Collectors

The ready-made collect() recipes the Collectors class offers: toList()/toSet(), joining(), groupingBy() (with counting()/mapping() as downstream collectors), partitioningBy(), toMap(). The three parts of a Collector: supplier, accumulator, combiner.

Intermediate 22 min

Last updated: October 19, 2026
Read online: https://www.learnforgex.com/en/topics/collectors

---

Collectors

In the "Terminal Operations" lesson, you used `Stream.toList()` to gather a stream into a list. `collect(Collectors.toList())` serves the same purpose, but it isn't an exact equivalent: `Stream.toList()` returns an unmodifiable list, while `Collectors.toList()` makes no guarantee about the returned list's type or mutability -- if you need a modifiable `ArrayList`, use `Collectors.toCollection(ArrayList::new)`. That lesson also never explored `collect()`'s real power. This lesson picks up exactly there: the ready-made recipes the `Collectors` class offers for grouping, joining, and transforming.

What Is Collectors?

`Collectors` is a utility class in the `java.util.stream` package; it produces ready-made `Collector` objects to hand to the `collect()` terminal operation. `collect()` itself is general-purpose -- it says "gather these elements into a result," but leaves **how** up to the `Collector`. Static methods like `Collectors.toList()` and `Collectors.groupingBy()` provide ready answers to the most common "how"s.

Why Does It Exist?

Turning a stream into a simple list is easy with `toList()` (the "Terminal Operations" lesson), but needs like "group elements by a property", "join elements into a single string", and "split elements into two groups" are far more common and require quite a bit of code with a hand-written `for` loop. `Collectors` reduces these common patterns to a single, named call -- `groupingBy(...)` tells the reader directly "this is a grouping operation", not loop mechanics.

History

The `Collectors` class arrived alongside the Stream API in Java 8 (2014). `collect()` itself was introduced at the same time; the two were designed together, since `collect()`'s signature takes a `Collector<T,A,R>` parameter directly -- without the `Collectors` class, every developer would have to write their own `Collector` from scratch.

The Three Parts of collect(): Supplier, Accumulator, Combiner

A `Collector` is made up of three functions: a **supplier** (creates an empty container to hold the result, like an empty `ArrayList`), an **accumulator** (adds each element into that container), and a **combiner** (merges partial results for parallel streams). Every static method in the `Collectors` class sets this trio up for you -- you'll rarely need to write your own `Collector` from scratch.

Collectors.toList() and Collectors.toSet(): Simple Collections

`Collectors.toList()` is very similar to `Stream.toList()` from the "Terminal Operations" lesson, but the two make different promises about the result: `Stream.toList()` (Java 16+) is guaranteed to return an **unmodifiable** list, while `collect(Collectors.toList())` makes **no guarantee** about the type or mutability of the list it returns (the current JDK happens to return an `ArrayList`, but that's an implementation detail, not part of the contract). If you need a list you can modify, ask for one explicitly with `collect(Collectors.toCollection(ArrayList::new))`. `Collectors.toSet()` gathers elements into a `Set` -- duplicates are removed automatically, but there's no ordering guarantee.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Set;
import java.util.stream.Collectors;

// Collectors.toList()/toSet() are the general-purpose way to collect a stream into a
// collection. Unlike Stream.toList() (always unmodifiable), Collectors.toList() makes
// no guarantee about the returned List's type or mutability.
class ToListToSetExample {
    public static void main(String[] args) {
        List<String> names = List.of("Ahmet", "Mehmet", "Ayse", "Ahmet");

        // Fine for just gathering elements -- we only read the result.
        List<String> upper = names.stream()
                .map(String::toUpperCase)
                .collect(Collectors.toList());
        System.out.println(upper);

        // Need to modify the result? Ask for a specific, mutable collection.
        List<String> mutableList = names.stream()
                .map(String::toUpperCase)
                .collect(Collectors.toCollection(ArrayList::new));
        mutableList.add("EXTRA");
        System.out.println(mutableList);

        // toSet() removes duplicates the way any Set does -- no guaranteed order.
        Set<String> unique = names.stream().collect(Collectors.toSet());
        System.out.println(unique.size());
    }
}
```

Collectors.joining(): Joining Strings

`Collectors.joining()` joins a stream of `String`s into a single `String` -- replacing a hand-written `StringBuilder` loop. It has three overloads: no arguments (plain concatenation), a delimiter, and a delimiter together with a prefix and suffix.

```
import java.util.List;
import java.util.stream.Collectors;

// Collectors.joining() concatenates a stream of Strings -- the collect()-based
// alternative to a manual StringBuilder loop, with three overloads: no arguments
// (plain concatenation), a delimiter, and a delimiter with a prefix and suffix.
class JoiningExample {
    public static void main(String[] args) {
        List<String> names = List.of("Ahmet", "Mehmet", "Ayse");

        String plain = names.stream().collect(Collectors.joining());
        System.out.println(plain);

        String withComma = names.stream().collect(Collectors.joining(", "));
        System.out.println(withComma);

        String withBrackets = names.stream()
                .collect(Collectors.joining(", ", "[", "]"));
        System.out.println(withBrackets);
    }
}
```

Expected Output

```
AhmetMehmetAyse
Ahmet, Mehmet, Ayse
[Ahmet, Mehmet, Ayse]
```

Collectors.groupingBy(): Grouping Elements

`Collectors.groupingBy(classifier)` derives a key from each element with a `Function`, then groups elements by that key; the result is a `Map<K, List<T>>` -- each distinct key maps to the list of every element that produced it.

```
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

// groupingBy(classifier) groups elements by a key derived from a Function, producing a
// Map<K, List<T>> -- each distinct key maps to a list of every element that produced it.
class GroupingByExample {
    public static void main(String[] args) {
        List<String> names = List.of("Ahmet", "Ali", "Mehmet", "Ayse", "Ata");

        Map<Character, List<String>> byFirstLetter = names.stream()
                .collect(Collectors.groupingBy(name -> name.charAt(0)));
        System.out.println(byFirstLetter);

        Map<Integer, List<String>> byLength = names.stream()
                .collect(Collectors.groupingBy(String::length));
        System.out.println(byLength);
    }
}
```

Downstream Collectors: counting() and mapping()

`groupingBy()` accepts a second parameter, a **downstream collector**: by default each group is collected into a list, but when a downstream collector is given, it decides what happens to each group's elements instead. `Collectors.counting()` reduces each group directly to its size (producing a `Map<K, Long>`). `Collectors.mapping()` lets you transform each element **before** it's grouped.

```
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

// groupingBy() accepts a second, "downstream" Collector -- instead of collecting each
// group into a List (the default), the downstream Collector decides what happens to
// each group's elements. counting() reduces each group to its size; mapping() lets you
// transform each element before it's collected into the group.
class GroupingByDownstreamExample {
    public static void main(String[] args) {
        List<String> names = List.of("Ahmet", "Ali", "Mehmet", "Ayse", "Ata");

        Map<Character, Long> countByFirstLetter = names.stream()
                .collect(Collectors.groupingBy(name -> name.charAt(0), Collectors.
counting()));
        System.out.println(countByFirstLetter);

        // mapping(): transform each element (here, to its length) BEFORE grouping it.
        Map<Character, List<Integer>> lengthsByFirstLetter = names.stream()
                .collect(Collectors.groupingBy(
                        name -> name.charAt(0),
                        Collectors.mapping(String::length, Collectors.toList())));
        System.out.println(lengthsByFirstLetter);
    }
}
```

Collectors.partitioningBy(): Splitting Into Two

`Collectors.partitioningBy(predicate)` is a special case of `groupingBy()`: it splits elements into exactly **two** groups (`true` / `false`) based on a `Predicate`. Unlike `groupingBy()`, both keys are always present in the resulting `Map` -- even if one group is empty, that key still appears in the map with an empty list.

```
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

// partitioningBy(predicate) is a special case of grouping into EXACTLY two groups --
// true and false -- based on a Predicate. Unlike groupingBy(), both keys always exist
// in the result map, even if one group is empty.
class PartitioningByExample {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8);

        Map<Boolean, List<Integer>> evenOrOdd = numbers.stream()
                .collect(Collectors.partitioningBy(n -> n % 2 == 0));
        System.out.println(evenOrOdd);
        System.out.println(evenOrOdd.get(true));
        System.out.println(evenOrOdd.get(false));

        // Even with no matches for one side, both keys are still present -- unlike
        // groupingBy(), which would simply omit a key with no elements.
        Map<Boolean, List<Integer>> allEven = List.of(2, 4, 6).stream()
                .collect(Collectors.partitioningBy(n -> n % 2 == 0));
        System.out.println(allEven.get(false));
    }
}
```

Collectors.toMap(): Building a Key-Value Mapping

`Collectors.toMap(keyMapper, valueMapper)` builds a `Map` from a stream. Its sharpest edge: if two different elements produce the **same key**, it throws `IllegalStateException` by default -- there's no automatic "last one wins" behavior like some other languages' equivalents. A third argument, a `BinaryOperator<V>`, lets you spell out exactly how to resolve that collision.

```
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

// toMap(keyMapper, valueMapper) builds a Map from a stream. Its sharpest edge: if two
// elements produce the SAME key, it throws IllegalStateException by default -- there's
// no automatic "last one wins" behavior like some other languages' equivalents.
class ToMapExample {
    public static void main(String[] args) {
        List<String> names = List.of("Ahmet", "Mehmet", "Ayse");

        Map<String, Integer> nameToLength = names.stream()
                .collect(Collectors.toMap(name -> name, String::length));
        System.out.println(nameToLength.get("Ahmet"));

        // Duplicate keys with the two-argument form throw IllegalStateException.
        List<String> withDuplicateFirstLetter = List.of("Ahmet", "Ali");
        try {
            withDuplicateFirstLetter.stream()
                    .collect(Collectors.toMap(name -> name.charAt(0), name -> name));
        } catch (IllegalStateException e) {
            System.out.println("caught: duplicate key");
        }

        // A third argument, a BinaryOperator, resolves the collision explicitly --
        // here, keep the longer name for each starting letter.
        Map<Character, String> longestByFirstLetter = withDuplicateFirstLetter.stream()
                .collect(Collectors.toMap(
                        name -> name.charAt(0),
                        name -> name,
                        (existing, incoming) -> existing.length() >= incoming.length() ?
existing : incoming));
        System.out.println(longestByFirstLetter);
    }
}
```

Best Practices

- **Choose deliberately between `collect(Collectors.toList())` and `Stream.toList()`.** If you need to add or remove elements from the result, use neither -- use `collect(Collectors.toCollection(ArrayList::new))`, which guarantees a mutable list; if you're only going to read it, use `Stream.toList()` (unmodifiable, and communicates intent more clearly).
- **Reach for the three-argument form of `toMap()` up front whenever key collisions are possible** -- the two-argument form can produce an unexpected `IllegalStateException` in production.
- **Prefer a `groupingBy()` + downstream collector chain over a hand-written nested loop** -- a single line like `groupingBy(classifier, counting())` is less error-prone than a `Map<K, List<T>>` plus a separate counting loop doing the same job.
- **Use `partitioningBy()` only when there are genuinely two groups** -- for more than two categories, `groupingBy()` is the right tool.

Common Mistakes

- **Using `Collectors.toMap()` with colliding keys and no merge function.** Assuming the data will always produce unique keys leads to an unexpected `IllegalStateException` when production data doesn't cooperate.
- **Relying on the list `collect(Collectors.toList())` returns being mutable.** Only `Stream.toList()`'s behavior is guaranteed (unmodifiable); `Collectors.toList()` promises nothing about mutability, so code that calls `add()` on its result depends on an implementation detail -- use `Collectors.toCollection(ArrayList::new)` when you need to modify the list.
- **Assuming every key will exist in a `groupingBy()` result.** Only `partitioningBy()` guarantees that; in `groupingBy()`, a key with no matching elements simply doesn't appear in the map at all.
- **Trying to use `joining()` on a stream that isn't a stream of Strings.** `Collectors.joining()` only works on a `Stream<String>`; for any other type, you need `map(Object::toString)` first.

Summary, Cheat Sheet, and Glossary

`Collectors` provides ready-made recipes for the `collect()` terminal operation: `toList()`/`toSet()` gather into simple collections (`toList()` doesn't guarantee mutability, while `Stream.toList()` is always unmodifiable -- use `toCollection(ArrayList::new)` for a guaranteed mutable list), `joining()` concatenates strings, `groupingBy()` groups by a key (optionally with a downstream collector like `counting()`/`mapping()`), `partitioningBy()` splits into exactly two groups, and `toMap()` builds a `Map` (which may need an explicit merge function for colliding keys).

Quick reference:

```
stream.collect(Collectors.toList())               // List, mutability not guaranteed
stream.collect(Collectors.toSet())                  // Set, no duplicates
stream.collect(Collectors.joining(", "))              // single String
stream.collect(Collectors.groupingBy(fn))               // Map<K, List<T>>
stream.collect(Collectors.groupingBy(fn, counting()))     // Map<K, Long>
stream.collect(Collectors.partitioningBy(pred))             // Map<Boolean, List<T>>
stream.collect(Collectors.toMap(keyFn, valFn))                 // Map<K, V>
```

Glossary

Collector — The object passed to `collect()` that defines how a stream is gathered into a result; made up of a supplier, an accumulator, and a combiner.

Collectors — The utility class that provides ready-made static methods for producing `Collector` objects.

Downstream collector — A second `Collector` passed to `groupingBy()` / `partitioningBy()` that determines what happens to each group's elements.

groupingBy — A collector that groups elements by a key, producing a `Map<K, List<T>>` (or a different value type, depending on the downstream collector).

partitioningBy — A special case of `groupingBy()` that splits elements into exactly two groups (`true` / `false`) based on a `Predicate` .