Terminal Operations

Stream pipeline'ını tetikleyip sonuç üreten terminal operation'lar: forEach(), reduce(), count(), min()/max(), findFirst()/findAny(), anyMatch()/allMatch()/noneMatch(), toList() /toArray(). Kısa devre (short-circuiting) davranışı ve count()'un source boyutundan doğrudan hesaplanabilme optimizasyonu.

Orta 22 dk

Son güncelleme: 10 Kasım 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/terminal-operations

---

Terminal Operations

"Stream API Temelleri" dersinde bir stream pipeline'ının üç aşamasını görmüştünüz: source, intermediate operation'lar, ve tam olarak bir terminal operation. O ders intermediate operation'lara odaklanmıştı; bu ders, pipeline'ı gerçekten çalıştırıp bir sonuç üreten **terminal operation**'ları ele alıyor.

Terminal Operation Nedir?

Bir terminal operation, bir stream pipeline'ını **tüketen** ve bir sonuç (bir değer, bir koleksiyon, ya da hiçbir şey -- `void`) üreten son adımdır. "Stream Pipeline: Source, Intermediate, Terminal" bölümünde de belirtildiği gibi, bir pipeline'da tam olarak bir terminal operation bulunur; çağrıldığı anda tüm intermediate operation'lar zincirleme olarak, eleman eleman çalıştırılır.

Bu derste sırasıyla `forEach()`, `reduce()`, `count()`, `min()` / `max()`, `findFirst()` / `findAny()`, `anyMatch()` / `allMatch()` / `noneMatch()`, ve `toList()` / `toArray()`'i göreceksiniz. `collect()`'in tam gücü (özellikle `Collectors` sınıfı) bir sonraki derste.

Neden Var?

Intermediate operation'lar tembeldir (Lazy Evaluation, "Stream API Temelleri" dersi) -- kendi başlarına hiçbir şey üretmezler, yalnızca pipeline'ın tanımına adım eklerler. Bir sonuca gerçekten ihtiyaç duyduğunuzda (bir sayı, bir liste, bir `boolean`) bu tanımı **tetikleyecek** bir şeye ihtiyacınız var: işte bu, terminal operation'ın işi. Terminal operation olmadan bir stream pipeline'ı yalnızca bir tanımdır, hiçbir zaman çalışmaz.

Tarihçe

Terminal operation'ların çoğu, Stream API ile birlikte Java 8'de (2014) geldi. `toList()` bir istisna -- Java 16'ya (2021) kadar bir stream'i listeye çevirmenin tek yolu `collect(Collectors.toList())` idi; `toList()`, bu çok sık karşılaşılan ihtiyaç için sonradan eklenen bir kolaylık metodudur -- ama "toList() ve toArray(): Basit Koleksiyona Dönüştürme" bölümünde görüleceği gibi, sonucu farklı bir sözleşmeye sahiptir.

forEach(): Yan Etki Uygulamak

`forEach(Consumer<T>)`, her elemanda bir yan etki çalıştırır ve `void` döndürür -- bir `for-each` döngüsünün terminal operation karşılığıdır. Hiçbir şey döndürmediği için, bir pipeline'ı yalnızca **bitirebilir**, asla devam ettiremez. Ancak bir `for-each` döngüsünden farklı olarak `forEach()`, elemanları hangi sırayla işleyeceğine dair hiçbir söz vermez -- sıralı bir stream'de bile (JDK orada pratikte karşılaşma sırasını korur, ama API bunu garanti etmez); paralel bir stream'de ise sıra gerçekten değişir. Sıra önemliyse, elemanları her stream'de karşılaşma sırasıyla işleyen `forEachOrdered()` kullanılır.

```
import java.util.List;

// forEach(Consumer<T>) runs a side effect on every element and returns void -- it's the
// terminal operation equivalent of a for-each loop. Since it returns nothing, it can only
// end a pipeline, never continue one.
class ForEachExample {
    public static void main(String[] args) {
        List<String> names = List.of("Ahmet", "Mehmet", "Ayse");
        names.stream()
                .map(String::toUpperCase)
                .forEach(System.out::println);

        // forEach() makes no promise about the order it visits the elements in -- not even
on
        // a sequential stream (the JDK keeps encounter order there in practice, but the API
        // does not guarantee it), and on a parallel stream the order really does vary.
        // forEachOrdered() processes the elements in encounter order on any stream.
        names.stream().forEachOrdered(name -> System.out.print(name + " "));
        System.out.println();
    }
}
```

count(): Eleman Sayısı

`count()`, terminal operation'a ulaşan eleman sayısını `long` olarak döndürür. Peek()/lazy evaluation bölümünde detaylandırılacağı gibi, `count()`'un davranışı göründüğünden daha ilginçtir.

reduce(): Elemanları Tek Bir Değere İndirgemek

`reduce()`, tüm elemanları, iki değeri birleştirip bir tane üreten bir `BinaryOperator` ile, tekrar tekrar uygulanarak **tek bir değere** indirger. Üç aşırı yüklemesi (overload) vardır, temel fark başlangıç değeri (identity) verilip verilmemesi:

- `reduce(identity, accumulator)`: `identity`'den başlar, boş stream için bile her zaman bir değer döndürür (doğrudan `identity`'yi).
- `reduce(accumulator)`: başlangıç değeri yok -- boş bir stream'in döndürecek bir değeri olmayacağından, bu aşırı yükleme `T` yerine `Optional<T>` döndürür.
- Üç parametreli `reduce(identity, accumulator, combiner)` (bu örnekte kullanılmadı) paralel stream'ler için kısmi sonuçları birleştirmeye yarar.

```
import java.util.List;
import java.util.Optional;

// reduce() combines all elements into a SINGLE value, using a BinaryOperator that
// combines two values into one, applied repeatedly. Three overloads exist, differing in
// whether a starting value (identity) is given.
class ReduceExample {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(1, 2, 3, 4, 5);

        // reduce(identity, accumulator): starts from `identity`, always returns a value
        // (never empty), even for an empty stream (it just returns the identity).
        int sum = numbers.stream().reduce(0, Integer::sum);
        System.out.println(sum);

        // reduce(accumulator) with no identity: since an empty stream would have no
        // value to return, this overload returns Optional<T> instead of T.
        Optional<Integer> product = numbers.stream().reduce((a, b) -> a * b);
        System.out.println(product.orElse(0));

        // Building a single String out of multiple strings -- reduce() isn't limited to
        // numbers, any associative combining operation works.
        List<String> words = List.of("Java", "Stream", "API");
        String joined = words.stream().reduce("", (a, b) -> a.isEmpty() ? b : a + " " + b);
        System.out.println(joined);
    }
}
```

Beklenen Çıktı

```
15
120
Java Stream API
```

min() ve max(): Comparator ile Uç Değerler

`min()` ve `max()`, bir `Comparator` gerektirir -- parametresiz bir aşırı yükleme yoktur, çünkü stream'in eleman tipi her zaman `Comparable` olmak zorunda değildir. `reduce(accumulator)` ile aynı sebeple, ikisi de `Optional<T>` döndürür: boş bir stream'in ne minimumu ne de maksimumu vardır.

```
import java.util.Comparator;
import java.util.List;
import java.util.Optional;

// count() returns how many elements reached the terminal operation, as a long.
// min()/max() need a Comparator to know what "smallest"/"largest" means -- there's no
// parameterless overload, since the stream's element type might not be Comparable.
// Both return Optional<T>, for the same reason as reduce(accumulator): an empty stream
// has no minimum or maximum.
class CountMinMaxExample {
    public static void main(String[] args) {
        List<String> names = List.of("Ahmet", "Mehmet", "Ayse", "Ali");

        long count = names.stream().filter(n -> n.length() > 3).count();
        System.out.println(count);

        Optional<String> shortest = names.stream().min(Comparator.comparingInt(String::
length));
        System.out.println(shortest.orElse("none"));

        Optional<String> longest = names.stream().max(Comparator.comparingInt(String::
length));
        System.out.println(longest.orElse("none"));

        // Comparator.naturalOrder() uses the type's own Comparable -- alphabetical here.
        Optional<String> firstAlphabetically = names.stream().min(Comparator.
naturalOrder());
        System.out.println(firstAlphabetically.orElse("none"));
    }
}
```

Beklenen Çıktı

```
3
Ali
Mehmet
Ahmet
```

findFirst() ve findAny(): İlk/Herhangi Bir Eşleşme

`findFirst()`, karşılaşma sırasına (encounter order) göre **ilk** elemanı, `findAny()` ise **herhangi bir** elemanı `Optional<T>` olarak döndürür. Bu kursta yalnızca sıralı (sequential)

stream'ler kullanıldığı için ikisi aynı şekilde davranır; fark yalnızca paralel stream'lerde ortaya çıkar (`findAny()` orada daha hızlı olabilir, çünkü ilk bulunan sonucu beklemek zorunda değildir).

anyMatch(), allMatch(), noneMatch(): Kısa Devre Kontrolleri

Bu üç metot, bir `Predicate` ile stream hakkında evet/hayır sorusu sorar ve düz bir `boolean` döndürür: `anyMatch()` en az bir eleman koşulu sağlıyor mu, `allMatch()` tüm elemanlar sağlıyor mu, `noneMatch()` hiçbir eleman sağlamıyor mu.

```
import java.util.List;
import java.util.Optional;

// findFirst()/findAny() return Optional<T> -- the FIRST element (in encounter order) or
// ANY element satisfying the pipeline so far; for a sequential stream both behave the
// same, findAny() only differs (and can be faster) for parallel streams.
// anyMatch()/allMatch()/noneMatch() ask a yes/no question about the whole stream, using
// a Predicate, and return a plain boolean.
class FindMatchExample {
    public static void main(String[] args) {
        List<String> names = List.of("Ahmet", "Mehmet", "Ayse", "Ali");

        Optional<String> firstLong = names.stream()
                .filter(n -> n.length() > 4)
                .findFirst();
        System.out.println(firstLong.orElse("none"));

        Optional<String> anyStartingWithA = names.stream()
                .filter(n -> n.startsWith("A"))
                .findAny();
        System.out.println(anyStartingWithA.orElse("none"));

        boolean hasShortName = names.stream().anyMatch(n -> n.length() <= 3);
        System.out.println(hasShortName);

        boolean allStartWithCapital = names.stream().allMatch(n -> Character.isUpperCase(n.
charAt(0)));
        System.out.println(allStartWithCapital);

        boolean noneAreEmpty = names.stream().noneMatch(String::isEmpty);
        System.out.println(noneAreEmpty);
    }
}
```

toList() ve toArray(): Basit Koleksiyona Dönüştürme

`toList()` (Java 16), `collect(Collectors.toList())` ile aynı amaca hizmet eder, ama ikisi sonuç hakkında aynı sözü vermez: `toList()`'in döndürdüğü liste **değiştirilemezdir** (unmodifiable), `collect(Collectors.toList())` ise döndürdüğü listenin türü veya değiştirilebilirliği hakkında hiçbir garanti vermez. Değiştirilebilir bir `ArrayList` gerekiyorsa `collect(Collectors.toCollection(ArrayList::new))` kullanın. `toArray()`, stream'i bir `List`

yerine bir diziye çevirir; eleman tipini bilen bir dizi üretmek için genellikle `String[]::new` gibi bir constructor reference alır (Built-in Functional Interfaces dersindeki `Class::new` biçimi).

```
import java.util.List;

// toList() (added in Java 16) serves the same purpose as the far more general
// collect(Collectors.toList()) -- covered fully in the next lesson, Collectors. It
// returns an UNMODIFIABLE List, while Collectors.toList() makes no guarantee about
// mutability (use Collectors.toCollection(ArrayList::new) when you need a mutable list).
// toArray() converts a stream into an array instead of a List.
class ToListToArrayExample {
    public static void main(String[] args) {
        List<String> names = List.of("Ahmet", "Mehmet", "Ayse");

        List<String> upper = names.stream().map(String::toUpperCase).toList();
        System.out.println(upper);

        try {
            upper.add("EXTRA");
        } catch (UnsupportedOperationException e) {
            System.out.println("caught: toList() result is unmodifiable");
        }

        String[] asArray = names.stream().toArray(String[]::new);
        System.out.println(asArray.length + " " + asArray[0]);
    }
}
```

Beklenen Çıktı

```
[AHMET, MEHMET, AYSE]
caught: toList() result is unmodifiable
3 Ahmet
```

Kısa Devre ve count()'un Şaşırtıcı Davranışı

Bazı terminal operation'lar **kısa devre yapar (short-circuit)**: cevap netleştiği anda pipeline'ı durdurur, kalan elemanları hiç işlemez. `anyMatch()` ilk eşleşmede durur; `findFirst()` ilk sonucu bulduğunda durur.

`count()` ise ayrı ve gerçekten şaşırtıcı bir durum: bazı durumlarda JDK, sayıyı doğrudan kaynağın bilinen boyutundan hesaplayabilir ve pipeline'ı **hiç çalıştırmadan** atlayabilir. Bu gerçekleştiğinde, aradaki `peek()` gibi intermediate operation'lar bile hiç çağrılmaz -- bu, JDK dokümantasyonunda açıkça belirtilen, kasıtlı bir optimizasyondur, bir hata değil. Aşağıdaki örnekte bunu gerçek bir `count()` çağrısıyla gözlemleyebilirsiniz: `peek()` içindeki yazdırma satırı **hiç çalışmaz**.

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

// Some terminal operations SHORT-CIRCUIT: they stop pulling elements through the
// pipeline as soon as the answer is known, instead of processing every element. This is
// only visible in combination with lazy evaluation (previous lesson) -- peek() lets us
// observe exactly how many elements were actually pulled through.
class ShortCircuitExample {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8);

        // anyMatch() stops at the FIRST match -- elements after it are never touched.
        boolean hasEven = numbers.stream()
                .peek(n -> System.out.println("checking: " + n))
                .anyMatch(n -> n % 2 == 0);
        System.out.println("result: " + hasEven);

        // findFirst() stops as soon as one matching element is found -- filter() itself
        // isn't short-circuiting, but findFirst() stops asking it for more once the
        // first match arrives.
        numbers.stream()
                .peek(n -> System.out.println("scanning: " + n))
                .filter(n -> n > 5)
                .findFirst();

        // count() is a special, surprising case -- NOT because it short-circuits after
        // finding an answer mid-stream, but because the JDK can sometimes compute the
        // count directly from the source's known size, skipping the pipeline entirely.
        // When that happens, peek() is never invoked at all (this is explicitly
        // documented behavior, not a bug): running this prints NO "counting: n" lines.
        long total = Stream.of(1, 2, 3)
                .peek(n -> System.out.println("counting: " + n))
                .count();
        System.out.println("total: " + total);
    }
}
```

Best Practices

- **`peek()`'e (bir önceki dersten) veya yan etkilere dayanan varsayımlar kurmayın.** `count()` örneğinde görüldüğü gibi, JDK bazı intermediate operation'ları atlayabilir; yan etkiler için `forEach()` veya doğrudan bir döngü kullanın.
- **`reduce(accumulator)` / `min()` / `max()`'ın `Optional<T>` döndürdüğünü unutmayın** -- boş bir stream ihtimaline karşı `orElse()` / `orElseThrow()` gibi bir sonlandırma her zaman gerekir (Optional, ayrı bir derste detaylı ele alınacak).
- **`findAny()`'i yalnızca gerçekten "hangi eleman olduğu önemli değil" durumunda kullanın** -- `findFirst()` niyeti daha net ifade eder ve sıralı stream'lerde ek bir performans kazancı sağlamaz.
- **`toList()`'in sonucunun değiştirilemez olduğunu unutmayın** -- değiştirilebilir bir liste gerekiyorsa `collect(Collectors.toCollection(ArrayList::new))` (bir sonraki dersin konusu) veya sonucu yeni bir `ArrayList`'e sarmayı düşünün.

Yaygın Hatalar

- **`reduce()`'un boş stream davranışını unutmak.** `reduce(accumulator)` boş bir stream için `Optional.empty()` döndürür; `.get()` ile doğrudan açmaya çalışmak `NoSuchElementException` fırlatır.
- **`min()` / `max()`'ın sonucunu kontrolsüz açmak.** Aynı risk `min()` / `max()` için de geçerli -- ikisi de boş stream'de boş `Optional` döner.
- **`count()`'un her zaman tüm elemanları işlediğini varsaymak.** Yukarıda görüldüğü gibi bu doğru değil; `peek()` ile debug yaparken `count()`'un beklenmedik şekilde hiçbir çıktı vermemesi bu yüzdendir.
- **`toList()`'in döndürdüğü listeye eleman eklemeye çalışmak.** `UnsupportedOperationException` fırlatır -- `List.of()`'un döndürdüğü listelerle aynı immutability kısıtı.

Özet, Cheat Sheet ve Terimler Sözlüğü

Bir terminal operation, bir stream pipeline'ını tüketip bir sonuç üretir: `forEach()` bir yan etki uygular, `reduce()` elemanları tek bir değere indirger, `count()` eleman sayısını verir (ama bazen pipeline'ı hiç çalıştırmadan), `min()` / `max()` bir `Comparator`'a göre uç değerleri bulur, `findFirst()` / `findAny()` bir eşleşmeyi `Optional<T>` olarak döndürür, `anyMatch()` / `allMatch()` / `noneMatch()` evet/hayır sorularına `boolean` döndürür, ve `toList()` / `toArray()` sonucu basit bir koleksiyona çevirir. `anyMatch()` ve `findFirst()` gibi bazı operation'lar kısa devre yapar; `count()` özel bir kaynak-boyutu optimizasyonuna sahiptir.

Hızlı referans:

```
stream.forEach(x -> ...)      // yan etki, void döner, sıra garantisi yok
stream.forEachOrdered(x -> ...) // aynısı, karşılaşma sırasıyla
stream.count()                 // long, bazen kaynaktan doğrudan hesaplanır
stream.reduce(id, op)           // T, her zaman değer döner
stream.reduce(op)                 // Optional<T>
stream.min(cmp) / .max(cmp)        // Optional<T>
stream.findFirst() / .findAny()     // Optional<T>
stream.anyMatch(p) / .allMatch(p)    // boolean, kısa devre
stream.noneMatch(p)                   // boolean, kısa devre
stream.toList() / .toArray(gen)        // List<T> (değiştirilemez) / T[]
```

Terimler Sözlüğü

Terminal operation — Bir stream pipeline'ını tüketip bir sonuç üreten, pipeline'ı tetikleyen son adım.

Short-circuiting (kısa devre) — Bir terminal operation'ın, cevap netleştiği anda kalan elemanları işlemeden pipeline'ı durdurması.

reduce — Tüm elemanları, ikili bir birleştirme fonksiyonuyla tekrar tekrar uygulanarak tek bir değere indirgeyen terminal operation.

Optional — Bir değerin bulunmama ihtimalini tip sisteminde ifade eden sarmalayıcı; `reduce(accumulator)`, `min()`, `max()`, `findFirst()`, `findAny()` tarafından döndürülür (ayrı bir derste detaylı ele alınıyor).

Encounter order (karşılaşma sırası) — Bir stream'in elemanlarının işlendiği sıra; `findFirst()`, bu sıraya göre ilk elemanı döndürür.