**Collectors**

**1. Joining**

Collectors.joining() is a collector that concatenates the elements of a Stream into a single String.

Methods:

* joining() = Joins elements without any delimiter
* joining(CharSequence delimiter) = Joins elements using the specified delimiter between them.
* joining(CharSequence delimiter, CharSequence prefix, CharSequence suffix) = Joins elements with a delimiter between elements, a prefix at the start and a suffix at the end.

**2. Averaging**

Collectors.averagingXXX() are collectors that calculate the average (mean) of numeric values extracted from the elements of a stream.

Methods:

* averagingInt(ToIntFunction<T> mapper)
* averagingLong(ToLongFunction<T> mapper)
* averagingDouble(ToDoubleFunction<T> mapper)

All of them return Double so the result will have decimal precision.

You can use method references or lambda expressions for mapping to a numeric value.

**3. Summing**

Collectors.summingXXX() are collectors that compute the sum of numeric values extracted from the elements of a stream.

Methods:

* summingInt(ToIntFunction<T> mapper)
* summingLong(ToLongFunction<T> mapper)
* summingDouble(ToDoubleFunction<T> mapper)

Use summingDouble if you want to avoid overflow with large sums or want decimal precision.

**4. Min and Max**

Used to find the maximum or minimum element of a stream according to a specified Comparator.

Methods:

* minBy(Comparator<? super T> comparator)
* maxBy(Comparator<? super T> comparator)

Return an Optional<T> because the stream might be empty.

**6. Reducing**

Collectors.reducing() is a generalized reduction collector that allows you to accumulate elements of a stream into a single result, applying a combining function (like sum, min, max, or any custom logic).

Methods:

* reducing(BinaryOperator<T> op)

Applies the given operator to reduce elements.

Returns Optional<T>, since no identity means the stream could be empty.

* reducing(U identity, BinaryOperator<U> op)

Starts with the identity value, and applies the operator to reduce the elements.

No Optional always returns a result (even for empty streams).

* reducing(U identity, Function<T, U> mapper, BinaryOperator<U> op)

Maps each element to a value first, then reduces.

**7. Counting**

Collectors.counting() counts the number of elements in a stream. It returns a Long representing the count.

It is equivalent to stream.count(), but useful in collect() chains, especially with groupingBy or partitioningBy.

Methods:

* counting()

**8. Filtering**

Collectors.filtering() is a collector that filters elements of a stream during collection inside the collect() pipeline. It allows you to apply a filter condition while collecting (e.g., inside a groupingBy, partitioningBy, etc).

Methods:

* filtering(Predicate predicate, Collector downstream)

A Predicate is condition to filter elements.

A downstream collector that defines what to do with the filtered elements.

**9. FlatMapping**

Collectors.flatMapping() can be used on a stream elements containing nested collections (e.g., lists, sets) and you want to flatten them inside groupingBy, partitioningBy.

Collectors.flatMapping() lets you:

* Map each element of a stream into a stream of zero or more elements and flatten those streams into a single stream.
* Then apply a downstream collector (e.g. toList, toSet, counting, etc).

If you have a List of List, you can’t apply Collectors.flatMapping() directly on the outer stream to flatten it.

Methods:

* flatMapping(Function mapper, Collector downstream)

**10. CollectingAndThen​**

Collectors.collectingAndThen() is a wrapper collector that:

* First applies a downstream collector (like toList, toSet, counting, etc).
* Then applies a finisher function on the result of that downstream collector.

Methods:

* collectingAndThen(Collector downstream, Function finisher)

**11. ToList, ToUnmodifiableList, ToSet, ToUnmodifiableSet**

Returns a Collector that accumulates the input elements into a new List or Set.

**12. ToMap**

Collectors.toMap() collects stream elements into a Map by:

* Applying a key mapping function and a value mapping function.
* (optionally) specifying a merge function (for duplicate keys).
* (optionally) specifying a map supplier (e.g. TreeMap, LinkedHashMap)

Methods:

* toMap(Function keyMapper, Function valueMapper) = Throws IllegalStateException if duplicate keys occur.
* toMap(Function keyMapper, Function valueMapper, BinaryOperator mergeFunction) = Use merge function to resolve key collisions.
* toMap(Function keyMapper, Function valueMapper, BinaryOperator mergeFunction, Supplier mapSupplier) = Let you specify the map implementation (e.g. TreeMap, LinkedHashMap).

**13. GroupingBy**

Collectors.groupingBy is a collector that:

* Groups elements of a stream by a classifier function.
* Returns a Map (or another type if you specify a downstream collector).

Methods:

* groupingBy(Function classifier) = Groups elements into lists by the classifier key.
* groupingBy(Function classifier, Collector downstream) = Groups elements, and applies a downstream collector to elements in each group.
* groupingBy(Function classifier, Supplier mapFactory, Collector downstream) = You can specify the type of map (e.g. TreeMap, LinkedHashMap) to collect into.

**Comparator**

**1. NaturalOrder**

Comparator.naturalOrder() compares elements using their natural order (e.g., String, Integer, etc).

**2. ReverseOrder**

Comparator.reverseOrder() compares elements using their natural order and returns them in reversed order.

**3. Comparing**

Methods:

* comparing(Function keyExtractor)
* comparing(Function keyExtractor, Comparator keyComparator)
* comparingInt​(ToIntFunction keyExtractor)
* comparingLong​(ToLongFunction keyExtractor)
* comparingDouble(ToDoubleFunction keyExtractor)

**4. NullsFirst and NullsLast**

A null-friendly comparator that considers null to be less/greater than non-null.

Methods:

* nullsFirst​(Comparator comparator)
* nullsLast​(Comparator comparator)

**5. ThenComparing**

Tie-breaker comparison.

Methods:

* thenComparing​(Comparator other)
* thenComparing​(Function keyExtractor)
* thenComparing​(Function keyExtractor, Comparator keyComparator)

**Stream Intermediate Operations**

**Intermediate operations are lazy and return a new stream. They don’t process data until a terminal operation is invoked.**

**1. Filtering**

Filter (select) elements that match a given condition.

Methods:

* filter​(Predicate predicate)

**2. Mapping**

Transforms each element of the stream using a mapping function.

Methods:

* map​(Function mapper)
* mapToInt(ToIntFunction mapper)
* mapToDouble(ToDoubleFunction mapper)
* mapToLong(ToLongFunction mapper)

**3. FlatMap**

Maps each element to a stream and flattens the result. Useful for handling nested collections (e.g., List of List).

Methods:

* flatMap​(Function mapper)
* flatMapToInt​(Function mapper)
* flatMap​ToDouble(Function mapper)
* flatMap​ToLong(Function mapper)

**4. Distinct**

Returns distinct elements according to equals() and hashCode().

Methods:

* distinct()

**5. Sorting**

Returns the elements sorted according to natural order or according to the provided Comparator.

Methods:

* sorted()
* sorted​(Comparator comparator)

**6. Limit and Skip**

Limit truncates the stream to the first n elements. Skip discards the first n elements.

Methods:

* limit​(long maxSize)
* skip​(long n)

**Stream Terminal Operations**

**Terminal operations trigger processing of the stream and produce a result or a side-effect.**

**1. ForEach**

Common for printing or performing final side-effects. Does not preserve order in parallel streams.

Methods:

* forEach​(Consumer action)

**2. Reduce**

Combines elements using an associative accumulation function.

Methods:

* reduce​(BinaryOperator accumulator)
* reduce​(T identity, BinaryOperator accumulator)
* reduce​(U identity, BiFunction accumulator, BinaryOperator combiner)

**3. Collect**

Accumulates elements into a summary result (e.g., List, Set, Map).

Methods:

* collect​(Collector collector)
* collect​(Supplier supplier, BiConsumer accumulator, BiConsumer combiner)

**4. Min and Max**

Returns the minimum/maximum element. Return type is Optional<T>.

Methods:

* max​(Comparator comparator)
* min​(Comparator comparator)

**5. Count**

Returns the number of elements in the stream. Returns a long value.

Methods:

* count()

**6. AnyMatch, AllMatch, NoneMatch**

These are short-circuiting operations. They need no process the entire stream to produce a result.

Methods:

* anyMatch​(Predicate predicate)
* allMatch​(Predicate predicate)
* noneMatch​(Predicate predicate)

**7. FindFirst and FindAny**

Returns the first matching or any matching element. Return type is Optional<T>.

Methods:

* findFirst()
* findAny()

**Distinct**

**Q. How does distinct() work internally? Does it rely on equals() and hashCode()?**

Internally, distinct() uses a LinkedHashSet and relies on both equals() and hashCode() methods to determine if two objects are equal.

The hashCode() is used to identify the bucket and equals() is used to check for actual equality in case of hash collisions.

**Q. Does the order of elements matter when using distinct()?**

In ordered streams (like those from a List), the distinct() method preserves the encounter order of the first occurrence of each unique element.

**Aggregate Functions**

**Q. What is the return type of min() and max() methods? Why is it Optional?**

The return type of both min() and max() methods in the Java Stream API is: Optional<T>. Optional<T> provides a safe wrapper that forces you to explicitly handle the absence of a result, preventing NullPointerExceptions.

**Q. What is the difference between count() and Collectors.counting()?**

stream.count() is a terminal operation used directly on a stream. Collectors.counting()is used with collect() as part of a reduction operation, usually during grouping or partitioning.