Primitive & Parallel Streams

Avoiding autoboxing cost with IntStream/LongStream/DoubleStream, the boxed() /mapToInt() bridges. Running in parallel with parallelStream(), the forEach() vs. forEachOrdered() ordering difference, a non-thread-safe shared state pitfall, when to use it, and why it is not always faster.

Intermediate 25 min

Last updated: November 5, 2026
Read online: https://www.learnforgex.com/en/topics/primitive-parallel-streams

---

Primitive & Parallel Streams

This is the final topic in the **Functional Interfaces & Streams** category. It brings together two separate but related subjects: streams specialized for primitive types like `int` / `long` / `double`, and parallel streams, which spread a pipeline's work across multiple threads.

What Is a Primitive Stream?

A `Stream<Integer>` holds each element as an `Integer` **object** -- every `int` value gets automatically wrapped (autoboxed) into an object. `IntStream` (and its counterparts `LongStream`, `DoubleStream`) are specialized stream types that skip that wrapping entirely, working directly on primitive values.

Why Does It Exist?

Autoboxing isn't free -- wrapping every `int` into an `Integer` object means extra memory allocation and a layer of indirection. Across a stream with millions of elements, that cost becomes noticeable. `IntStream` / `LongStream` / `DoubleStream` eliminate it entirely, and also expose methods that only make sense for numbers -- `sum()`, `average()` -- which a plain `Stream<T>` doesn't have directly.

History

Primitive stream types arrived alongside the Stream API in Java 8 (2014) -- the result of a deliberate decision by the designers to make avoiding autoboxing cost a core part of the API. There are three types: `IntStream`, `LongStream`, `DoubleStream` -- there's no separate stream type for `short`, `byte`, or `float`; those are widened to `int` / `double` when needed.

Creating an IntStream: range(), rangeClosed(), of()

`IntStream.range(start, end)` produces a range that **excludes** the end (`[start, end)`); `IntStream.rangeClosed(start, end)` **includes** it. `IntStream.of(...)` builds a stream from literal values -- the primitive counterpart of `Stream.of(...)`. The example prints the ranges with `forEachOrdered()` because only it guarantees encounter order; why plain `forEach()` does not is covered in the "Ordering: forEach() vs. forEachOrdered()" section.

```
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);
    }
}
```

Expected Output

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

Methods Specific to Primitive Streams: sum(), average(), max(), min()

`sum()` returns a plain `int` / `long` / `double` directly (`0` for an empty stream). `average()` , `min()` , and `max()` return `OptionalInt` / `OptionalLong` / `OptionalDouble` instead of `Optional<T>` -- separate, unboxed `Optional` variants for primitive types. These methods aren't directly available on a plain `Stream<Integer>` ; this is exactly one of the reasons `IntStream` exists.

Boxing and Unboxing: mapToObj() and boxed()

`mapToObj()` converts a primitive stream (like `IntStream`) into a `Stream<T>` of any object type. `boxed()` goes the same direction but is a special case: it converts an `IntStream` directly into a `Stream<Integer>` -- wrapping each primitive value into its corresponding boxed type. This comes up often when moving into an API that only works with object streams, like `collect()` / `Collectors` (the previous lesson).

From an Object Stream to a Primitive Stream: mapToInt(), mapToLong(), mapToDouble()

The bridge in the opposite direction from `boxed()` : `mapToInt()` , `mapToLong()` , and `mapToDouble()` convert a `Stream<T>` into the corresponding primitive stream -- typically used when you want a numeric aggregate like `sum()` / `average()` from an object stream.

```
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());
    }
}
```

Expected Output

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

What Is a Parallel Stream? parallelStream() and stream().parallel()

A `Collection`'s `parallelStream()` method (or calling `.parallel()` on any stream) splits the pipeline's work across multiple threads in the common `ForkJoinPool`, instead of a single thread. For an associative operation, the result is **identical** -- only the execution strategy changes. The example below directly observes both that the result stays the same and that multiple threads are genuinely used (by collecting the thread names).

```
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);
    }
}
```

Ordering: forEach() vs. forEachOrdered()

On a parallel stream, `forEach()` processes elements **not** by encounter order, but in whatever order each thread happens to pick them up -- there's no ordering guarantee. `forEachOrdered()` forces the result back into encounter order, at a cost: it gives up most of the speed benefit parallelism provides. The example below directly observes, on the same 10-element list, that `forEach()` genuinely breaks the order while `forEachOrdered()` preserves it.

```
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));
    }
}
```

A Common Pitfall: Non-Thread-Safe Shared State

Writing into a plain (non-thread-safe) structure like an `ArrayList` from inside a parallel `forEach()` creates a genuine data race. The example below demonstrates this with a 100,000-element list: writing to `ArrayList::add` in parallel can, **without throwing any exception**, silently produce fewer elements than expected -- real runs observed sizes ranging from 96,901 to the full 100,000 (some runs happened to come out correct by luck, which makes the bug even more dangerous). The correct fix is `collect(Collectors.toList())` -- it handles thread-safety internally, without leaking any shared state into your own code.

```
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());
    }
}
```

When Should You Use It?

Parallel streams pay off when all of these hold together: the dataset is large enough (thousands or millions of elements), the operation is CPU-intensive (real computation per element, not just a fast I/O wait), and the operation is **associative/stateless** -- the order elements are processed in, or any state shared across elements, must not affect the result.

Why Isn't It Always Faster?

Parallelizing has a real cost: splitting the work, coordinating threads through the `ForkJoinPool`, and merging partial results all take time. For a small dataset or a cheap operation, that cost can easily outweigh the benefit.

A single-shot `nanoTime()` measurement can't show this **correctly**, though -- the JVM interprets code before its JIT compiler kicks in, so whichever path runs **first** looks unfairly slow purely from warmup cost, not from sequential-vs-parallel execution itself. The example below runs both paths thousands of times first to warm them up, and only **then** takes a real measurement -- in a typical run on this sandbox, for a small 100-element list, the sequential path took about 15ms and the parallel path about 41ms (the exact numbers vary run to run, but the direction -- sequential winning for small data/cheap operations -- was consistent).

```
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

- **Default to sequential (`stream()`), and only switch to parallel after measuring.** If the conditions in "When Should You Use It?" aren't met, `parallelStream()` usually adds complexity without a performance gain.
- **Never write into a shared, non-thread-safe structure from inside a parallel `forEach()`** -- always use a `collect()` / `Collectors` instead (demonstrated with a real example above).
- **Use `forEachOrdered()` where order matters** -- but knowing it cancels out most of the benefit of parallelism; when order is required, plain sequential `stream()` is often the simpler choice anyway.

- **Back up any real performance claim with a warmed-up, repeated measurement** -- a single-shot `nanoTime()` difference can be misleading.

Common Mistakes

- **Adding elements to a non-thread-safe collection (like `ArrayList`) inside a parallel `forEach()`.** This creates a real race condition -- the result size can come out **silently** smaller than expected, with no exception thrown at all (genuinely observed in the example above: some runs delivered only about 96,900-99,200 of the expected 100,000 elements). The fact that the bug doesn't happen every time makes it even more dangerous -- it can slip past testing unnoticed.
- **Assuming "more threads always means faster."** As shown in "Why Isn't It Always Faster?", parallel streams are often slower for small data or cheap operations.
- **Relying on ordering in a parallel stream.** `forEach()` doesn't preserve it; use `forEachOrdered()` if order is required (on a plain sequential `stream()` it costs nothing extra).
- **Using `IntStream` / `LongStream` / `DoubleStream` where it isn't needed.** With only a handful of elements, or no numeric aggregation involved, the autoboxing cost is negligible; unnecessary `mapToInt()` / `boxed()` chains just add complexity.

Summary, Cheat Sheet, and Glossary

Primitive streams (`IntStream` , `LongStream` , `DoubleStream`) eliminate autoboxing cost and expose numeric methods like `sum()` / `average()` / `max()` / `min()` directly; `mapToInt()` / `mapToLong()` / `mapToDouble()` bridge from an object stream to a primitive stream, and `boxed()` / `mapToObj()` bridge back the other way. Parallel streams (`parallelStream()` / `.parallel()`) split a pipeline's work across multiple threads -- useful for large, CPU-intensive, associative operations, but they have a real cost and can produce silent data races when combined with non-thread-safe shared state.

Quick reference:

```
IntStream.range(0, 5)            // 0..4, end excluded
IntStream.rangeClosed(0, 5)        // 0..5, end included
IntStream.of(1, 2, 3)                // literal values

intStream.sum() / .average() / .max() / .min()     // numeric aggregates

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

collection.parallelStream()                  // run in parallel
stream.forEach(x -> ...)                       // no ordering guarantee (visible in
parallel)
stream.forEachOrdered(x -> ...)                  // ordering guaranteed
```

Glossary

Primitive stream — A stream type specialized for a primitive type like `int` / `long` / `double`, with no autoboxing cost (`IntStream`, `LongStream`, `DoubleStream`).

Autoboxing — Automatically wrapping a primitive value (`int`) into its corresponding object type (`Integer`).

Parallel stream — A stream that splits its pipeline's work across multiple threads in the common `ForkJoinPool`.

Race condition — An unpredictable, often silent bug caused by multiple threads writing to the same shared state without synchronization.

Warmup — The repeated-execution time a JVM's JIT compiler needs before it compiles frequently-run code into machine code; an un-warmed-up measurement can be misleading.