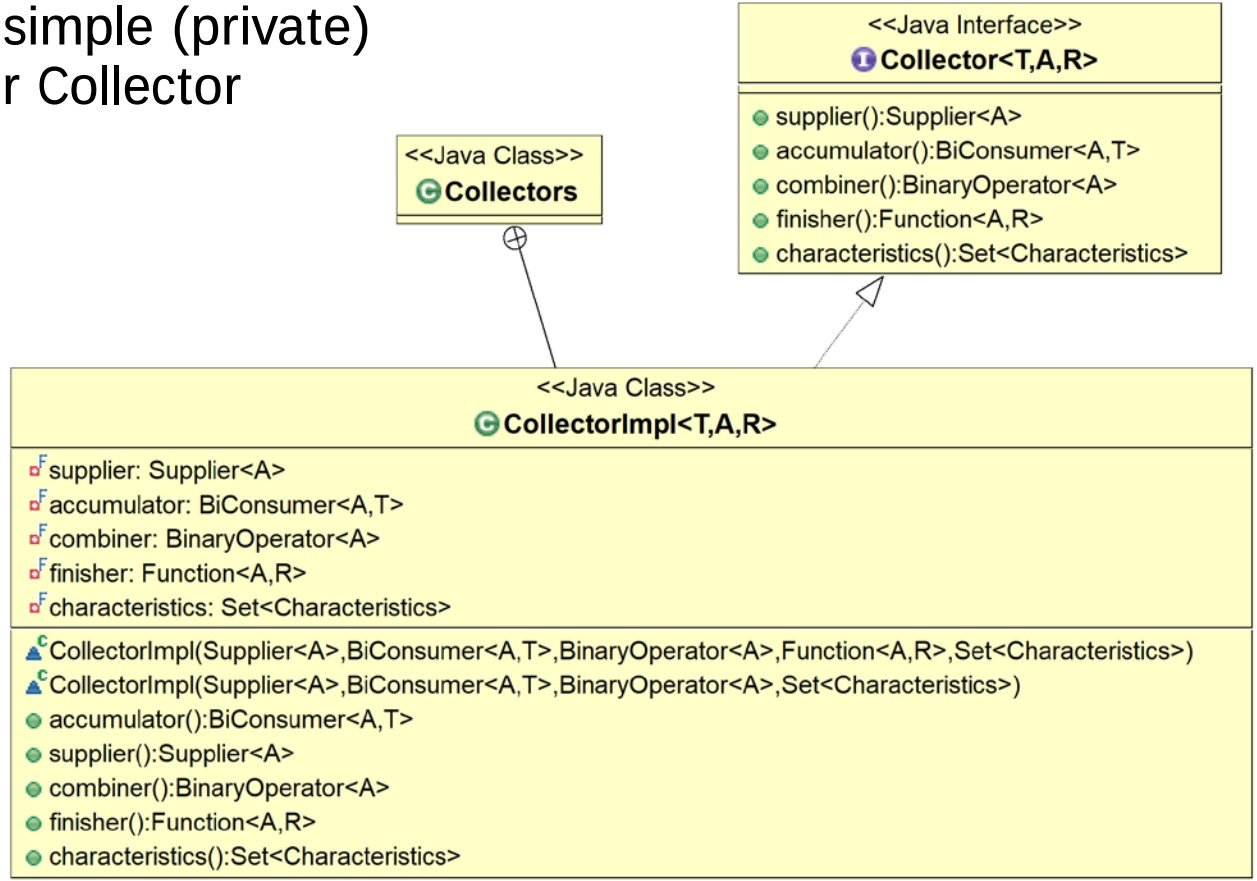
- The Java class library defines collectors for common types
  - e.g., returns a Collector that accumulates input elements into a new (Array)List

```
final class Collectors {
  ...
  public static <T> Collector<T, ?,
                              List<T>>
    toList() {
      return new CollectorImpl<>
        ((Supplier<List<T>>)
         ArrayList::new,
         List::add,
         (left, right) -> {
            left.addAll(right);
            return left;
         },
         CH_ID);
    } ...
```

See docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html#toList

Implementing a Collector

- CollectorImpl defines a simple (private) implementation class for Collector



See openjdk/8-b132/java/util/stream/Collectors.java#Collectors.CollectorImpl

Implementing a Collector

- Collector.of() defines a simple (public) factory method that implements a Collector using CollectorImpl

```
interface Collector<T, A, R> {
  ...
  static<T, R> Collector<T, R, R> of
    (Supplier<R> supplier,
     BiConsumer<R, T> accumulator,
     BinaryOperator<R> combiner,
     Characteristics... chars) {
     ...
    return new Collectors
      .CollectorImpl<>
        (supplier,
         accumulator,
         combiner,
         cs);
    } ...
```

See openjdk/8-b132/java/util/stream/Collectors.java#Collectors.CollectorImpl

Implementing a Collector

- You can implement custom collectors via Collector.of()

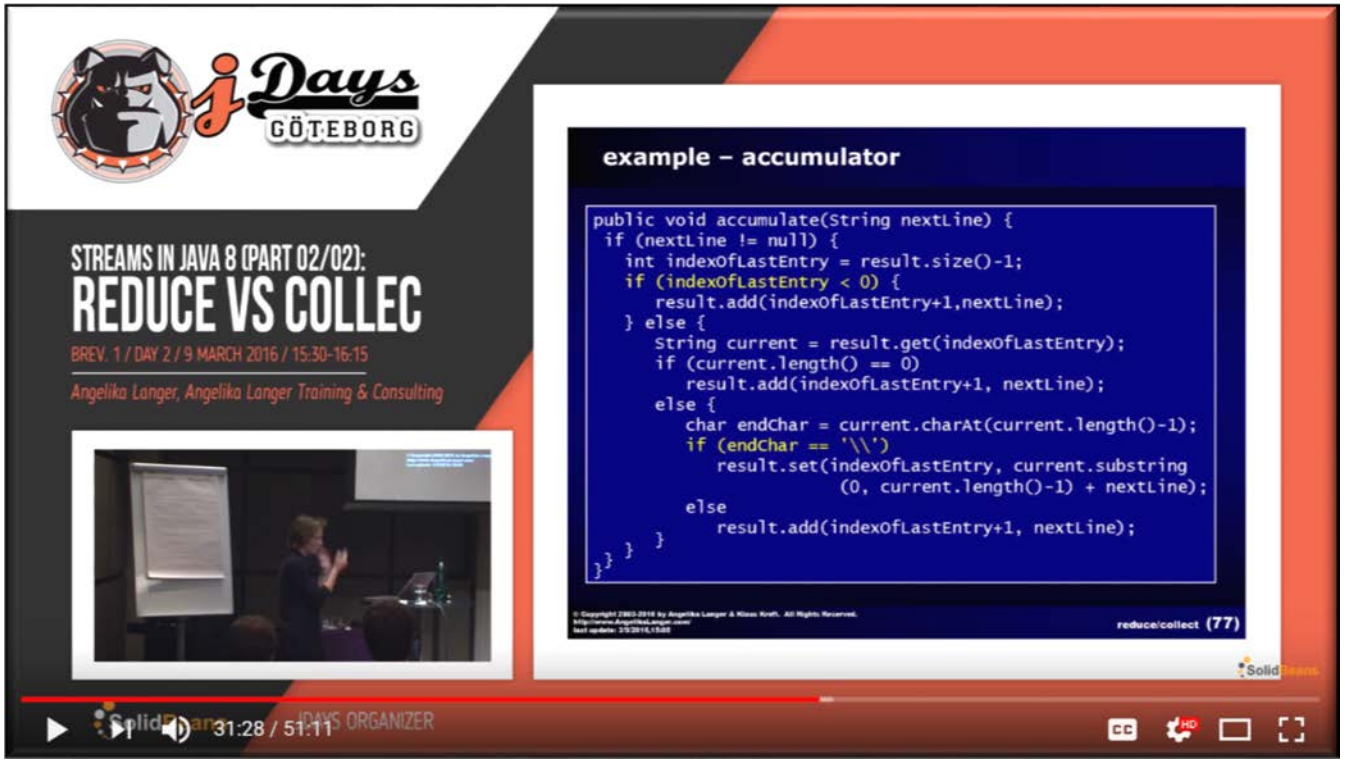
```
mList
   .stream()
   .collect(Collector.of
     (() -> new StringJoiner("|"),
      (j, result) ->
         j.add(result.toString()),
      StringJoiner::merge,
      StringJoiner::toString));
```

SearchResults's custom collector formats itself

See SimpleSearchStream/src/main/java/search/SearchResults.java

Implementing a Collector

- More information on implementing custom collectors can be found online



See www.youtube.com/watch?v=H7VbRz9aj7c

End of Overview of Java 8 Streams (Part 4)