Stream API Temelleri

Stream API'nin temelleri: Stream nedir, Collection'dan Stream'e geçiş, pipeline'ın üç aşaması (source/intermediate/terminal). Intermediate operation'lar: filter(), map(), flatMap(), distinct(), sorted(), peek(), limit(), skip(). Lazy evaluation ve stream'in tek kullanımlık doğası.

Orta 25 dk

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

---

Stream API Fundamentals & Intermediate Operations

Bir önceki iki derste lambda syntax'ını ve `java.util.function` paketindeki hazır interface'leri gördünüz. Bu ders, onların asıl var olma sebebine geliyor: **Stream API**. Bir koleksiyon üzerinde "şunu filtrele, şunu dönüştür, şöyle sırala" gibi işlemleri, `for` döngüsü yazmadan, bildirimsel (declarative) bir zincir olarak ifade etmenin yolu.

Stream Nedir?

Bir `Stream<T>`, bir veri kaynağından (genellikle bir `Collection`) gelen elemanlar üzerinde sırayla çalışan bir **işlem hattıdır (pipeline)**. Kritik nokta: Stream, veriyi **saklamaz**. Bir liste veya set gibi bir veri yapısı değildir; kaynaktaki verinin üzerinden bir kez geçmenizi sağlayan, tek kullanımlık bir borudur.

Bir stream pipeline'ı üç parçadan oluşur: bir **source** (`list.stream()` gibi), sıfır veya daha fazla **intermediate operation** (`filter()`, `map()` gibi -- bu dersin konusu), ve tam olarak bir **terminal operation** (`toList()`, `forEach()` gibi -- bir sonraki dersin konusu).

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

// A Stream doesn't store data -- it's a pipeline that pulls elements from a SOURCE.
// The three most common sources: a Collection's stream() method, Stream.of() for
// literal values, and Arrays.stream() for arrays.
class StreamCreationExample {
    public static void main(String[] args) {
        List<String> names = List.of("Ahmet", "Mehmet", "Ayse");
        Stream<String> fromCollection = names.stream();
        System.out.println(fromCollection.count());

        Stream<String> fromLiterals = Stream.of("a", "b", "c");
        System.out.println(fromLiterals.count());

        String[] array = {"x", "y", "z"};
        Stream<String> fromArray = Arrays.stream(array);
        System.out.println(fromArray.count());

        // Stream.empty() and Stream.generate()/Stream.iterate() are less common sources --
        // iterate() (like generate()) has no natural end, so it needs a limit() or it
never stops.
        Stream<Integer> generated = Stream.iterate(1, n -> n * 2).limit(5);
        System.out.println(generated.toList());
    }
}
```

Beklenen Çıktı

```
3
3
3
[1, 2, 4, 8, 16]
```

Neden Var?

Java 8 öncesinde bir koleksiyonu filtreleyip dönüştürmek, elle yazılmış bir `for` döngüsü, geçici bir sonuç listesi ve döngü içinde `if` kontrolleri gerektirirdi -- **nasıl** yapılacağını adım adım anlatan (imperative) bir kod. Stream API, aynı işi **ne** istediğinizi tanımlayan (declarative) bir zincirle ifade etmenizi sağlar: `filter(...).map(...).toList()` okuyan kişiye doğrudan niyeti anlatır, döngü mekaniğini değil.

Bu farkı küçük bir örnekle görelim:

```
List<String> names = List.of("Ahmet", "Mehmet", "Ayse", "Ali");
List<String> result = names.stream()
        .filter(name -> name.startsWith("A"))
        .map(String::toUpperCase)
        .toList();
```

Burada `filter` bir `Predicate<String>` ("Built-in Functional Interfaces" dersindeki `Predicate`), `map` bir `Function<String,String>` (aynı dersteki `Function`) bekliyor, ve `name -> name.startsWith("A")` ile `String::toUpperCase` sırasıyla bir lambda ve bir method reference ("Lambda Expressions" ve "Built-in Functional Interfaces" dersleri). Yani bu zincir, önceki üç dersin ("Interface" dersindeki functional interface temeli, "Lambda Expressions", "Built-in Functional Interfaces") tam olarak bir araya geldiği yer.

Tarihçe

Stream API, `java.util.function` paketiyle birlikte Java 8'de (2014) geldi. İkisi birbirine sıkı sıkıya bağlıdır: Stream API'nin `filter()`, `map()`, `reduce()` gibi metotları, parametre olarak tam olarak `java.util.function` paketindeki tipleri (`Predicate`, `Function`, `BinaryOperator`) bekler. Stream API olmadan bu interface'lerin çoğu bu kadar sık kullanılmazdı; bu interface'ler olmadan da Stream API'nin metotları tip güvenli bir şekilde tanımlanamazdı.

Collection'dan Stream'e: stream() ve of()

En yaygın kaynak, herhangi bir `Collection`'ın (`List`, `Set`, ...) `stream()` metodudur. Bunun dışında `Stream.of(...)` literal değerlerden, `Arrays.stream(array)` bir diziden, `Stream.iterate(...)` ise bir üretim kuralından stream oluşturur -- `iterate()` doğal bir sonu olmadığı için genellikle `limit()` ile sınırlandırılır.

Stream Pipeline: Source, Intermediate, Terminal

Bir stream pipeline'ının üç aşaması vardır: **source** verinin nereden geldiğini belirler, **intermediate operation'lar** (bu dersin konusu -- `filter`, `map`, `flatMap`, `distinct`, `sorted`, `peek`, `limit`, `skip`) veriyi adım adım dönüştürür, ve tam olarak bir **terminal operation** (bir sonraki derste) pipeline'ı tetikleyip sonucu üretir. Intermediate operation'ların hepsi bir `Stream` döndürür -- bu da zincirlemeyi (method chaining) mümkün kılar.

filter(): Eleme

`filter(Predicate<T>)`, yalnızca verilen koşulu sağlayan elemanları bırakır; stream **kısalabilir** ama eleman tipi değişmez. `Predicate`, "Built-in Functional Interfaces" dersinde detaylıca işlenmişti -- burada doğrudan kullanılıyor.

map() ve flatMap(): Dönüştürme ve Düzleştirme

`map(Function<T,R>)`, her elemanı bire bir başka bir değere dönüştürür; stream'in **uzunluğu değişmez**, ama eleman tipi/değeri değişebilir.

`flatMap()`, `map()`'in bir tuzağını çözer: dönüştürme fonksiyonu kendisi bir `Stream`/koleksiyon döndürürse, `map()` bir "stream'lerin stream'i" üretir -- kullanışsız, iç içe bir yapı. `flatMap()`, her elemanı bir stream'e çevirip bu stream'leri **tek bir düz stream'de**

birleştirir. Tipik kullanım: bir "liste listesi"ni tek bir listeye düzleştirmek, ya da her cümleyi kelimelerine ayırıp hepsini tek bir kelime listesinde toplamak.

```
import java.util.List;

// filter() keeps only elements matching a Predicate -- the stream may get SHORTER.
// map() transforms each element with a Function -- the stream stays the same length,
// but element type/value can change. Chaining them is the core Stream idiom.
class FilterMapExample {
    public static void main(String[] args) {
        List<String> names = List.of("Ahmet", "Mehmet", "Ayse", "Ali");

        List<String> result = names.stream()
                .filter(name -> name.startsWith("A"))
                .map(String::toUpperCase)
                .toList();
        System.out.println(result);

        // Order matters for performance (not correctness here): filtering before an
        // expensive map() means map() runs on fewer elements.
        List<Integer> lengths = names.stream()
                .filter(name -> name.length() > 4)
                .map(String::length)
                .toList();
        System.out.println(lengths);
    }
}
```

```
import java.util.Arrays;
import java.util.List;

// map() turns each element into ONE new element -- if the mapping function itself
// returns a Stream/List, map() produces a Stream of Streams, which is rarely what you
// want. flatMap() turns each element into a Stream and then MERGES all of those
// streams into a single, flat stream -- exactly the tool for "list of lists" ->
// "single list" problems.
class FlatMapExample {
    public static void main(String[] args) {
        List<List<String>> nested = List.of(
                List.of("a", "b"),
                List.of("c"),
                List.of("d", "e", "f")
        );

        // map(List::stream) would give a Stream<Stream<String>> -- each inner list
        // becomes its OWN stream, still nested one level too deep to use directly.
        long innerStreamCount = nested.stream().map(List::stream).count();
        System.out.println(innerStreamCount);

        // flatMap(List::stream) merges every inner stream into one flat Stream<String>.
        List<String> flat = nested.stream()
                .flatMap(List::stream)
                .toList();
        System.out.println(flat);

        // A common real use: splitting each sentence into words, then flattening into
        // a single list of all words across all sentences.
        List<String> sentences = List.of("hello world", "java streams");
        List<String> words = sentences.stream()
                .flatMap(sentence -> Arrays.stream(sentence.split(" ")))
                .toList();
        System.out.println(words);
    }
}
```

Beklenen Çıktı

```
3
[a, b, c, d, e, f]
[hello, world, java, streams]
```

distinct(), sorted(), peek()

`distinct()`, `equals()`'a göre yinelenen elemanları eler. `sorted()`, ya doğal sıralamaya (`Comparable`) ya da verilen bir `Comparator`'a göre elemanları sıraya koyar. `peek()`, stream'i **değiştirmeden** her elemanda bir `Consumer` çalıştırır -- yalnızca gözlemlemek içindir, genellikle debug amaçlı; üretim kodunda yan etki için `peek()`'e güvenmek önerilmez (bkz. Yaygın Hatalar).

```
import java.util.List;

// distinct() removes duplicates (using equals()). sorted() orders elements, either by
// natural ordering (Comparable) or a given Comparator. peek() runs a Consumer on each
// element WITHOUT changing the stream -- it exists to observe a pipeline, most often
// for debugging, not for production side effects.
class DistinctSortedPeekExample {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(3, 1, 2, 3, 1, 4);

        List<Integer> distinctSorted = numbers.stream()
                .distinct()
                .sorted()
                .toList();
        System.out.println(distinctSorted);

        // sorted(Comparator) for custom ordering -- here, longest name first.
        List<String> names = List.of("Ayse", "Ali", "Mehmet");
        List<String> byLengthDesc = names.stream()
                .sorted((a, b) -> b.length() - a.length())
                .toList();
        System.out.println(byLengthDesc);

        // peek() prints each element as it flows through the pipeline, without
        // changing what the next operation sees.
        List<Integer> result = numbers.stream()
                .distinct()
                .peek(n -> System.out.println("peeked: " + n))
                .sorted()
                .toList();
        System.out.println(result);
    }
}
```

limit() ve skip()

`limit(n)`, pipeline'dan en fazla ilk `n` elemanı bırakır ve ardından işlemi erken durdurur. `skip(n)`, ilk `n` elemanı atlar, kalanını bırakır. İkisi birlikte, sayfalama (pagination) mantığının temel taşlarıdır: `skip((sayfa - 1) * sayfaBoyutu).limit(sayfaBoyutu)`.

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

// limit(n) keeps at most the first n elements and then stops the pipeline early.
// skip(n) discards the first n elements and keeps the rest. Together, they're the
// building blocks of pagination: skip((page - 1) * pageSize).limit(pageSize).
class LimitSkipExample {
    public static void main(String[] args) {
        List<Integer> numbers = IntStream.rangeClosed(1, 10).boxed().toList();

        List<Integer> firstThree = numbers.stream().limit(3).toList();
        System.out.println(firstThree);

        List<Integer> skipFirstSeven = numbers.stream().skip(7).toList();
        System.out.println(skipFirstSeven);

        // A simple "page 2 of size 3" -- skip the first page, then take the second.
        int pageSize = 3;
        int page = 2;
        List<Integer> secondPage = numbers.stream()
                .skip((long) (page - 1) * pageSize)
                .limit(pageSize)
                .toList();
        System.out.println(secondPage);
    }
}
```

Beklenen Çıktı

```
[1, 2, 3]
[8, 9, 10]
[4, 5, 6]
```

Lazy Evaluation: Intermediate Operation'lar Ne Zaman Çalışır?

Intermediate operation'lar **tembeldir (lazy)**: `filter()` veya `map()` çağırmak, henüz hiçbir şeyi çalıştırmaz, yalnızca pipeline'ın tanımına bir adım ekler. Gerçek çalışma, ancak bir **terminal operation** çağrıldığında başlar -- ve o zaman bile eleman eleman, tek bir geçişte ilerler (her eleman, sırayla tüm intermediate operation'lardan geçer, sonra bir sonraki elemana geçilir).

Ayrıca bir stream **tek kullanımlıktır**: bir terminal operation çalıştığında stream kapanır; aynı stream referansını tekrar kullanmaya çalışmak `IllegalStateException` fırlatır.

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

// Intermediate operations (filter, map, peek, ...) are LAZY -- calling them just
// builds up a pipeline description, nothing runs yet. Only a TERMINAL operation
// (count, toList, forEach, ...) actually triggers execution, pulling elements through
// the whole pipeline one at a time. A stream is also SINGLE-USE: once a terminal
// operation runs, that stream is closed and reusing it throws.
class LazyEvaluationExample {
    public static void main(String[] args) {
        Stream<String> pipeline = Stream.of("a", "b", "c")
                .peek(s -> System.out.println("processing: " + s))
                .filter(s -> !s.equals("b"));

        System.out.println("pipeline built, nothing printed yet");
        long count = pipeline.count();
        System.out.println("count: " + count);

        // Reusing an already-consumed stream throws IllegalStateException -- each
        // Stream is meant to be built and consumed exactly once.
        Stream<String> once = Stream.of("x", "y");
        once.count();
        try {
            once.count();
        } catch (IllegalStateException e) {
            System.out.println("caught: " + e.getMessage());
        }
    }
}
```

Best Practices

- **Zinciri küçük ve okunabilir tutun.** Her satıra bir operation koymak (`filter` bir satır, `map` bir satır) zinciri tarayarak okumayı kolaylaştırır.
- **`filter()`'ı mümkün olduğunca erken uygulayın.** Pahalı bir `map()`'ten önce ucuz bir `filter()` koymak, `map()`'in daha az elemanda çalışmasını sağlar.
- **`peek()`'i yalnızca gözlem/debug için kullanın**, üretim mantığının parçası olarak değil -- bir sonraki bölümde bu ayrım detaylandırılıyor.
- **Stream'i bir kez kullanıp bırakın.** Bir stream'i bir değişkende saklayıp birden fazla terminal operation'la tekrar tekrar kullanmaya çalışmayın; bunun yerine ihtiyaç duyduğunuzda kaynaktan (`list.stream()`) yeni bir stream oluşturun.

Yaygın Hatalar

- **`peek()`'i yan etki üretmek için kullanmak.** `peek()`, dokümantasyonda açıkça "öncelikle debug amaçlı" olarak tanımlanır; JVM optimizasyonları bazı durumlarda `peek()` çağrılarını atlayabilir, bu yüzden ona güvenilir bir yan etki mekanizması gibi davranmak kırılgan koddur.

- **`map()` ile `flatMap()`'i karıştırmak.** Dönüştürme fonksiyonu bir `Stream` / `List` döndürüyorsa ve siz `map()` kullandıysanız, elinizde işe yaramaz bir "stream'lerin stream'i" kalır; ihtiyacınız `flatMap()`'tir.
- **Tüketilmiş bir stream'i tekrar kullanmaya çalışmak.** Bir terminal operation'dan sonra aynı `Stream` referansına tekrar bir operation uygulamak `IllegalStateException` fırlatır -- her ihtiyaçta kaynaktan yeni bir stream alın.
- **`Stream.iterate()`'i `limit()` olmadan çağırmak.** Doğal bir sonu olmayan bir üretim kuralına sınır koymamak, pipeline'ın sonsuza kadar (ya da bellek tükenene kadar) çalışmasına yol açar.

Özet, Cheat Sheet ve Terimler Sözlüğü

Bir `Stream`, veri saklamayan, bir kaynak üzerinde tek geçişlik bir işlem hattıdır: bir **source** (`collection.stream()`, `Stream.of()`, `Arrays.stream()`), sıfır veya daha fazla **intermediate operation** (`filter`, `map`, `flatMap`, `distinct`, `sorted`, `peek`, `limit`, `skip` -- hepsi tembel, hepsi bir `Stream` döndürür) ve tam olarak bir **terminal operation**'dan oluşur. Bir stream tek kullanımlıktır; tüketildikten sonra tekrar kullanılamaz.

Hızlı referans:

```
list.stream()
    .filter(x -> ...)     // eleme, stream kısalabilir
    .map(x -> ...)         // dönüştürme, uzunluk sabit
    .flatMap(x -> ...)     // dönüştür + düzleştir
    .distinct()             // yinelenenleri ele
    .sorted()                // sırala
    .peek(x -> ...)          // gözlemle, değiştirme
    .limit(n)                 // ilk n eleman
    .skip(n)                   // ilk n elemanı atla
```

Terimler Sözlüğü

Stream — Bir veri kaynağı üzerinde sırayla çalışan, veri saklamayan, tek kullanımlık işlem hattı.

Source (kaynak) — Bir stream pipeline'ının veri aldığı yer (`collection.stream()`, `Stream.of()` gibi).

Intermediate operation — Bir `Stream` alıp bir `Stream` döndüren, tembel çalışan pipeline adımı (`filter`, `map`, `flatMap`, `distinct`, `sorted`, `peek`, `limit`, `skip`).

Terminal operation — Pipeline'ı tetikleyip bir sonuç üreten, stream'i tüketen adım; sonraki derste ele alınıyor.

Lazy evaluation (tembel değerlendirme) — Intermediate operation'ların, bir terminal operation çağrılana kadar hiçbir iş yapmaması.

flatMap — Her elemanı bir stream'e çevirip bu stream'leri tek bir düz stream'de birleştiren operation.