Stream API

An *aggregate* operation computes a single value from a collection of values.

A *stream* is a sequence of data elements supporting sequential and parallel aggregate operations.

Differences between streams and collections:

- Collections focus on storage of data elements for efficient access.
  - Collections support imperative programming using external iteration.

- Streams focus on aggregate computations on data elements from a data source that is typically, but not necessarily, collections.
  - Streams have no storage.
  - Streams can represent a sequence of infinite elements.
  - Streams are designed to support functional (declarative) programming using internal iteration.
  - Streams support lazy operations.
  - Streams can be ordered or unordered.
  - Streams are designed to be processed in parallel with no additional work from the developers.
  - Streams cannot be reused.

Methods in the `Stream` interface

All Methods**Static Methods****Instance Methods****Abstract Methods****Default Methods**

Modifier and Type	Method and Description
boolean	**allMatch**(**Predicate**<? super **T**> predicate) Returns whether all elements of this stream match the provided predicate.
boolean	**anyMatch**(**Predicate**<? super **T**> predicate) Returns whether any elements of this stream match the provided predicate.
static <T> **Stream.Builder**<T>	**builder**() Returns a builder for a Stream.
<R,A> R	**collect**(**Collector**<? super **T**,A,R> collector) Performs a **mutable reduction** operation on the elements of this stream using a Collector.
<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.
static <T> **Stream**<T>	**concat**(**Stream**<? extends T> a, **Stream**<? extends T> b) Creates a lazily concatenated stream whose elements are all the elements of the first stream followed by all the elements of the second stream.
long	**count**() Returns the count of elements in this stream.
Stream<**T**>	**distinct**() Returns a stream consisting of the distinct elements (according to **Object.equals(Object)**) of this stream.
static <T> **Stream**<T>	**empty**() Returns an empty sequential Stream.
Stream<**T**>	**filter**(**Predicate**<? super **T**> predicate) Returns a stream consisting of the elements of this stream that match the given predicate.
Optional<**T**>	**findAny**() Returns an **Optional** describing some element of the stream, or an empty Optional if the stream is empty.
Optional<**T**>	**findFirst**() Returns an **Optional** describing the first element of this stream, or an empty Optional if the stream is empty.
<R> **Stream**<R>	**flatMap**(**Function**<? super **T**,? extends **Stream**<? extends R>> mapper) Returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.
DoubleStream	**flatMapToDouble**(**Function**<? super **T**,? extends **DoubleStream**> mapper) Returns an DoubleStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.
IntStream	**flatMapToInt**(**Function**<? super **T**,? extends **IntStream**> mapper) Returns an IntStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.

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

T	reduce(T identity, BinaryOperator<T> accumulator) Performs a reduction on the elements of this stream, using the provided identity value and an associative accumulation function, and returns the reduced value.
<U> U	reduce(U identity, 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.
Stream<T>	skip(long n) Returns a stream consisting of the remaining elements of this stream after discarding the first n elements of the stream.
Stream<T>	sorted() Returns a stream consisting of the elements of this stream, sorted according to natural order.
Stream<T>	sorted(Comparator<? super T> comparator) Returns a stream consisting of the elements of this stream, sorted according to the provided Comparator.
Object[]	toArray() Returns an array containing the elements of this stream.
<A> A[]	toArray(IntFunction<A[]> generator) Returns an array containing the elements of this stream, using the provided generator function to allocate the returned array, as well as any additional arrays that might be required for a partitioned execution or for resizing.

Remark:

- Lazy (non result-bearing) operator returns a `Stream`

- Eager (result-bearing) operator returns a value (object of some result type) or void.

Example codes:

Compute the sum of the squares of odd values in a collection using conventional imperative programming (external iteration).

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

int sum = 0;
for (int n : numbers)
   if (n % 2 == 1)
   {
      int square = n * n;
      sum = sum + square;
   }
```

Compute the sum of the squares of odd values in a stream using declarative programming (internal iteration).

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

int sum = numbers.stream()
                 .filter(n -> n % 2 == 1)
                 .map(n -> n * n)
                 .reduce(0, Integer::sum);

/*
int sum = 0;               // identity value
for (int n : numbers)      // numbers.stream()
   if (n % 2 == 1)         //        .filter(Predicate)
   {
      int square = n * n;  //        .map(Function)
      sum = sum + square;  //        .reduce(identity,
   }                       //                BinaryOperator)
*/
```

Streams are designed to process their elements in parallel with built-in support using the Fork/Join framework (the Fork/Join framework will be discussed in multi-thread program design).

```
int sum = numbers.parallelStream()
                 .filter(n -> n % 2 == 1)
                 .map(n -> n * n)
                 .reduce(0, Integer::sum);
```

Intermediate and Terminal operations

Terminal operations are known as *eager* (or *result-bearing*) operations.

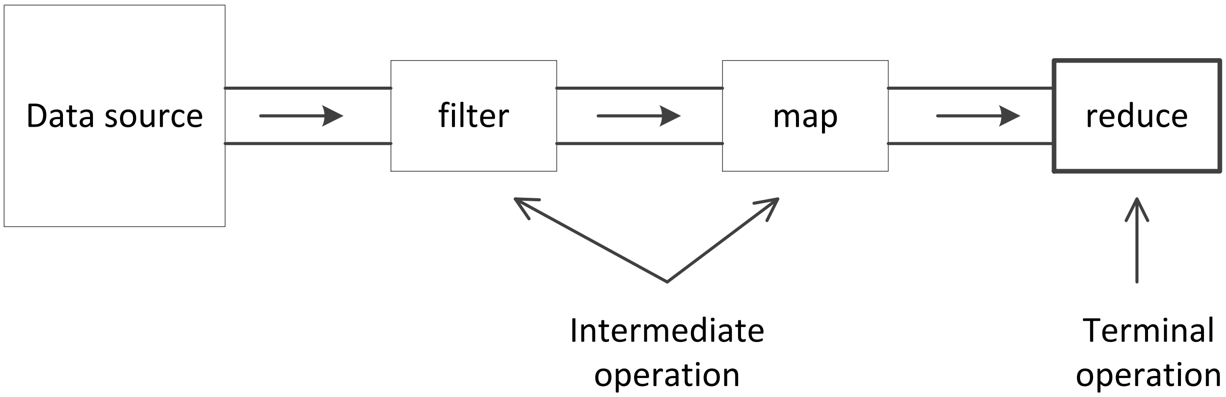
Intermediate operations are known as *lazy* (or *non result-bearing*) operations.

- A lazy operation on a stream does not process the elements of the stream until an eager operation is called on the stream.
- Stream processing does not start until a terminal operation is called.

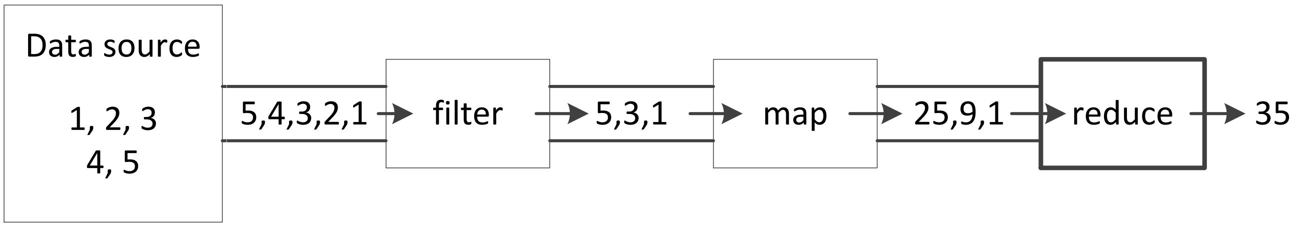
Example:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
int sum = numbers.stream()
                 .filter(n -> n % 2 == 1)
                 .map(n -> n * n)
                 .reduce(0, Integer::sum);
```

A stream pipeline corresponding to the above example:



Visualization of the stream pipeline:



Streams are not reusable.

A stream cannot be reused after calling a terminal operation on it.

If you need to perform a computation on the same elements from the same data source again, you must recreate the stream pipeline.

Debugging a Stream pipeline

Each operation in the stream pipeline transforms the elements of the input stream either producing another stream or a result.

Sometimes you may need to look at the elements of the streams as they pass through the pipeline.

You can do so by using the **`peek(Consumer<? super T> action)`** method.

Example

```
int sum = numbers.stream()
                 .filter(n -> n % 2 == 1)
                 .peek(e -> System.out.println(
                            "Filtered element: " + e))
                 .map(n -> n * n)
                 .peek(e -> System.out.println(
                            "Mapped element: " + e))
                 .reduce(0, Integer::sum);
```

Creating `Stream` from values using the static method `of()`

```
    <T> Stream<T> of(T t)          // single value
    <T> Stream<T> of(T... values) // multiple values
```

```
Stream<Integer> intStream = Stream.of(1,2,3,4,5);
```

The `Stream` interface also supports creating a stream using the `Stream.Builder<T>` interface.

The `Stream.Builder<T>` interface contains the following methods:

```
    void accept(T t)
    Stream.Builder<T> add(T t)
    Stream<T> build()
```

Example:

```
// Obtain a builder
Stream.Builder<Integer> builder = Stream.builder();

// Add elements and build the stream
Stream<Integer> intStream = builder.add(1)
                                   .add(2)
                                   .add(3)
                                   .add(4)
                                   .add(5)
                                   .build();

// A more convenient way to build an integer stream using
// the IntStream interface
IntStream oneToFive = IntStream.range(1, 6);

// Or
IntStream oneToFive = IntStream.rangeClosed(1, 5);
```

Stream from file

We can read text from a file as a stream of strings in which each element represents one line of text from the file.

We need a method that reads a file lazily and returns the contents as a stream of strings.

The `Scanner` class is not suitable for this purpose.

We may use the method `lines()` in the `java.nio.file.Files` class.

Example:

```
String filename = “testdata.txt”;
Path filepath = Paths.get(filename);

try (Stream<String> lines = Files.lines(filepath))
{
   lines.forEach(System.out::println);
}
catch (IOException e)
{
   e.printStackTrace();
}
```

Generating an infinite stream by program

An infinite stream is a stream with a data source capable of generating infinite number of elements.

The `Stream` interface contains 2 static methods to generate an infinite stream

```
<T> Stream<T> iterate(T seed, UnaryOperator<T> f)
// elements: seed, f(seed), f(f(seed)), f(f(f(seed))), ...

<T> Stream<T> generate(Supplier<T> s)


Example:

// Create a stream of odd natural numbers
Stream<Long> oddNaturalNum = Stream.iterate(1L, n -> n+2);


// Create a stream of the first 10 odd natural numbers
// and print it to standard output using the forEach method.
// Data type L for the seed value must be provided.
// Compiler uses it to infer the data type of stream elements.
Stream.iterate(1L, n -> n+2)
      .limit(10)
      .forEach(System.out::println);
```

Example: Data source with more complex logic.
Generate a stream of prime numbers

```
// PrimeUtil.java
public class PrimeUtil
{
   private long lastPrime = 1L;

   public long next()  // instance method
   {
      lastPrime = next(lastPrime); // call static method
      return lastPrime;
   }

   public static long next(long after)
   {
      long counter = after;
      while (!isPrime(++counter))
         ;
      return counter;
   }

   public static boolean isPrime(long num)
   {
      if (num <= 1)
         return false;

      if (num == 2)
         return true;

      if (num % 2 == 0)
         return false;

      long maxDivisor = (long)Math.sqrt(num);
      for (int k = 3; k <= maxDivisor; k += 2)
         if (num % k == 0)
            return false;

      return true;
   }
}
```

```
// Print the first 5 prime numbers: 2, 3, 5, 7, 11
// Data type L for the seed value must be provided.
// Compiler uses it to infer the data type of stream elements.
// Use static method next(long)
Stream.iterate(2L, PrimeUtil::next) // n -> PrimeUtil.next(n)
      .limit(5)
      .forEach(System.out::println);


// Alternative design
Stream.iterate(2L, n -> n+1)
      .filter(PrimeUtil::isPrime) // n -> PrimeUtil.isPrime(n)
      .limit(5)
      .forEach(System.out::println);


// Design using the Stream.generate(Supplier) method,
// and skip the first 100 prime numbers
// use instance method primeUtilObj.next()
Stream.generate(new PrimeUtil()::next)
      .skip(100)
      .limit(5)
      .forEach(System.out::println);


/*
Supplier<Long> s = new Supplier()
                   {
                      PrimeUtil pu = new PrimeUtil();
                      public Long get()
                      {
                         return pu.next();
                      }
                   };

Stream.generate(s)
      .skip(100)
      .limit(5)
      .forEach(System.out::println);
*/
```

Finding and Matching in Stream

The Stream API supports different types of find and match operations on stream elements.

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

Optional<T> findAny()
Optional<T> findFirst()
```

Examples

```
List<Person> persons = Person.persons();

// Check if all persons in the list are male
boolean allMales = persons.stream()
                          .allMatch(Person::isMale);


// Check if any person was born in 1970
boolean anyoneBornIn1970 =
   persons.stream()
          .anyMath(p -> p.getDob().getYear() == 1970);


// Find the first male
Optional<Person> firstMale =
   persons.stream()
          .filter(Person::isMale)
          .findFirst();

System.out.println(firstMale.orElse("No male found"));
```

Collecting output data of a stream using Collectors

The `collect()` method of the `Stream<T>` interface

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

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

Consider the 1st version of the `collect()` method that requires 3 arguments,

- a **`supplier`** that supplies a container to store (collect) the results
- an **`accumulator`** that accumulates the results into the container
- a **`combiner`** that combines the partial results when the reduction operation takes place in parallel (i.e. using parallel stream)

Suppose we have a steam of people, and we want to collect the names of all the people in an `ArrayList<String>`.

```
// Create supplier, accumulator, combiner using
// Lambda expression
Supplier<ArrayList<String>>
   supplier = () -> new ArrayList();

BiConsumer<ArrayList<String>, String>
   accumulator = (list, name) -> list.add(name);

BiConsumer<ArrayList<String>, ArrayList<String>>
   combiner = (list1, list2) -> list1.addAll(list2);
```

```
// Create supplier, accumulator, combiner using
// method reference
Supplier<ArrayList<String>> supplier = ArrayList::new;

BiConsumer<ArrayList<String>, String>
   accumulator = ArrayList::add;

BiConsumer<ArrayList<String>, ArrayList<String>>
  combiner = ArrayList::addAll;
```

To collect the names of all people in a list

```
ArrayList<Person> persons = new ArrayList();
...

Supplier<ArrayList<String>> supplier = ArrayList::new;

BiConsumer<ArrayList<String>, String>
   accumulator = ArrayList::add;

BiConsumer<ArrayList<String>, ArrayList<String>>
   combiner = ArrayList::addAll;

List<String> names = persons.stream()
                            .map(Person::getName)
                            .collect(supplier,
                                     accumulator,
                                     combiner);

/* Using anonymous objects

List<String> names = persons.stream()
                            .map(Person::getName)
                            .collect(ArrayList::new,
                                     ArrayList::add,
                                     ArrayList::addAll);
*/
```

The utility class `Collectors` provides out-of-box implementation for commonly used collectors.

All Methods**Static Methods****Concrete Methods**

Modifier and Type	Method and Description
static <T> **Collector**<T,?,**Double**>	**averagingDouble**(**ToDoubleFunction**<? super T> mapper) Returns a Collector that produces the arithmetic mean of a double-valued function applied to the input elements.
static <T> **Collector**<T,?,**Double**>	**averagingInt**(**ToIntFunction**<? super T> mapper) Returns a Collector that produces the arithmetic mean of an integer-valued function applied to the input elements.
static <T> **Collector**<T,?,**Double**>	**averagingLong**(**ToLongFunction**<? super T> mapper) Returns a Collector that produces the arithmetic mean of a long-valued function applied to the input elements.
static <T,A,R,RR> **Collector**<T,A,RR>	**collectingAndThen**(**Collector**<T,A,R> downstream, **Function**<R,RR> finisher) Adapts a Collector to perform an additional finishing transformation.
static <T> **Collector**<T,?,**Long**>	**counting**() Returns a Collector accepting elements of type T that counts the number of input elements.
static <T,K> **Collector**<T,?,**Map**<K,**List**<T>>>	**groupingBy**(**Function**<? super T,? extends K> classifier) Returns a Collector implementing a "group by" operation on input elements of type T, grouping elements according to a classification function, and returning the results in a Map.
static <T,K,A,D> **Collector**<T,?,**Map**<K,D>>	**groupingBy**(**Function**<? super T,? extends K> classifier, **Collector**<? super T,A,D> downstream) Returns a Collector implementing a cascaded "group by" operation on input elements of type T, grouping elements according to a classification function, and then performing a reduction operation on the values associated with a given key using the specified downstream Collector.
static <T,K,D,A,M extends **Map**<K,D>> **Collector**<T,?,M>	**groupingBy**(**Function**<? super T,? extends K> classifier, **Supplier**<M> mapFactory, **Collector**<? super T,A,D> downstream) Returns a Collector implementing a cascaded "group by" operation on input elements of type T, grouping elements according to a classification function, and then performing a reduction operation on the values associated with a given key using the specified downstream Collector.
static <T,K> **Collector**<T,?,**ConcurrentMap**<K,**List**<T>>>	**groupingByConcurrent**(**Function**<? super T,? extends K> classifier) Returns a concurrent Collector implementing a "group by" operation on input elements of type T, grouping elements according to a classification function.
static <T,K,A,D> **Collector**<T,?,**ConcurrentMap**<K,D>>	**groupingByConcurrent**(**Function**<? super T,? extends K> classifier, **Collector**<? super T,A,D> downstream) Returns a concurrent Collector implementing a cascaded

	"group by" operation on input elements of type T, grouping elements according to a classification function, and then performing a reduction operation on the values associated with a given key using the specified downstream Collector.
static <T,K,A,D,M extends **ConcurrentMap**<K,D>> **Collector**<T,?,M>	**groupingByConcurrent**(**Function**<? super T,? extends K> classifier, **Supplier**<M> mapFactory, **Collector**<? super T,A,D> downstream) Returns a concurrent Collector implementing a cascaded "group by" operation on input elements of type T, grouping elements according to a classification function, and then performing a reduction operation on the values associated with a given key using the specified downstream Collector.
static **Collector**<**CharSequence**,?,**String**>	**joining**() Returns a Collector that concatenates the input elements into a String, in encounter order.
static **Collector**<**CharSequence**,?,**String**>	**joining**(**CharSequence** delimiter) Returns a Collector that concatenates the input elements, separated by the specified delimiter, in encounter order.
static **Collector**<**CharSequence**,?,**String**>	**joining**(**CharSequence** delimiter, **CharSequence** prefix, **CharSequence** suffix) Returns a Collector that concatenates the input elements, separated by the specified delimiter, with the specified prefix and suffix, in encounter order.
static <T,U,A,R> **Collector**<T,?,R>	**mapping**(**Function**<? super T,? extends U> mapper, **Collector**<? super U,A,R> downstream) Adapts a Collector accepting elements of type U to one accepting elements of type T by applying a mapping function to each input element before accumulation.
static <T> **Collector**<T,?,**Optional**<T>>	**maxBy**(**Comparator**<? super T> comparator) Returns a Collector that produces the maximal element according to a given Comparator, described as an Optional<T>.
static <T> **Collector**<T,?,**Optional**<T>>	**minBy**(**Comparator**<? super T> comparator) Returns a Collector that produces the minimal element according to a given Comparator, described as an Optional<T>.
static <T> **Collector**<T,?,**Map**<**Boolean**,**List**<T>>>	**partitioningBy**(**Predicate**<? super T> predicate) Returns a Collector which partitions the input elements according to a Predicate, and organizes them into a Map<Boolean, List<T>>.
static <T,D,A> **Collector**<T,?,**Map**<**Boolean**,D>>	**partitioningBy**(**Predicate**<? super T> predicate, **Collector**<? super T,A,D> downstream) Returns a Collector which partitions the input elements according to a Predicate, reduces the values in each partition according to another Collector, and organizes them into a Map<Boolean, D> whose values are the result of the downstream reduction.
static <T> **Collector**<T,?,**Optional**<T>>	**reducing**(**BinaryOperator**<T> op) Returns a Collector which performs a reduction of its input elements under a specified BinaryOperator.
static <T> **Collector**<T,?,T>	**reducing**(T identity, **BinaryOperator**<T> op)

	Returns a Collector which performs a reduction of its input elements under a specified BinaryOperator using the provided identity.
static <T,U> **Collector**<T,?,U>	**reducing**(U identity, **Function**<? super T,? extends U> mapper, **BinaryOperator**<U> op) Returns a Collector which performs a reduction of its input elements under a specified mapping function and BinaryOperator.
static <T> **Collector**<T,?,**DoubleSummaryStatistics**>	**summarizingDouble**(**ToDoubleFunction**<? super T> mapper) Returns a Collector which applies an double-producing mapping function to each input element, and returns summary statistics for the resulting values.
static <T> **Collector**<T,?,**IntSummaryStatistics**>	**summarizingInt**(**ToIntFunction**<? super T> mapper) Returns a Collector which applies an int-producing mapping function to each input element, and returns summary statistics for the resulting values.
static <T> **Collector**<T,?,**LongSummaryStatistics**>	**summarizingLong**(**ToLongFunction**<? super T> mapper) Returns a Collector which applies an long-producing mapping function to each input element, and returns summary statistics for the resulting values.
static <T> **Collector**<T,?,**Double**>	**summingDouble**(**ToDoubleFunction**<? super T> mapper) Returns a Collector that produces the sum of a double-valued function applied to the input elements.
static <T> **Collector**<T,?,**Integer**>	**summingInt**(**ToIntFunction**<? super T> mapper) Returns a Collector that produces the sum of a integer-valued function applied to the input elements.
static <T> **Collector**<T,?,**Long**>	**summingLong**(**ToLongFunction**<? super T> mapper) Returns a Collector that produces the sum of a long-valued function applied to the input elements.
static <T,C extends **Collection**<T>> **Collector**<T,?,C>	**toCollection**(**Supplier**<C> collectionFactory) Returns a Collector that accumulates the input elements into a new Collection, in encounter order.
static <T,K,U> **Collector**<T,?,**ConcurrentMap**<K,U>>	**toConcurrentMap**(**Function**<? super T,? extends K> keyMapper, **Function**<? super T,? extends U> valueMapper) Returns a concurrent Collector that accumulates elements into a ConcurrentMap whose keys and values are the result of applying the provided mapping functions to the input elements.
static <T,K,U> **Collector**<T,?,**ConcurrentMap**<K,U>>	**toConcurrentMap**(**Function**<? super T,? extends K> keyMapper, **Function**<? super T,? extends U> valueMapper, **BinaryOperator**<U> mergeFunction) Returns a concurrent Collector that accumulates elements into a ConcurrentMap whose keys and values are the result of applying the provided mapping functions to the input elements.
static <T,K,U,M extends **ConcurrentMap**<K,U>> **Collector**<T,?,M>	**toConcurrentMap**(**Function**<? super T,? extends K> keyMapper, **Function**<? super T,? extends U> valueMapper, **BinaryOperator**<U> mergeFunction, **Supplier**<M> mapSupplier)

	Returns a concurrent Collector that accumulates elements into a ConcurrentMap whose keys and values are the result of applying the provided mapping functions to the input elements.
static <T> **Collector**<T,?,**List**<T>>	**toList**() Returns a Collector that accumulates the input elements into a new List.
static <T,K,U> **Collector**<T,?,**Map**<K,U>>	**toMap**(**Function**<? super T,? extends K> keyMapper, **Function**<? super T,? extends U> valueMapper) Returns a Collector that accumulates elements into a Map whose keys and values are the result of applying the provided mapping functions to the input elements.
static <T,K,U> **Collector**<T,?,**Map**<K,U>>	**toMap**(**Function**<? super T,? extends K> keyMapper, **Function**<? super T,? extends U> valueMapper, **BinaryOperator**<U> mergeFunction) Returns a Collector that accumulates elements into a Map whose keys and values are the result of applying the provided mapping functions to the input elements.
static <T,K,U,M extends **Map**<K,U>> **Collector**<T,?,M>	**toMap**(**Function**<? super T,? extends K> keyMapper, **Function**<? super T,? extends U> valueMapper, **BinaryOperator**<U> mergeFunction, **Supplier**<M> mapSupplier) Returns a Collector that accumulates elements into a Map whose keys and values are the result of applying the provided mapping functions to the input elements.
static <T> **Collector**<T,?,**Set**<T>>	**toSet**() Returns a Collector that accumulates the input elements into a new Set.

Three of the most commonly used methods of the `Collectors` class are `toList()`, `toSet()`, and `toCollection()`.

The above example can be written using `Collectors.toList()`

```
List<String> names = Person.persons()
                           .stream()
                           .map(Person::getName)
                           .collect(Collectors.toList());
```

To obtain a set of unique names (removing duplicates)

```
Set<String>
   uniqueNames = Person.persons()
                       .stream()
                       .map(Person::getName)
                       .collect(Collectors.toSet());
```

To obtain a set of unique names sorted in natural order

```
SortedSet<String> uniqueSortedNames =
    Person.persons()
          .stream()
          .map(Person::getName)
          .collect(Collectors.toCollection(TreeSet::new));
```

Example: Find the top10 most popular video

```
public class VideoRec
{
    private final long timestamp;
    private final String vid;
    private final String client;

    public VideoRec(long t, String v, String c)
    {
        timestamp = t;
        vid = v;
        client = c;
    }

    public long getTimestamp()
    {
        return timestamp;
    }

    public String getVid()
    {
        return vid;
    }

    public String getClient()
    {
        return client;
    }

    @Override
    public String toString()
    {
        return timestamp + "," + vid + "," + client;
    }
}
```

```
public class Pair<S, T>
{
    private S first;
    private T second;

    public Pair(S n1, T n2)
    {
        first = n1;
        second = n2;
    }

    public S getFirst()
    {
        return first;
    }

    public T getSecond()
    {
        return second;
    }

    public void setFirst(S v)
    {
        first = v;
    }

    public void setSecond(T v)
    {
        second = v;
    }

    @Override
    public String toString()
    {
        return "(" + first + ", " + second + ")";
    }
}
```

1. Conventional imperative programming

```
ArrayList<VideoRec> list = readDataFile(fname);

list.sort(comparing(VideoRec::getVid));

ArrayList<Pair<String, Integer>> viewCountList = new ArrayList();

int i = 0;
while (i < list.size())
{
   String curVid = list.get(i).getVid();
   int j = i + 1;
   while (j < list.size() && list.get(j).getVid().equals(curVid))
      j++;

   viewCountList.add(new Pair(curVid, j-i));
   i = j;
}

viewCountList.sort((a, b) -> b.getSecond() - a.getSecond());

int end = (viewCountList.size() >= 10) ? 10 : viewCountList.size();

viewCountList.subList(0, end).forEach(System.out::println);
```

```
private static ArrayList<VideoRec> readDataFile(String fname)
{
   // Read in the VideoRec from data file
   ArrayList<VideoRec> list = new ArrayList();

   try (Scanner sc = new Scanner(new File(fname)))
   {
      while (sc.hasNextLine())
      {
         String line = sc.nextLine();
         String[] token = line.split(",");
         list.add(new VideoRec(Long.parseLong(token[0]),
                                                  token[1], token[2]));
      }
   }
   catch(FileNotFoundException e)
   {  }
   return list;
}
```

---

2. Functional programming using the `FunctionUtil` class

```
ArrayList<VideoRec> list = readDataFile(fname);

list.sort(comparing(VideoRec::getVid));

BiConsumer<List<Pair<String, Integer>>, VideoRec> action =
  (result, v) -> {
      if (result.isEmpty())
          result.add(new Pair(v.getVid(), 1));
      else
      {
          Pair<String, Integer> item = result.get(result.size()-1);
          if (item.getFirst().equals(v.getVid()))
              item.setSecond(item.getSecond() + 1);
          else
              result.add(new Pair(v.getVid(), 1));
      }
  };

List<Pair<String, Integer>>
viewCountList = FunctionUtil.transform(list, action);

viewCountList.sort((a, b) -> b.getSecond() - a.getSecond());

// Cannot use viewCountList.sort(comparing(Pair::getSecond).reversed())

int end = (viewCountList.size() >= 10) ? 10 : viewCountList.size();

viewCountList.subList(0, end).forEach(System.out::println);
```

3. Functional programming using the `Stream` API

```
BiConsumer<ArrayList<Pair<String, Integer>>, VideoRec> accumulator =
    (result, v) -> {
        if (result.isEmpty())
            result.add(new Pair(v.getVid(), 1));
        else
        {
            Pair<String, Integer> item = result.get(result.size()-1);
            if (item.getFirst().equals(v.getVid()))
                item.setSecond(item.getSecond()+1);
            else
                result.add(new Pair(v.getVid(), 1));
        }
    };

Function<String, VideoRec> mapper =
    line -> {
       String[] token = line.split(",");
       return new VideoRec(Long.parseLong(token[0]),
                           token[1], token[2]);
   };


try (Stream<String> lines = Files.lines(filepath))
{
    lines.map(mapper) // map a line to VideoRec
         .sorted(comparing(VideoRec::getVid))
         .collect(ArrayList::new,
                  accumulator,
                  ArrayList::addAll)
         .stream()
         .sorted((a, b) -> b.getSecond() - a.getSecond())
         .limit(10)
         .forEach(System.out::println);
}
catch(IOException e)
{ }
```

Collecting summary statistics

In data-centric application, very often we need to compute the summary statistics on a group of numeric data.

Java provides 3 classes to collect statistics

```
java.util.DoubleSummaryStatistics
java.util.LongSummaryStatistics
java.util.IntSummaryStatistics
```

Commonly used methods in the above classes

```
accept()    // add a value to the data set
getCount()
getSum()
getMin()
getAverage()
getMax()
```

Example code to compute the statistics of the income of the persons in a list.

```
// External iteration
DoubleSummaryStatistics stats = new
                                DoubleSummaryStatistics();

List<Person> persons = Person.persons();
for (Person p : persons)
   stats.accept(p.getIncome());

// Get statistics
long count = stats.getCount();
double sum = stats.getSum();
double min = stats.getMin();
...

/* Alternative design using Stream
DoubleSummaryStatistics stats =
    Person.persons()
          .stream()
          .map(Person::getIncome)
          .collect(DoubleSummaryStatistics::new,
                   DoubleSummaryStatistics::accept,
                   DoubleSummaryStatistics::combine);

*/
```

Collecting data from a stream into a `Map` using `Collectors.toMap()`

```
// Map object is created by the toMap() method
toMap(Function<? super T, ? extends K> keyMapper,
      Function<? super T, ? extends U> valueMapper)

// Map object is created by the toMap() method
// mergeFunction is used to resolve collisions in the Map
toMap(Function<? super T, ? extends K> keyMapper,
      Function<? super T, ? extends U> valueMapper,
      BinaryOperator<U> mergeFunction)

// User to provide a Supplier to create the Map object
toMap(Function<? super T, ? extends K> keyMapper,
      Function<? super T, ? extends U> valueMapper,
      BinaryOperator<U> mergeFunction,
      Supplier<M> mapSupplier)
```

Example 1

To collect student’s name based on student ID (student ID is unique) from a list.

```
List<Student> list = ... ;

Map<Integer, String> sidToNamesMap =
    list.stream()
        .collect(Collectors.toMap(Student::getSid,
                                  Student::getName));
```

Example 2

To collect person’s name based on gender (gender is not unique) from a list.

```
List<Person> list = ... ;

Map<Gender, String> genderToNamesMap =
    list.stream()
        .collect(Collectors.toMap(
                    Person::getGender,
                    Person::getName,
                    (s1, s2) -> String.join(“, ”, s1, s2)));

// mergeFunction : (s1, s2) -> String.join(“, ”, s1, s2)
// s1 = oldValue (existing names)
// s2 = newValue (name of current element)
```

Example 3
To summarize the number of persons in a list by gender

```
List<Person> list = ... ;
Map<Gender, Long> countByGender =
    list.stream()
        .collect(Collectors.toMap(
                       Person::getGender,
                       p -> 1L,
                       (oldCount, newCount) -> oldCount++));

// valueMapper : p -> 1L
// When a person belonging to a gender is encountered the
// first time, the value is set to 1.
```

Joining `Strings` using `Collectors`

```
joining()

joining(CharSequence delimiter)

// prefix is added to the beginning of result
// suffix is added to the end of result
joining(CharSequence delimiter,
        CharSequence prefix,
        CharSequence suffix)
```

Example 4
Collecting the names of persons, delimited by “,”

```
List<Person> list = ... ;
String names = list.stream()
                   .map(Person::getName)
                   .collect(Collectors.joining(“, ”));
```

Example 5
Collecting the names of persons, delimited by “,” and with the prefix “List of names” and suffix “END”

```
String names = list.stream()
                   .map(Person::getName)
                   .collect(Collectors.joining(“, ”,
                    “List of names: “, “END”));
```

Grouping data

Grouping data for reporting purposes is common.

The `Collectors.groupingBy()` method returns a collector that groups the data before collecting them in a `Map`.

```
// A non-concurrent Map is returned.
// Has performance overhead when the steam is processed
// in parallel.
// The groupingByConcurrent method will return a concurrent
// collector that is more suitable for parallel processing.
groupingBy(Function<? super T, ? extends K> classifier)

groupingBy(Function<? super T, ? extends K> classifier,
           Collector<? super T,A,D> downstream)


// User specifies a Supplier to create the Map object
groupingBy(Function<? super T, ? extends K> classifier,
           Supplier<M> mapFactory,
           Collector<? super T,A,D> downstream)
```

- **classifier** is a function to generate the keys in the map
- collector (**downstream**) performs a reduction operator on the values associated with each key

Example 5
Count the number of persons by gender. Alternative implement of example 3.

```
List<Person> list = ... ;
Map<Gender, Long> countByGender =
    list.stream()
        .collect(Collectors.groupingBy(
                     Person::getGender,
                     Collectors.counting()));

// The Collectors.counting() method count the number of
// elements in a stream.
```

Example 6
Collect the names of persons grouped by gender.
Alternative implementation of example 2.

```
List<Person> list = ... ;

Map<Gender, String> genderToNamesMap =
    list.stream()
        .collect(Collectors.groupingBy(
                    Person::getGender,
                    Collectors.mapping(
                        Person.getName,
                        Collectors.joining(‶, ″))));

// The first step groups the elements by gender.
// The second step uses the mapping() method to extract
// the name and a nested collector (Collectors.joining())
// to merge the names.
```

Partitioning data

- Partitioning data is a special case of grouping data.
- Grouping data is based on the keys returned by the key extractor (mapper) function.
- Partitioning data is based on a predicate .
- Note that the `Map` returned from the collector always contains 2 entries: one with the key value as `true` and another with the key value as `false`.
- The values for a key are stored in a `List`.

```
partitioningBy(Predicate<? super T> predicate)

partitioningBy(Predicate<? super T> predicate,
               Collector<? super T,A,D> downstream)
```

Example 7,
A variant of the implementation of example 6.

```
List<Person> list = ... ;

// Note the data type of the Key field.
Map<Boolean, String> partitionByMaleGender =
    list.stream()
        .collect(Collectors.partitioningBy(Person::isMale,
                 Collectors.mapping(
                      Person::getName,
                      Collectors.joining(“, ”))));
```

Adapting the Collector results

There is one more type of collector that collects the data, and before returning the result to the caller, lets you modify the result in any way you want.

Such a collector is returned by using the `collectingAndThen()` method of the `Collectors` class.

```
collectingAndThen(Collector<T,A,R> downstream,
                  Function<R, RR> finisher)
```

Example
We want to print a calendar that contains the names of people by the month of their birth.

```
Map<Month, String> dobCalendar =
    Person.persons()
          .stream()
          .collect(groupingBy(p -> p.getDob().getMonth(),
                   mapping(Person::getName, joining(", ")))));

dobCalendar.entrySet().forEach(System.out::println);

// dobCalendar.entrySet() returns a set view of the Map
// forEach() method of Set requires a Consumer<T>

// Remark: forEach() method of Map requires a BiConsumer<K, V>
```

The output may look like:

```
JANUARY=John
SEPTEMBER=Wally, Donna
```

The output may not be sorted by month.
The output may not contain all months.
The returned `Map` from the `collect()` method is modifiable.

We want to modify the program such that

- the output is sorted by month
- add the missing month
- wrap the `Map` in an unmodifiable `Map`

```
// Revised codes
Map<Month, String> dobCalendar =
   Person.persons()
         .stream()
         .collect(collectingAndThen(
                 groupingBy(p -> p.getDob().getMonth(),
                 mapping(Person::getName, joining(", "))),

                 // finisher
                 result -> {
                    for (Month m : Month.values())
                       result.putIfAbsent(m, "None");

                    return Collections.unmodifiableMap(
                           new TreeMap(result));
                 }));

dobCalendar.entrySet().forEach(System.out::println);
```

The output looks like:

```
JANUARY=John
FEBURARY=None
MARCH=None
...
SEPTEMBER=Wally, Donna
OCTOBER=None
NOVEMBER=None
DECEMBER=None
```