Collectors

Collectors sınıfının sunduğu hazır collect() tarifleri: toList()/toSet(), joining(), groupingBy() (+ downstream collector olarak counting()/mapping()), partitioningBy(), toMap(). Bir Collector'ın üç bileşeni: supplier, accumulator, combiner.

Orta 22 dk

Son güncelleme: 19 Ekim 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/collectors

---

Collectors

"Terminal Operations" dersinde bir stream'i listeye toplamak için `Stream.toList()`'i kullanmıştınız. `collect(Collectors.toList())` aynı amaca hizmet eder, ama birebir eşdeğeri değildir: `Stream.toList()` değiştirilemez (unmodifiable) bir liste döndürür, `Collectors.toList()` ise döndürdüğü listenin türü veya değiştirilebilirliği hakkında hiçbir garanti vermez -- değiştirilebilir bir `ArrayList` gerekiyorsa `Collectors.toCollection(ArrayList::new)` kullanın. O derste `collect()`'in asıl gücüne de hiç değinilmemişti. Bu ders tam olarak orada kalıyor: `Collectors` sınıfının sunduğu, gruplama, birleştirme ve dönüştürme için hazır tarifler.

Collectors Nedir?

`Collectors`, `java.util.stream` paketindeki bir yardımcı (utility) sınıftır; `collect()` terminal operation'ına verilecek hazır `Collector` nesneleri üretir. `collect()`'in kendisi genel amaçlıdır -- "bu elemanları bir sonuca topla" der, ama **nasıl** toplanacağını `Collector`'a bırakır. `Collectors.toList()`, `Collectors.groupingBy()` gibi statik metotlar, en sık ihtiyaç duyulan "nasıl"ları hazır sağlar.

Neden Var?

Bir stream'i basit bir listeye çevirmek `toList()` ile kolaydır (Terminal Operations dersi), ama "elemanları bir özelliğe göre grupla", "elemanları tek bir string'de birleştir", "elemanları iki gruba ayır" gibi ihtiyaçlar çok daha yaygındır ve elle yazılan bir `for` döngüsüyle epey kod gerektirir. `Collectors`, bu yaygın desenleri tek satırlık, isimlendirilmiş çağrılara indirger -- `groupingBy(...)` okuyan kişiye doğrudan "bu bir gruplama işlemi" der, döngü mantığını değil.

Tarihçe

`Collectors` sınıfı, Stream API ile birlikte Java 8'de (2014) geldi. `collect()`'in kendisi de aynı zamanda tanıtıldı; ikisi birlikte tasarlandı çünkü `collect()`'in imzası doğrudan bir `Collector<T,A,R>` parametresi alır -- `Collectors` sınıfı olmadan her geliştirici kendi `Collector`'ını sıfırdan yazmak zorunda kalırdı.

collect()'in Üç Bileşeni: Supplier, Accumulator, Combiner

Bir `Collector`, üç fonksiyondan oluşur: bir **supplier** (sonucu tutacak boş bir kap oluşturur, örneğin boş bir `ArrayList`), bir **accumulator** (her elemanı bu kaba ekler), ve bir **combiner** (paralel stream'lerde kısmi sonuçları birleştirir). `Collectors` sınıfındaki her statik metot, bu üçlüyü sizin için hazır kurar -- kendi `Collector`'ınızı sıfırdan yazmanız neredeyse hiç gerekmez.

Collectors.toList() ve Collectors.toSet(): Basit Koleksiyonlar

`Collectors.toList()`, `Terminal Operations` dersindeki `Stream.toList()`'e çok benzer, ama ikisi sonuç hakkında farklı sözler verir: `Stream.toList()` (Java 16+) **değiştirilemez** (unmodifiable) bir liste döndürmeyi garanti eder; `collect(Collectors.toList())` ise döndürdüğü listenin türü veya değiştirilebilirliği hakkında **hiçbir garanti vermez** (bugünkü JDK'da bir `ArrayList` döner, ama bu sözleşmenin değil uygulamanın bir ayrıntısıdır). Değiştirebileceğiniz bir listeye ihtiyacınız varsa bunu `collect(Collectors.toCollection(ArrayList::new))` ile açıkça isteyin. `Collectors.toSet()`, elemanları bir `Set`'e toplar -- yinelenenler otomatik elenir, ama sıralama garantisi yoktur.

```
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(): String Birleştirme

`Collectors.joining()`, bir `String` stream'ini tek bir `String`'de birleştirir -- elle bir `StringBuilder` döngüsü yazmanın yerini alır. Üç aşırı yüklemesi vardır: argümansız (düz birleştirme), bir ayraç (delimiter) alan, ve bir ayraç ile bir önek/sonek (prefix/suffix) birlikte alan.

```
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);
    }
}
```

Beklenen Çıktı

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

Collectors.groupingBy(): Elemanları Gruplamak

`Collectors.groupingBy(classifier)`, her elemandan bir `Function` ile bir anahtar (key) türetip, elemanları bu anahtara göre gruplar; sonuç bir `Map<K, List<T>>`'dir -- her farklı anahtar, o anahtarı üreten tüm elemanların listesine eşlenir.

```
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 Collector'lar: counting() ve mapping()

`groupingBy()`, ikinci bir parametre olarak bir **downstream collector** kabul eder: varsayılan olarak her grup bir listeye toplanır, ama bir downstream collector verildiğinde, her grubun elemanlarına ne olacağına o karar verir. `Collectors.counting()`, her grubu doğrudan boyutuna indirger (`Map<K, Long>` üretir). `Collectors.mapping()`, her elemanı gruplanmadan **önce** dönüştürmenizi sağlar.

```
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(): İkiye Ayırma

`Collectors.partitioningBy(predicate)`, `groupingBy()`'ın özel bir halidir: elemanları bir `Predicate`'e göre tam olarak **iki** gruba (`true` / `false`) ayırır. `groupingBy()`'dan farklı olarak, sonuç `Map`'inde her iki anahtar da her zaman bulunur -- bir grup boş olsa bile, o anahtar boş bir liste ile birlikte haritada yer alır.

```
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(): Anahtar-Değer Eşlemesi Oluşturmak

`Collectors.toMap(keyMapper, valueMapper)`, bir stream'den bir `Map` oluşturur. En sivri köşesi: iki farklı eleman **aynı anahtarı** üretirse, varsayılan olarak `IllegalStateException` fırlatır -- bazı dillerin eşdeğerlerindeki gibi otomatik bir "sonuncusu kazanır" davranışı yoktur. Üçüncü bir argüman, bir `BinaryOperator<V>`, bu çakışmayı açıkça nasıl çözeceğinizi belirtmenizi sağlar.

```
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

- **`collect(Collectors.toList())` ile `Stream.toList()` arasında bilinçli seçim yapın.** Sonuca eleman eklemeniz/çıkarmanız gerekiyorsa ikisini de değil, değiştirilebilir bir liste garanti eden `collect(Collectors.toCollection(ArrayList::new))`'i kullanın; yalnızca okuyacaksanız `Stream.toList()` (immutable, niyeti daha net ifade eder).
- **`toMap()`'in üç argümanlı halini, anahtar çakışması olasılığı varsa baştan kullanın** -- iki argümanlı hali production'da beklenmedik bir `IllegalStateException`'a yol açabilir.
- **`groupingBy()` + downstream collector zincirini, iç içe elle yazılmış bir döngüden tercih edin** -- `groupingBy(classifier, counting())` gibi bir satır, aynı işi yapan bir `Map<K, List<T>>` + ayrı bir sayma döngüsünden daha az hataya açıktır.
- **`partitioningBy()`'ı yalnızca gerçekten iki grup varken kullanın** -- ikiden fazla kategori için `groupingBy()` doğru araçtır.

Yaygın Hatalar

- **`Collectors.toMap()`'i çakışan anahtarlarla, merge fonksiyonu olmadan kullanmak.** Veri her zaman benzersiz anahtarlar üretecekmiş gibi varsaymak, üretimde beklenmedik girdilerle `IllegalStateException`'a yol açar.
- **`collect(Collectors.toList())`'in döndürdüğü listenin değiştirilebilir olduğuna güvenmek.** Yalnızca `Stream.toList()`'in davranışı garantilidir (immutable); `Collectors.toList()` değiştirilebilirlik konusunda hiçbir söz vermez, yani sonucuna `add()` çağıran kod bir uygulama ayrıntısına bağımlıdır -- listeyi değiştirmeniz gerekiyorsa `Collectors.toCollection(ArrayList::new)` kullanın.
- **`groupingBy()`'ın sonucundaki her anahtarın var olacağını varsaymak.** Yalnızca `partitioningBy()` bunu garanti eder; `groupingBy()`'da hiç elemanı olmayan bir anahtar haritada hiç yer almaz.
- **`joining()`'i `String` olmayan bir stream'de kullanmaya çalışmak.** `Collectors.joining()` yalnızca `Stream<String>` üzerinde çalışır; başka bir tipte önce `map(Object::toString)` gerekir.

Özet, Cheat Sheet ve Terimler Sözlüğü

`Collectors`, `collect()` terminal operation'ına verilecek hazır tarifler sunar: `toList()`/`toSet()` basit koleksiyonlara toplar (`toList()` değiştirilebilirlik garantisi vermez, `Stream.toList()` ise her zaman immutable'dır -- garantili mutable bir liste için `toCollection(ArrayList::new)`), `joining()` string'leri birleştirir, `groupingBy()` bir anahtara göre gruplar (isteğe bağlı bir downstream collector'la, örneğin `counting()`/`mapping()`), `partitioningBy()` tam olarak iki gruba ayırır, ve `toMap()` bir `Map` oluşturur (çakışan anahtarlar için açık bir merge fonksiyonu gerektirebilir).

Hızlı referans:

```
stream.collect(Collectors.toList())             // List, mutability garantisi yok
stream.collect(Collectors.toSet())                // Set, yinelenensiz
stream.collect(Collectors.joining(", "))            // tek 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>
```

Terimler Sözlüğü

Collector — `collect()`'e verilen, bir stream'in nasıl bir sonuca toplanacağını tanımlayan nesne; bir supplier, accumulator ve combiner'dan oluşur.

Collectors — `Collector` nesneleri üreten hazır statik metotlar sağlayan yardımcı sınıf.

Downstream collector — `groupingBy()`/`partitioningBy()`'a verilen, her grubun elemanlarına ne olacağını belirleyen ikinci bir `Collector`.

groupingBy — Elemanları bir anahtara göre gruplayıp `Map<K, List<T>>` (veya downstream collector'a göre farklı bir değer tipi) üreten collector.

partitioningBy — Elemanları bir `Predicate`'e göre tam olarak iki gruba (`true`/`false`) ayıran, `groupingBy()`'ın özel bir hali olan collector.