**Stream**

|  |  |  |
| --- | --- | --- |
| **API** | **Example** | **DSL** |
| **Init:**   * **generate** * **iterate** * **range(startIncl, endExcl)** | Stream<String> streamGenerated = Stream.generate(() -> "a").limit(10);  Stream<Integer> streamIterated = Stream.iterate(40, n -> n + 2).limit(20);  IntStream intStream = IntStream.range(1, 3).limit(10); | stream("a", i->f(i, p))  stream(1, i->i+1).limit(3)  stream(0, i->random(0, 350) |
| **Pipeline (intermediate):**   * **filter (stateless)** * **map/flatMap (stateless)** * **distinct (stateful)** * **sorted (stateful)** * **skip** * **shuffle** | Stream<String> filtered = Stream.of("a", "b", "c").filter(element -> element.contains("b"))  Stream<String> onceModifiedStream = Stream.of("abcd", "bbcd", "cbcd").skip(1);  Stream<String> twiceModifiedStream = onceModifiedStream.map(element -> element.substring(0, 3)); | .filter(**e**, e->f(e, p))  .map(**e**, e->f(e, p)) |
| **Reduction (terminal):**   * **sum** * **max/min** * **count** * **average** * **findFirst/findAny** * **allMatch/anyMatch/noneMatch** * **groupingBy** | **OptionalInt** reduced = IntStream.range(1, 4).reduce((a, b) -> a + b); // reduced = 6 (1 + 2 + 3)  **int** twoParams = IntStream.range(1, 4).reduce(10, (a, b) -> a + b); // twoParams = 16 (10 + 1 + 2 + 3)  **String** reduced = Stream.of("a", "b", "c").collect(Collectors.joining(", ", "prefix", "suffix"));  Map<Boolean, List<String>> mapByBool = strStream.collect(Collectors.groupingBy(e -> e.length() > 1));  Map<Integer, List<String>> mapByInt = strStream.collect(Collectors.groupingBy(e -> e.length())); | .reduce((a, b)->f(a, b))  .reduce(accum, (a, b)->f(a, b))  .groupingBy(e->f(e,p)) |
| **RegEx:**   * **split As Stream** | Stream<String> streamOfString = Pattern.compile(", ").splitAsStream("a, b, c"); |  |

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

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

**Stream operations and pipelines**

Stream operations are divided into **intermediate** and **terminal** operations, and are combined to form stream pipelines. A stream pipeline consists of a source (such as a Collection, an array, a generator function, or an I/O channel); followed by zero or more intermediate operations such as Stream.filter or Stream.map; and a terminal operation such as Stream.forEach or Stream.reduce.

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

Terminal operations, such as Stream.forEach or IntStream.sum, may traverse the stream to produce a result or a side-effect. After the terminal operation is performed, the stream pipeline is considered consumed, and can no longer be used; if you need to traverse the same data source again, you must return to the data source to get a new stream. In almost all cases, terminal operations are eager, completing their traversal of the data source and processing of the pipeline before returning. Only the terminal operations iterator() and spliterator() are not; these are provided as an "escape hatch" to enable arbitrary client-controlled pipeline traversals in the event that the existing operations are not sufficient to the task.

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

Intermediate operations are further divided into **stateless and stateful** operations. Stateless operations, such as filter and map, retain no state from previously seen element when processing a new element -- each element can be processed independently of operations on other elements. Stateful operations, such as distinct and sorted, may incorporate state from previously seen elements when processing new elements.

Stateful operations may need to process the entire input before producing a result. For example, one cannot produce any results from sorting a stream until one has seen all elements of the stream. As a result, under parallel computation, some pipelines containing stateful intermediate operations may require multiple passes on the data or may need to buffer significant data. Pipelines containing exclusively stateless intermediate operations can be processed in a single pass, whether sequential or parallel, with minimal data buffering.

Further, some operations are deemed short-circuiting operations. An intermediate operation is short-circuiting if, when presented with infinite input, it may produce a finite stream as a result. A terminal operation is short-circuiting if, when presented with infinite input, it may terminate in finite time. Having a short-circuiting operation in the pipeline is a necessary, but not sufficient, condition for the processing of an infinite stream to terminate normally in finite time.

**Parallelism**

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

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

The only difference between the serial and parallel versions of this example is the creation of the initial stream, using "parallelStream()" instead of " stream()". The stream pipeline is executed sequentially or in parallel depending on the mode of the stream on which the terminal operation is invoked. The sequential or parallel mode of a stream can be determined with the BaseStream.isParallel() method, and the stream's mode can be modified with the BaseStream.sequential() and BaseStream.parallel() operations. The most recent sequential or parallel mode setting applies to the execution of the entire stream pipeline.

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

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

**Non-interference**

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

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

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

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

**Stateless behaviors**

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

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

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

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

**Side-effects**

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

If the behavioral parameters do have side-effects, unless explicitly stated, there are no guarantees as to:

the visibility of those side-effects to other threads;

that different operations on the "same" element within the same stream pipeline are executed in the same thread; and

that behavioral parameters are always invoked, since a stream implementation is free to elide operations (or entire stages) from a stream pipeline if it can prove that it would not affect the result of the computation.

The ordering of side-effects may be surprising. Even when a pipeline is constrained to produce a result that is consistent with the encounter order of the stream source (for example, IntStream.range(0,5).parallel().map(x -> x\*2).toArray() must produce [0, 2, 4, 6, 8]), no guarantees are made as to the order in which the mapper function is applied to individual elements, or in what thread any behavioral parameter is executed for a given element.

The eliding of side-effects may also be surprising. With the exception of terminal operations forEach and forEachOrdered, side-effects of behavioral parameters may not always be executed when the stream implementation can optimize away the execution of behavioral parameters without affecting the result of the computation. (For a specific example see the API note documented on the count operation.)

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

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

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

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

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

**Ordering**

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

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

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

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

**Reduction operations**

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

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

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

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

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

or:

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

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

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

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

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

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

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

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

Here, the identity element is both an initial seed value for the reduction and a default result if there are no input elements. The accumulator function takes a partial result and the next element, and produces a new partial result. The combiner function combines two partial results to produce a new partial result. (The combiner is necessary in parallel reductions, where the input is partitioned, a partial accumulation computed for each partition, and then the partial results are combined to produce a final result.)

More formally, the identity value must be an identity for the combiner function. This means that for all u, combiner.apply(identity, u) is equal to u. Additionally, the combiner function must be associative and must be compatible with the accumulator function: for all u and t, combiner.apply(u, accumulator.apply(identity, t)) must be equals() to accumulator.apply(u, t).

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

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

though the explicit map-reduce form is more readable and therefore should usually be preferred. The generalized form is provided for cases where significant work can be optimized away by combining mapping and reducing into a single function.

**Mutable reduction**

A mutable reduction operation accumulates input elements into a mutable result container, such as a Collection or StringBuilder, as it processes the elements in the stream.

If we wanted to take a stream of strings and concatenate them into a single long string, we could achieve this with ordinary reduction:

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

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

The mutable reduction operation is called collect(), as it collects together the desired results into a result container such as a Collection. A collect operation requires three functions: a supplier function to construct new instances of the result container, an accumulator function to incorporate an input element into a result container, and a combining function to merge the contents of one result container into another. The form of this is very similar to the general form of ordinary reduction:

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

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

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

Or we could use a parallelizable collect form:

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

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

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

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

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

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

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

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

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

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

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

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

A a1 = supplier.get(); accumulator.accept(a1, t1); accumulator.accept(a1, t2); R r1 = finisher.apply(a1); // result without splitting A a2 = supplier.get(); accumulator.accept(a2, t1); A a3 = supplier.get(); accumulator.accept(a3, t2); R r2 = finisher.apply(combiner.apply(a2, a3)); // result with splitting

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

**Reduction, concurrency, and ordering**

With some complex reduction operations, for example a collect() that produces a Map, such as:

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

it may actually be counterproductive to perform the operation in parallel. This is because the combining step (merging one Map into another by key) can be expensive for some Map implementations.

Suppose, however, that the result container used in this reduction was a concurrently modifiable collection -- such as a java.util.concurrent.ConcurrentHashMap. In that case, the parallel invocations of the accumulator could actually deposit their results concurrently into the same shared result container, eliminating the need for the combiner to merge distinct result containers. This potentially provides a boost to the parallel execution performance. We call this a concurrent reduction.

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

The stream is parallel;

The collector has the Collector.Characteristics.CONCURRENT characteristic, and;

Either the stream is unordered, or the collector has the Collector.Characteristics.UNORDERED characteristic.

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

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

(where Collectors.groupingByConcurrent is the concurrent equivalent of groupingBy).

Note that if it is important that the elements for a given key appear in the order they appear in the source, then we cannot use a concurrent reduction, as ordering is one of the casualties of concurrent insertion. We would then be constrained to implement either a sequential reduction or a merge-based parallel reduction.

**Associativity**

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

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

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

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

So we can evaluate (a op b) in parallel with (c op d), and then invoke op on the results.

Examples of associative operations include numeric addition, min, and max, and string concatenation.

Low-level stream construction

So far, all the stream examples have used methods like java.util.Collection.stream() or java.util.Arrays.stream(Object[]) to obtain a stream. How are those stream-bearing methods implemented?

The class StreamSupport has a number of low-level methods for creating a stream, all using some form of a java.util.Spliterator. A spliterator is the parallel analogue of an java.util.Iterator; it describes a (possibly infinite) collection of elements, with support for sequentially advancing, bulk traversal, and splitting off some portion of the input into another spliterator which can be processed in parallel. At the lowest level, all streams are driven by a spliterator.

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

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

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

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

|  |  |
| --- | --- |
| **Clause** | **Example** |
| [from](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/from-clause) | var q = **from** num in numbers  where num < 5  select num;  var q = **from** Student student in arrList where student.Scores[0] > 95  select student;  List<Student> students = […];  var q = **from** student in students  from score in student.Scores where score > 90  select new { Last = student.LastName, score }; |
| [where](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/where-clause) | var queryLowNums2 = from num in numbers  **where** num < 5 && num % 2 == 0  select num;  var queryLowNums3 = from num in numbers  **where** num < 5  **where** num % 2 == 0  select num; |
| [group](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/group-clause) | var studentQuery = from student in students  let avg = (int)student.Scores.Average()  **group** student by (avg / 10) into g  orderby g.Key  select g; |
| [into](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/into) | var wordGroups1 = from w in words  group w by w[0] **into** fruitGroup  where fruitGroup.Count() >= 2  select new { FirstLetter = fruitGroup.Key, Words = fruitGroup.Count() }; |
| [join](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/join-clause) | var innerJoinQuery = from category in categories  **join** prod in products on category.ID equals prod.CategoryID  select new { ProductName = prod.Name, Category = category.Name }; ;produces flat sequence  var innerGroupJoinQuery = from category in categories  join prod in products on category.ID equals prod.CategoryID into prodGroup  select new { CategoryName = category.Name, Products = prodGroup }; |
| [let](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/let-clause) | var earlyBirdQuery = from sentence in strings  **let** words = sentence.Split(' ')  from word in words let w = word.ToLower()  where w[0] == 'a' || w[0] == 'e' || w[0] == 'i'  select word; |

**Knapsack(w:int[], v:int[], N:int, c:int) -> (int[], int)**

from int[] vector where ALL(FROM vector.element WHERE 0 == vector.element || vector.element == 1)

**query\_expression:** from\_clause query\_body;

**from\_clause:** 'from' type? identifier 'in' expression;

**query\_body:** query\_body\_clauses? select\_or\_group\_clause query\_continuation?;

**query\_body\_clauses:** query\_body\_clause | query\_body\_clauses query\_body\_clause;

**query\_body\_clause:** from\_clause | let\_clause | where\_clause | join\_clause | join\_into\_clause | orderby\_clause;

**let\_clause:** 'let' identifier '=' expression;

**where\_clause:** 'where' boolean\_expression;

**join\_clause:** 'join' type? identifier 'in' expression 'on' expression 'equals' expression;

**join\_into\_clause:** 'join' type? identifier 'in' expression 'on' expression 'equals' expression 'into' identifier;

**orderby\_clause:** 'orderby' orderings;

**orderings:** ordering (',' ordering)\*;

**ordering:** expression ordering\_direction?;

**ordering\_direction:** 'ascending' | 'descending';

**select\_or\_group\_clause:** select\_clause | group\_clause;

**select\_clause:** 'select' expression;

**group\_clause:** 'group' expression 'by' expression;

**query\_continuation:** 'into' identifier query\_body;

|  |  |
| --- | --- |
| [Aggregate](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.aggregate?view=net-8.0#system-linq-enumerable-aggregate-3(system-collections-generic-ienumerable((-0))-1-system-func((-1-0-1))-system-func((-1-2))))(accum, f\_accum, f\_result) | Applies an accumulator function over a sequence. The specified seed value is used as the initial accumulator value, and the specified function is used to select the result value. |
| [All](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.all?view=net-8.0#system-linq-enumerable-all-1(system-collections-generic-ienumerable((-0))-system-func((-0-system-boolean))))(p) | Determines whether all elements of a sequence satisfy a condition. |
| [Any](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.any?view=net-8.0#system-linq-enumerable-any-1(system-collections-generic-ienumerable((-0)))) | Determines whether a sequence contains any elements. |
| [Any](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.any?view=net-8.0#system-linq-enumerable-any-1(system-collections-generic-ienumerable((-0))-system-func((-0-system-boolean))))(p) | Determines whether any element of a sequence satisfies a condition. |
| [Average](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.average?view=net-8.0#system-linq-enumerable-average(system-collections-generic-ienumerable((system-decimal)))) | Computes the average of a sequence of [Decimal](https://learn.microsoft.com/en-us/dotnet/api/system.decimal?view=net-8.0) values. |
| [Average](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.average?view=net-8.0#system-linq-enumerable-average-1(system-collections-generic-ienumerable((-0))-system-func((-0-system-decimal))))(f) | Computes the average of a sequence of [Decimal](https://learn.microsoft.com/en-us/dotnet/api/system.decimal?view=net-8.0) values that are obtained by invoking a transform function on each element of the input sequence. |
| [Chunk](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.chunk?view=net-8.0#system-linq-enumerable-chunk-1(system-collections-generic-ienumerable((-0))-system-int32)) | Splits the elements of a sequence into chunks of size at most size. |
| [Concat](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.concat?view=net-8.0#system-linq-enumerable-concat-1(system-collections-generic-ienumerable((-0))-system-collections-generic-ienumerable((-0)))) | Concatenates two sequences. |
| [Contains](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.contains?view=net-8.0#system-linq-enumerable-contains-1(system-collections-generic-ienumerable((-0))-0)) | Determines whether a sequence contains a specified element by using the default equality comparer. |
| [Count(](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.count?view=net-8.0#system-linq-enumerable-count-1(system-collections-generic-ienumerable((-0))-system-func((-0-system-boolean))))f) | Returns a number that represents how many elements in the specified sequence satisfy a condition. |
| [DefaultIfEmpty](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.defaultifempty?view=net-8.0#system-linq-enumerable-defaultifempty-1(system-collections-generic-ienumerable((-0))-0))(const) | Returns the elements of the specified sequence or the specified value in a singleton collection if the sequence is empty. |
| [Distinct](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.distinct?view=net-8.0#system-linq-enumerable-distinct-1(system-collections-generic-ienumerable((-0)))) | Returns distinct elements from a sequence by using the default equality comparer to compare values. |
| [DistinctBy(f)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.distinctby?view=net-8.0#system-linq-enumerable-distinctby-2(system-collections-generic-ienumerable((-0))-system-func((-0-1)))) | Returns distinct elements from a sequence according to a specified key selector function. |
| [ElementAt(index)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.elementat?view=net-8.0#system-linq-enumerable-elementat-1(system-collections-generic-ienumerable((-0))-system-index)) | Returns the element at a specified index in a sequence. |
| [ElementAtOrDefault(index)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.elementatordefault?view=net-8.0#system-linq-enumerable-elementatordefault-1(system-collections-generic-ienumerable((-0))-system-index)) | Returns the element at a specified index in a sequence or a default value if the index is out of range. |
| [Empty()](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.empty?view=net-8.0#system-linq-enumerable-empty-1) | Returns an empty [IEnumerable](https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.ienumerable-1?view=net-8.0) that has the specified type argument. |
| [Except](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.except?view=net-8.0#system-linq-enumerable-except-1(system-collections-generic-ienumerable((-0))-system-collections-generic-ienumerable((-0)))) | Produces the set difference of two sequences by using the default equality comparer to compare values. |
| [ExceptBy(f)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.exceptby?view=net-8.0#system-linq-enumerable-exceptby-2(system-collections-generic-ienumerable((-0))-system-collections-generic-ienumerable((-1))-system-func((-0-1)))) | Produces the set difference of two sequences according to a specified key selector function. |
| [First(p)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.first?view=net-8.0#system-linq-enumerable-first-1(system-collections-generic-ienumerable((-0))-system-func((-0-system-boolean)))) | Returns the first element in a sequence that satisfies a specified condition. |
| [FirstOrDefault(P, Const)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.firstordefault?view=net-8.0#system-linq-enumerable-firstordefault-1(system-collections-generic-ienumerable((-0))-system-func((-0-system-boolean))-0)) | Returns the first element of the sequence that satisfies a condition, or a specified default value if no such element is found. |
| [GroupBy(Func, Func, Func<TKey,IEnumerable<TElement>,TResult>)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.groupby?view=net-8.0#system-linq-enumerable-groupby-4(system-collections-generic-ienumerable((-0))-system-func((-0-1))-system-func((-0-2))-system-func((-1-system-collections-generic-ienumerable((-2))-3)))) | Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. The elements of each group are projected by using a specified function. |
| [GroupBy(Func, Func)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.groupby?view=net-8.0#system-linq-enumerable-groupby-3(system-collections-generic-ienumerable((-0))-system-func((-0-1))-system-func((-0-2)))) | Groups the elements of a sequence according to a specified key selector function and projects the elements for each group by using a specified function. |
| [GroupBy(Func, Func<TKey,IEnumerable,TResult>)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.groupby?view=net-8.0#system-linq-enumerable-groupby-3(system-collections-generic-ienumerable((-0))-system-func((-0-1))-system-func((-1-system-collections-generic-ienumerable((-0))-2)))) | Groups the elements of a sequence according to a specified key selector function and creates a result value from each group and its key. |
| [GroupBy(Func)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.groupby?view=net-8.0#system-linq-enumerable-groupby-2(system-collections-generic-ienumerable((-0))-system-func((-0-1)))) | Groups the elements of a sequence according to a specified key selector function. |
| [GroupJoin(IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.groupjoin?view=net-8.0#system-linq-enumerable-groupjoin-4(system-collections-generic-ienumerable((-0))-system-collections-generic-ienumerable((-1))-system-func((-0-2))-system-func((-1-2))-system-func((-0-system-collections-generic-ienumerable((-1))-3)))) | Correlates the elements of two sequences based on equality of keys and groups the results. The default equality comparer is used to compare keys. |
| [Intersect(IEnumerable)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.intersect?view=net-8.0#system-linq-enumerable-intersect-1(system-collections-generic-ienumerable((-0))-system-collections-generic-ienumerable((-0)))) | Produces the set intersection of two sequences by using the default equality comparer to compare values. |
| [IntersectBy(IEnumerable<TKey>, Func)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.intersectby?view=net-8.0#system-linq-enumerable-intersectby-2(system-collections-generic-ienumerable((-0))-system-collections-generic-ienumerable((-1))-system-func((-0-1)))) | Produces the set intersection of two sequences according to a specified key selector function. |
| [Join(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.join?view=net-8.0#system-linq-enumerable-join-4(system-collections-generic-ienumerable((-0))-system-collections-generic-ienumerable((-1))-system-func((-0-2))-system-func((-1-2))-system-func((-0-1-3)))) | Correlates the elements of two sequences based on matching keys. The default equality comparer is used to compare keys. |
| [Last(P)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.last?view=net-8.0#system-linq-enumerable-last-1(system-collections-generic-ienumerable((-0))-system-func((-0-system-boolean)))) | Returns the last element of a sequence that satisfies a specified condition. |
| [LastOrDefault(P, Const)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.lastordefault?view=net-8.0#system-linq-enumerable-lastordefault-1(system-collections-generic-ienumerable((-0))-system-func((-0-system-boolean))-0)) | Returns the last element of a sequence that satisfies a condition, or a specified default value if no such element is found. |
| [Max(Func)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.max?view=net-8.0#system-linq-enumerable-max-2(system-collections-generic-ienumerable((-0))-system-func((-0-1)))) | Invokes a transform function on each element of a generic sequence and returns the maximum resulting value. |
| [Max()](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.max?view=net-8.0#system-linq-enumerable-max-1(system-collections-generic-ienumerable((-0)))) | Returns the maximum value in a generic sequence. |
| [MaxBy(Func)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.maxby?view=net-8.0#system-linq-enumerable-maxby-2(system-collections-generic-ienumerable((-0))-system-func((-0-1)))) | Returns the maximum value in a generic sequence according to a specified key selector function. |
| [OfType<TResult>(IEnumerable)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.oftype?view=net-8.0#system-linq-enumerable-oftype-1(system-collections-ienumerable)) | Filters the elements of an [IEnumerable](https://learn.microsoft.com/en-us/dotnet/api/system.collections.ienumerable?view=net-8.0) based on a specified type. |
| [Order(IEnumerable)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.order?view=net-8.0#system-linq-enumerable-order-1(system-collections-generic-ienumerable((-0)))) | Sorts the elements of a sequence in ascending order. |
| [OrderBy(Func)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.orderby?view=net-8.0#system-linq-enumerable-orderby-2(system-collections-generic-ienumerable((-0))-system-func((-0-1)))) | Sorts the elements of a sequence in ascending order according to a key. |
| [OrderByDescending(Func)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.orderbydescending?view=net-8.0#system-linq-enumerable-orderbydescending-2(system-collections-generic-ienumerable((-0))-system-func((-0-1)))) | Sorts the elements of a sequence in descending order according to a key. |
| [OrderDescending(IEnumerable)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.orderdescending?view=net-8.0#system-linq-enumerable-orderdescending-1(system-collections-generic-ienumerable((-0)))) | Sorts the elements of a sequence in descending order. |
| [Prepend(Const)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.prepend?view=net-8.0#system-linq-enumerable-prepend-1(system-collections-generic-ienumerable((-0))-0)) | Adds a value to the beginning of the sequence. |
| [Range(Int32, Int32)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.range?view=net-8.0#system-linq-enumerable-range(system-int32-system-int32)) | Generates a sequence of integral numbers within a specified range. |
| [Repeat(TResult, Int32)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.repeat?view=net-8.0#system-linq-enumerable-repeat-1(-0-system-int32)) | Generates a sequence that contains one repeated value. |
| [Reverse(IEnumerable)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.reverse?view=net-8.0#system-linq-enumerable-reverse-1(system-collections-generic-ienumerable((-0)))) | Inverts the order of the elements in a sequence. |
| [Select(Func<Const,Int32,TResult>)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.select?view=net-8.0#system-linq-enumerable-select-2(system-collections-generic-ienumerable((-0))-system-func((-0-system-int32-1)))) | Projects each element of a sequence into a new form by incorporating the element's index. |
| [SelectMany(Func<Const,Int32,IEnumerable<TCollection>>, Func<Const,TCollection,TResult>)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.selectmany?view=net-8.0#system-linq-enumerable-selectmany-3(system-collections-generic-ienumerable((-0))-system-func((-0-system-int32-system-collections-generic-ienumerable((-1))))-system-func((-0-1-2)))) | Projects each element of a sequence to an [IEnumerable](https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.ienumerable-1?view=net-8.0), flattens the resulting sequences into one sequence, and invokes a result selector function on each element therein. The index of each source element is used in the intermediate projected form of that element. |
| [SelectMany(Func<Const,IEnumerable<TResult>>)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.selectmany?view=net-8.0#system-linq-enumerable-selectmany-2(system-collections-generic-ienumerable((-0))-system-func((-0-system-collections-generic-ienumerable((-1)))))) | Projects each element of a sequence to an [IEnumerable](https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.ienumerable-1?view=net-8.0) and flattens the resulting sequences into one sequence. |
| [SelectMany(Func<Const,Int32,IEnumerable<TResult>>)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.selectmany?view=net-8.0#system-linq-enumerable-selectmany-2(system-collections-generic-ienumerable((-0))-system-func((-0-system-int32-system-collections-generic-ienumerable((-1)))))) | Projects each element of a sequence to an [IEnumerable](https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.ienumerable-1?view=net-8.0), and flattens the resulting sequences into one sequence. The index of each source element is used in the projected form of that element. |
| [SequenceEqual(IEnumerable)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.sequenceequal?view=net-8.0#system-linq-enumerable-sequenceequal-1(system-collections-generic-ienumerable((-0))-system-collections-generic-ienumerable((-0)))) | Determines whether two sequences are equal by comparing the elements by using the default equality comparer for their type. |
| [Single(P)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.single?view=net-8.0#system-linq-enumerable-single-1(system-collections-generic-ienumerable((-0))-system-func((-0-system-boolean)))) | Returns the only element of a sequence that satisfies a specified condition, and throws an exception if more than one such element exists. |
| [SingleOrDefault(P)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.singleordefault?view=net-8.0#system-linq-enumerable-singleordefault-1(system-collections-generic-ienumerable((-0))-system-func((-0-system-boolean)))) | Returns the only element of a sequence that satisfies a specified condition or a default value if no such element exists; this method throws an exception if more than one element satisfies the condition. |
| [SingleOrDefault(P, Const)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.singleordefault?view=net-8.0#system-linq-enumerable-singleordefault-1(system-collections-generic-ienumerable((-0))-system-func((-0-system-boolean))-0)) | Returns the only element of a sequence that satisfies a specified condition, or a specified default value if no such element exists; this method throws an exception if more than one element satisfies the condition. |
| [Skip(Int32)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.skip?view=net-8.0#system-linq-enumerable-skip-1(system-collections-generic-ienumerable((-0))-system-int32)) | Bypasses a specified number of elements in a sequence and then returns the remaining elements. |
| [SkipLast(Int32)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.skiplast?view=net-8.0#system-linq-enumerable-skiplast-1(system-collections-generic-ienumerable((-0))-system-int32)) | Returns a new enumerable collection that contains the elements from source with the last count elements of the source collection omitted. |
| [SkipWhile(P)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.skipwhile?view=net-8.0#system-linq-enumerable-skipwhile-1(system-collections-generic-ienumerable((-0))-system-func((-0-system-boolean)))) | Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements. |
| [SkipWhile(Func<Const,Int32,Boolean>)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.skipwhile?view=net-8.0#system-linq-enumerable-skipwhile-1(system-collections-generic-ienumerable((-0))-system-func((-0-system-int32-system-boolean)))) | Bypasses elements in a sequence as long as a specified condition is true and then returns the remaining elements. The element's index is used in the logic of the predicate function. |
| [Sum()](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.sum?view=net-8.0#system-linq-enumerable-sum(system-collections-generic-ienumerable((system-decimal)))) | Computes the sum of a sequence of [Decimal](https://learn.microsoft.com/en-us/dotnet/api/system.decimal?view=net-8.0) values. |
| [Sum(Func<Const,Decimal>)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.sum?view=net-8.0#system-linq-enumerable-sum-1(system-collections-generic-ienumerable((-0))-system-func((-0-system-decimal)))) | Computes the sum of the sequence of [Decimal](https://learn.microsoft.com/en-us/dotnet/api/system.decimal?view=net-8.0) values that are obtained by invoking a transform function on each element of the input sequence. |
| [Take(Int32)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.take?view=net-8.0#system-linq-enumerable-take-1(system-collections-generic-ienumerable((-0))-system-int32)) | Returns a specified number of contiguous elements from the start of a sequence. |
| [Take(Range)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.take?view=net-8.0#system-linq-enumerable-take-1(system-collections-generic-ienumerable((-0))-system-range)) | Returns a specified range of contiguous elements from a sequence. |
| [TakeLast(Int32)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.takelast?view=net-8.0#system-linq-enumerable-takelast-1(system-collections-generic-ienumerable((-0))-system-int32)) | Returns a new enumerable collection that contains the last count elements from source. |
| [TakeWhile(P)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.takewhile?view=net-8.0#system-linq-enumerable-takewhile-1(system-collections-generic-ienumerable((-0))-system-func((-0-system-boolean)))) | Returns elements from a sequence as long as a specified condition is true. |
| [TakeWhile(Func<Const,Int32,Boolean>)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.takewhile?view=net-8.0#system-linq-enumerable-takewhile-1(system-collections-generic-ienumerable((-0))-system-func((-0-system-int32-system-boolean)))) | Returns elements from a sequence as long as a specified condition is true. The element's index is used in the logic of the predicate function. |
| [ThenBy(Func)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.thenby?view=net-8.0#system-linq-enumerable-thenby-2(system-linq-iorderedenumerable((-0))-system-func((-0-1)))) | Performs a subsequent ordering of the elements in a sequence in ascending order according to a key. |
| [ThenByDescending(Func)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.thenbydescending?view=net-8.0#system-linq-enumerable-thenbydescending-2(system-linq-iorderedenumerable((-0))-system-func((-0-1)))) | Performs a subsequent ordering of the elements in a sequence in descending order, according to a key. |
| [TryGetNonEnumeratedCount(Int32)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.trygetnonenumeratedcount?view=net-8.0#system-linq-enumerable-trygetnonenumeratedcount-1(system-collections-generic-ienumerable((-0))-system-int32@)) | Attempts to determine the number of elements in a sequence without forcing an enumeration. |
| [Union()](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.union?view=net-8.0#system-linq-enumerable-union-1(system-collections-generic-ienumerable((-0))-system-collections-generic-ienumerable((-0)))) | Produces the set union of two sequences by using the default equality comparer. |
| [UnionBy(Func)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.unionby?view=net-8.0#system-linq-enumerable-unionby-2(system-collections-generic-ienumerable((-0))-system-collections-generic-ienumerable((-0))-system-func((-0-1)))) | Produces the set union of two sequences according to a specified key selector function. |
| [Where(Func<Const,Int32,Boolean>)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.where?view=net-8.0#system-linq-enumerable-where-1(system-collections-generic-ienumerable((-0))-system-func((-0-system-int32-system-boolean)))) | Filters a sequence of values based on a predicate. Each element's index is used in the logic of the predicate function. |
| [Zip(IEnumerable<TSecond>, Func<TFirst,TSecond,TResult>)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.zip?view=net-8.0#system-linq-enumerable-zip-3(system-collections-generic-ienumerable((-0))-system-collections-generic-ienumerable((-1))-system-func((-0-1-2)))) | Applies a specified function to the corresponding elements of two sequences, producing a sequence of the results. |
| [Zip(IEnumerable<TSecond>)](https://learn.microsoft.com/en-us/dotnet/api/system.linq.enumerable.zip?view=net-8.0#system-linq-enumerable-zip-2(system-collections-generic-ienumerable((-0))-system-collections-generic-ienumerable((-1)))) | Produces a sequence of tuples with elements from the two specified sequences. |