Primitive & Parallel Streams

IntStream/LongStream/DoubleStream ile autoboxing maliyetinden kaçınmak, boxed() /mapToInt() köprüleri. parallelStream() ile paralel çalışma, forEach() vs forEachOrdered() sıralama farkı, thread-safe olmayan paylaşılan durum tuzağı, ne zaman kullanılmalı ve neden her zaman daha hızlı olmadığı.

Orta 25 dk

Son güncelleme: 5 Kasım 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/primitive-parallel-streams

---

Primitive & Parallel Streams

Bu, **Functional Interface'ler ve Stream'ler** kategorisinin son konusu. İki ayrı ama ilişkili konuyu bir araya getiriyor: `int` / `long` / `double` gibi primitive tipler için özelleşmiş stream'ler, ve bir stream pipeline'ını birden çok thread'e dağıtan paralel stream'ler.

Primitive Stream Nedir?

Bir `Stream<Integer>`, her elemanı bir `Integer` **nesnesi** olarak tutar -- her `int` değeri, otomatik olarak (autoboxing) bir nesneye sarılır. `IntStream` (ve karşılıkları `LongStream`, `DoubleStream`), bu sarmalamayı atlayan, doğrudan primitive değerler üzerinde çalışan özelleşmiş stream tipleridir.

Neden Var?

Autoboxing bedava değildir -- her `int`'i bir `Integer` nesnesine çevirmek, ekstra bellek ayırma ve bir referans katmanı demektir. Milyonlarca elemanlı bir stream'de bu maliyet gözle görülür hale gelir. `IntStream` / `LongStream` / `DoubleStream`, bu maliyeti tamamen ortadan kaldırır; ayrıca `sum()`, `average()` gibi, yalnızca sayılar için anlamlı olan ve genel `Stream<T>`'de bulunmayan metotları doğrudan sunar.

Tarihçe

Primitive stream tipleri, Stream API ile birlikte Java 8'de (2014) geldi -- tasarımcıların, autoboxing maliyetini bilinçli olarak API'nin temel bir parçası haline getirme kararının bir sonucu. Üç tip vardır: `IntStream`, `LongStream`, `DoubleStream` -- `short`, `byte`, `float` için ayrı bir stream tipi yoktur, bunlar gerektiğinde `int` / `double`'a genişletilir (widening).

IntStream Oluşturmak: range(), rangeClosed(), of()

`IntStream.range(başlangıç, bitiş)`, bitiş **hariç** bir aralık üretir (`[başlangıç, bitiş)`); `IntStream.rangeClosed(başlangıç, bitiş)`, bitiş **dahil** üretir. `IntStream.of(...)`, literal değerlerden bir stream oluşturur -- `Stream.of(...)`'un primitive karşılığı. Örnek, aralıkları `forEachOrdered()` ile yazdırıyor, çünkü karşılaşma sırasını yalnızca o garanti eder; düz `forEach()`'in neden etmediği "Sıralama: forEach() vs forEachOrdered()" bölümünde anlatılıyor.

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

// IntStream (LongStream/DoubleStream work the same way) is a stream SPECIALIZED for a
// primitive type -- avoiding the overhead of boxing every element into an Integer
// object. It offers aggregate methods a generic Stream<Integer> doesn't have directly:
// sum(), average(), max(), min().
class IntStreamCreationExample {
    public static void main(String[] args) {
        // forEachOrdered() processes the elements in encounter order (1, 2, 3, ...), which
the
        // API guarantees; plain forEach() promises no order, not even on a sequential
stream.
        IntStream.range(1, 5).forEachOrdered(n -> System.out.print(n + " ")); // exclusive
end
        System.out.println();

        IntStream.rangeClosed(1, 5).forEachOrdered(n -> System.out.print(n + " ")); //
inclusive
        System.out.println();

        int sum = IntStream.rangeClosed(1, 100).sum();
        System.out.println(sum);

        OptionalDouble average = IntStream.of(2, 4, 6, 8).average();
        System.out.println(average.orElse(0));

        int max = IntStream.of(3, 7, 2, 9, 1).max().orElse(Integer.MIN_VALUE);
        System.out.println(max);
    }
}
```

Beklenen Çıktı

```
1 2 3 4
1 2 3 4 5
5050
5.0
9
```

Primitive Stream'e Özel Metotlar: sum(), average(), max(), min()

`sum()`, doğrudan bir `int` / `long` / `double` döndürür (boş stream için `0`). `average()`, `min()`, `max()` ise `Optional<T>` yerine `OptionalInt` / `OptionalLong` / `OptionalDouble` döndürür -- primitive tipler için ayrı, kutulanmamış (unboxed) `Optional` varyantları. Bu metotlar, genel `Stream<Integer>`'da doğrudan bulunmaz; `IntStream`'in var olma sebeplerinden biri tam olarak budur.

Boxing ve Unboxing: mapToObj() ve boxed()

`mapToObj()`, bir primitive stream'i (örneğin `IntStream`) herhangi bir nesne tipinde bir `Stream<T>`'e çevirir. `boxed()`, aynı yönde ama özel bir hali: `IntStream`'i doğrudan `Stream<Integer>`'a çevirir -- primitive değerleri ilgili kutulanmış (boxed) tipe sarar. `collect()` / `Collectors` (bir önceki ders) gibi yalnızca nesne stream'leriyle çalışan API'lere geçerken sıkça ihtiyaç duyulur.

Object Stream'den Primitive Stream'e: mapToInt(), mapToLong(), mapToDouble()

`boxed()`'in tersi yöndeki köprü: `mapToInt()`, `mapToLong()`, `mapToDouble()`, bir `Stream<T>`'i ilgili primitive stream'e çevirir -- genellikle bir nesne stream'inde `sum()` / `average()` gibi sayısal bir toplama yapmak istediğinizde kullanılır.

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

// mapToInt() converts a Stream<T> into an IntStream -- typically to run an aggregate
// like sum()/average() that a plain object Stream doesn't offer. boxed() goes the other
// way, wrapping each primitive back into its object type (int -> Integer), needed
// whenever an API requires a Stream<Integer> instead of an IntStream.
class BoxingMapToIntExample {
    public static void main(String[] args) {
        List<String> names = List.of("Ahmet", "Mehmet", "Ayse");

        int totalLength = names.stream().mapToInt(String::length).sum();
        System.out.println(totalLength);

        double averageLength = names.stream().mapToInt(String::length).average().orElse(0);
        System.out.println(averageLength);

        // boxed(): IntStream -> Stream<Integer>, needed for object-based APIs like
        // collect() with Collectors (the previous lesson).
        List<Integer> lengths = names.stream().mapToInt(String::length).boxed().toList();
        System.out.println(lengths);

        // mapToObj(): the reverse direction, IntStream -> Stream<T> for any T, not just
        // the boxed wrapper type.
        Stream<String> labeled = IntStream.rangeClosed(1, 3).mapToObj(n -> "item-" + n);
        System.out.println(labeled.toList());
    }
}
```

Beklenen Çıktı

```
15
5.0
[5, 6, 4]
[item-1, item-2, item-3]
```

Parallel Stream Nedir? parallelStream() ve stream().parallel()

Bir `Collection`'ın `parallelStream()` metodu (ya da herhangi bir stream üzerinde `.parallel()` çağırmak), pipeline'ın işini tek bir thread yerine, ortak `ForkJoinPool`'daki birden çok thread'e böler. Sonuç, birleştirme (associative) bir işlem için **aynıdır** -- yalnızca çalışma stratejisi değişir. Aşağıdaki örnek, hem sonucun aynı kaldığını hem de gerçekten birden fazla thread'in kullanıldığını (thread isimlerini toplayarak) doğrudan gözlemliyor.

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

// parallelStream() (on a Collection) or stream().parallel() splits the pipeline's work
// across multiple threads from the common ForkJoinPool, instead of running it on a
// single thread. For an associative operation like sum(), the result is identical --
// only the execution strategy changes.
class ParallelBasicsExample {
    public static void main(String[] args) {
        List<Integer> numbers = IntStream.rangeClosed(1, 1_000_000).boxed().toList();

        long sequentialSum = numbers.stream().mapToLong(Integer::longValue).sum();
        long parallelSum = numbers.parallelStream().mapToLong(Integer::longValue).sum();
        System.out.println(sequentialSum == parallelSum);

        // Proof that multiple threads are actually used: collect the distinct thread
        // names that touched the pipeline.
        Set<String> threadNames = numbers.parallelStream()
                .map(n -> Thread.currentThread().getName())
                .collect(Collectors.toSet());
        System.out.println(threadNames.size() > 1);
    }
}
```

Sıralama: forEach() vs forEachOrdered()

Paralel bir stream'de `forEach()`, elemanları **karşılaşma sırasına göre değil**, hangi thread hangi elemanı ne zaman işlerse o sırayla işler -- sıra garantisi yoktur. `forEachOrdered()`, sonucu tekrar karşılaşma sırasına zorlar, ama bunun bir bedeli vardır: paralelliğin sağladığı hız kazancının büyük kısmından vazgeçilir. Aşağıdaki örnek, aynı 10 elemanlı listede `forEach()`'in gerçekten sırayı bozduğunu, `forEachOrdered()`'ın ise korduğunu doğrudan gözlemliyor.

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

// forEach() on a parallel stream does NOT guarantee encounter order -- elements are
// processed by whichever thread picks them up, in whatever order that happens.
// forEachOrdered() forces processing back into encounter order, at the cost of giving
// up most of the parallelism benefit.
class ParallelOrderingExample {
    public static void main(String[] args) {
        List<Integer> numbers = IntStream.rangeClosed(1, 10).boxed().toList();

        List<Integer> unordered = new CopyOnWriteArrayList<>();
        numbers.parallelStream().forEach(unordered::add);
        System.out.println(unordered.equals(numbers));

        List<Integer> ordered = new CopyOnWriteArrayList<>();
        numbers.parallelStream().forEachOrdered(ordered::add);
        System.out.println(ordered.equals(numbers));
    }
}
```

Yaygın Bir Tuzak: Thread-Safe Olmayan Paylaşılan Durum

Paralel bir `forEach()` içinde, sıradan (thread-safe olmayan) bir `ArrayList` gibi paylaşılan bir yapıya yazmak, gerçek bir veri yarışına (race condition) yol açar. Aşağıdaki örnek bunu 100.000 elemanlı bir listeyle gösteriyor: `ArrayList::add`'e paralel olarak yazmak **hiçbir istisna fırlatmadan**, sessizce, beklenenden daha az elemanla sonuçlanabiliyor -- gerçek çalıştırmalarda gözlemlenen boyutlar 96.901 ile 100.000 arasında değişti (bazı çalıştırmalarda şans eseri doğru çıktı, bu da hatayı daha da tehlikeli kılıyor). Doğru çözüm, `collect(Collectors.toList())` kullanmaktır -- thread-safety'yi kendi içinde, sizin kodunuza hiçbir paylaşılan durum sızdırmadan halleder.

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

// A classic parallel-stream pitfall: using forEach() to add into an ordinary,
// NOT-thread-safe ArrayList. Multiple threads calling add() on the same ArrayList at
// the same time can corrupt its internal state -- the result size may come out wrong,
// or an exception may be thrown, depending on timing (a real, observable data race).
// The fix: use a proper collector, which handles thread-safety internally.
class ParallelPitfallExample {
    public static void main(String[] args) {
        List<Integer> source = IntStream.rangeClosed(1, 100_000).boxed().toList();

        List<Integer> unsafe = new ArrayList<>();
        try {
            source.parallelStream().forEach(unsafe::add);
            System.out.println("no exception, size: " + unsafe.size() + " (expected " +
source.size() + ")");
        } catch (Exception e) {
            System.out.println("exception: " + e.getClass().getSimpleName());
        }

        // The safe fix: collect() handles combining thread-local partial results
        // correctly, with no shared mutable state exposed to your own code.
        List<Integer> safe = source.parallelStream().collect(Collectors.toList());
        System.out.println("safe size: " + safe.size());
    }
}
```

Ne Zaman Kullanılmalı?

Paralel stream'ler, şu koşullar bir arada olduğunda fayda sağlar: veri kümesi yeterince büyük (binlerce/milyonlarca eleman), işlem CPU-yoğun (her eleman için gerçek hesaplama gerektiriyor, yalnızca hızlı bir I/O beklemesi değil), ve işlem **birleştirilebilir/durumsuz** (associative/stateless) -- her elemanın işlenme sırası ya da diğer elemanlarla paylaşılan bir durum sonucu etkilememeli.

Neden Her Zaman Daha Hızlı Değildir?

Paralelleştirmenin gerçek bir maliyeti vardır: işi bölmek, thread'leri `ForkJoinPool` üzerinden koordine etmek, ve kısmi sonuçları birleştirmek zaman alır. Küçük bir veri kümesinde ya da ucuz bir işlemde, bu maliyet kazançtan fazla olabilir.

Bunu, tek seferlik bir `nanoTime()` ölçümüyle **doğru** göstermek mümkün değildir -- JVM, JIT derleyicisi devreye girmeden önce kodu yorumlar (interpret eder), bu yüzden hangi yol **önce** çalışırsa o, sırf ısınma (warmup) maliyeti yüzünden haksız yere yavaş görünür. Aşağıdaki örnek önce her iki yolu da binlerce kez çalıştırıp ısıtıyor, ancak ondan **sonra** gerçek bir

ölçüm alıyor -- bu sandbox'ta 100 elemanlık küçük bir liste için tipik bir çalıştırmada sıralı yol yaklaşık 15ms, paralel yol yaklaşık 41ms sürdü (tam sayılar çalıştırmadan çalıştırmaya değişir, ama küçük veri/ucuz işlem için sıralı yolun kazandığı yön tutarlı).

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

// Parallel streams have real overhead: splitting the work, coordinating threads via
// the ForkJoinPool, and merging partial results all cost time. For a small dataset or
// a cheap operation, that overhead can easily outweigh the benefit.
//
// A NAIVE one-shot nanoTime() comparison is misleading, though -- the JVM interprets
// code before its JIT compiler kicks in, so whichever path runs FIRST is unfairly
// slowed down by warmup cost, not by sequential-vs-parallel execution itself. This
// example runs each path repeatedly first (to let the JIT warm up), THEN times a
// later iteration -- the only way to get a measurement that means anything.
class ParallelOverheadExample {
    public static void main(String[] args) {
        List<Integer> smallList = IntStream.rangeClosed(1, 100).boxed().toList();

        // Warm up both paths so neither is penalized for still being interpreted.
        for (int i = 0; i < 10_000; i++) {
            smallList.stream().mapToLong(Integer::longValue).sum();
            smallList.parallelStream().mapToLong(Integer::longValue).sum();
        }

        long startSequential = System.nanoTime();
        for (int i = 0; i < 10_000; i++) {
            smallList.stream().mapToLong(Integer::longValue).sum();
        }
        long sequentialNanos = System.nanoTime() - startSequential;

        long startParallel = System.nanoTime();
        for (int i = 0; i < 10_000; i++) {
            smallList.parallelStream().mapToLong(Integer::longValue).sum();
        }
        long parallelNanos = System.nanoTime() - startParallel;

        System.out.println("sequential, 10000 runs: " + (sequentialNanos / 1_000_000) +
"ms");
        System.out.println("parallel, 10000 runs: " + (parallelNanos / 1_000_000) + "ms");
        System.out.println("sequential was faster: " + (sequentialNanos < parallelNanos));
    }
}
```

Best Practices

- **Varsayılan olarak sıralı (`stream()`) kullanın, yalnızca ölçtükten sonra paralele geçin.** "Ne Zaman Kullanılmalı?" bölümündeki koşullar sağlanmıyorsa, `parallelStream()` genellikle ek karmaşıklık getirir, performans kazandırmaz.
- **Paylaşılan, thread-safe olmayan bir yapıya asla paralel `forEach()` içinde yazmayın** -- bunun yerine her zaman bir `collect()` / `Collectors` kullanın (bir sonraki bölümde gerçek bir örnekle gösteriliyor).

- **Sıralamanın önemli olduğu yerlerde `forEachOrdered()` kullanın** -- ama bunun paralelliğin çoğu faydasını iptal ettiğini bilerek; sıralama gerekiyorsa çoğu zaman `stream()` (sıralı) zaten daha basit bir seçimdir.
- **Gerçek performans iddialarını her zaman ısıtılmış (warmed-up), tekrarlı bir ölçümle destekleyin** -- tek seferlik bir `nanoTime()` farkı yanıltıcı olabilir.

Yaygın Hatalar

- **Paralel `forEach()` içinde thread-safe olmayan bir koleksiyona (`ArrayList` gibi) eleman eklemek.** Bu, gerçek bir veri yarışı (race condition) oluşturur -- sonuç boyutu beklenenden **sessizce** küçük çıkabilir, hiçbir istisna fırlatılmadan (aşağıdaki örnekte gerçekten gözlemlendi: bazı çalıştırmalarda 100.000 beklenen elemandan yalnızca ~96.900-99.200'ü sonuca ulaştı). Hatanın her zaman değil, yalnızca bazen ortaya çıkması, bu hatayı daha da tehlikeli kılar -- testlerde fark edilmeyebilir.
- **"Daha çok thread, her zaman daha hızlı" varsayımı.** "Neden Her Zaman Daha Hızlı Değildir?" bölümünde görüldüğü gibi, küçük veri/ucuz işlem için paralel stream genellikle daha yavaştır.
- **Paralel bir stream'de sıralamaya güvenmek.** `forEach()` sırayı korumaz; sıra gerekiyorsa `forEachOrdered()` kullanılmalı (düz, sıralı bir `stream()` üzerinde ek bir maliyeti yoktur).
- **`IntStream` / `LongStream` / `DoubleStream`'i gerekmediği yerde kullanmak.** Yalnızca birkaç eleman varsa ya da sayısal bir toplama yapılmıyorsa, autoboxing maliyeti önemsizdir; gereksiz `mapToInt()` / `boxed()` zincirleri kodu karmaşıklaştırır.

Özet, Cheat Sheet ve Terimler Sözlüğü

Primitive stream'ler (`IntStream` , `LongStream` , `DoubleStream`), autoboxing maliyetini ortadan kaldırır ve `sum()` / `average()` / `max()` / `min()` gibi sayısal metotları doğrudan sunar; `mapToInt()` / `mapToLong()` / `mapToDouble()` bir nesne stream'inden primitive stream'e, `boxed()` / `mapToObj()` ters yöne köprü kurar. Paralel stream'ler (`parallelStream()` / `.parallel()`), bir pipeline'ın işini birden çok thread'e böler -- büyük veri ve CPU-yoğun, birleştirilebilir işlemler için faydalıdır, ama gerçek bir maliyeti vardır ve thread-safe olmayan paylaşılan durumla birlikte kullanıldığında sessiz veri yarışlarına yol açabilir.

Hızlı referans:

```
IntStream.range(0, 5)          // 0..4, bitiş hariç
IntStream.rangeClosed(0, 5)      // 0..5, bitiş dahil
IntStream.of(1, 2, 3)              // literal değerler

intStream.sum() / .average() / .max() / .min()   // sayısal toplamalar

stream.mapToInt(fn)                    // object -> primitive stream
intStream.boxed() / .mapToObj(fn)        // primitive -> object stream

collection.parallelStream()               // paralel çalışma
stream.forEach(x -> ...)                    // sıra garantisi yok (paralelde görünür)
stream.forEachOrdered(x -> ...)               // sıra garantili
```

Terimler Sözlüğü

Primitive stream — `int` / `long` / `double` gibi bir primitive tip için özelleşmiş, autoboxing maliyeti olmayan stream tipi (`IntStream` , `LongStream` , `DoubleStream`).

Autoboxing — Bir primitive değerin (`int`) otomatik olarak karşılık gelen nesne tipine (`Integer`) sarılması.

Parallel stream — Pipeline'ın işini ortak `ForkJoinPool` 'daki birden çok thread'e bölen stream.

Race condition (veri yarışı) — Birden fazla thread'in, senkronizasyon olmadan aynı paylaşılan duruma aynı anda yazmasından kaynaklanan, öngörülemez ve genellikle sessiz hata.

Warmup (ısınma) — Bir JVM'in JIT derleyicisinin, sık çalıştırılan kodu makine koduna derlemesi için gereken tekrarlı çalıştırma süresi; ısıtılmamış bir ölçüm yanıltıcı sonuçlar verebilir.