Stream API Fundamentals

The fundamentals of the Stream API: what a Stream is, moving from a Collection to a Stream, the three stages of a pipeline (source/intermediate/terminal). Intermediate operations: filter(), map(), flatMap(), distinct(), sorted(), peek(), limit(), skip(). Lazy evaluation and a stream's single-use nature.

Intermediate 25 min

Last updated: November 10, 2026
Read online: https://www.learnforgex.com/en/topics/stream-fundamentals

---

Stream API Fundamentals & Intermediate Operations

The last two lessons covered lambda syntax and the ready-made interfaces in `java.util.function`. This lesson arrives at the reason they exist in the first place: the **Stream API**. It's the way to express operations on a collection -- "filter this, transform this, sort it like that" -- as a declarative chain, without writing a `for` loop.

What Is a Stream?

A `Stream<T>` is a **pipeline** that processes elements from a data source (usually a `Collection`) in sequence. The critical point: a stream doesn't **store** data. It isn't a data structure like a list or a set -- it's a single-use pipe that lets you pass over the source data exactly once.

A stream pipeline has three parts: a **source** (like `list.stream()`), zero or more **intermediate operations** (like `filter()` and `map()` -- this lesson's topic), and exactly one **terminal operation** (like `toList()` and `forEach()` -- the next lesson's topic).

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

Expected Output

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

Why Does It Exist?

Before Java 8, filtering and transforming a collection meant a hand-written `for` loop, a temporary result list, and `if` checks inside the loop -- imperative code that spells out **how** to do it, step by step. The Stream API lets you express the same work as a declarative chain that describes **what** you want: `filter(...).map(...).toList()` tells the reader your intent directly, not the loop mechanics.

Here is that difference in a small example:

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

Here, `filter` expects a `Predicate<String>` (the `Predicate` from the "Built-in Functional Interfaces" lesson), `map` expects a `Function<String,String>` (the same lesson's `Function`), and `name -> name.startsWith("A")` and `String::toUpperCase` are a lambda and a method reference respectively (the "Lambda Expressions" and "Built-in Functional Interfaces" lessons). This chain is exactly where the last three lessons -- the functional interface foundation from "Interface", "Lambda Expressions", "Built-in Functional Interfaces" -- come together.

History

The Stream API arrived alongside the `java.util.function` package in Java 8 (2014). The two are tightly coupled: Stream API methods like `filter()`, `map()`, and `reduce()` expect parameters of exactly the types in `java.util.function` (`Predicate`, `Function`, `BinaryOperator`). Without the Stream API, most of these interfaces wouldn't be used nearly as often; without these interfaces, the Stream API's methods couldn't be defined in a type-safe way.

From a Collection to a Stream: stream() and of()

The most common source is any `Collection`'s (`List`, `Set`, ...) `stream()` method. Beyond that, `Stream.of(...)` builds a stream from literal values, `Arrays.stream(array)` from an array, and `Stream.iterate(...)` from a generation rule -- since `iterate()` has no natural end, it's usually bounded with `limit()`.

The Stream Pipeline: Source, Intermediate, Terminal

A stream pipeline has three stages: the **source** determines where data comes from, the **intermediate operations** (this lesson's topic -- `filter`, `map`, `flatMap`, `distinct`, `sorted`, `peek`, `limit`, `skip`) transform the data step by step, and exactly one **terminal operation** (next lesson) triggers the pipeline and produces a result. Every intermediate operation returns a `Stream` -- that's what makes method chaining possible.

filter(): Filtering

`filter(Predicate<T>)` keeps only the elements matching the given condition; the stream can get **shorter**, but element type doesn't change. `Predicate` was covered in detail in the "Built-in Functional Interfaces" lesson -- here it's used directly.

map() and flatMap(): Transforming and Flattening

`map(Function<T,R>)` transforms each element into exactly one other value; the stream's **length doesn't change**, but element type/value can.

`flatMap()` solves a trap `map()` falls into: if the mapping function itself returns a `Stream` /collection, `map()` produces a "stream of streams" -- an awkward, nested structure. `flatMap()` turns each element into a stream and **merges those streams into a single flat stream**. Typical use: flattening a "list of lists" into a single list, or splitting each sentence into words and collecting all of them into one flat list of words.

```
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);
    }
}
```

Expected Output

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

distinct(), sorted(), peek()

`distinct()` removes duplicate elements based on `equals()`. `sorted()` orders elements either by natural ordering (`Comparable`) or by a given `Comparator`. `peek()` runs a `Consumer` on each element **without changing** the stream -- it exists purely to observe, most often for debugging; relying on `peek()` for a side effect in production code isn't recommended (see Common Mistakes).

```
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() and skip()

`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)`.

```
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);
    }
}
```

Expected Output

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

Lazy Evaluation: When Do Intermediate Operations Actually Run?

Intermediate operations are **lazy**: calling `filter()` or `map()` doesn't run anything yet -- it just adds a step to the pipeline's description. Real work only starts once a **terminal operation** is called -- and even then, it proceeds element by element, in a single pass (each element flows through every intermediate operation in turn before the next element starts).

A stream is also **single-use**: once a terminal operation runs, the stream is closed, and trying to reuse the same stream reference throws `IllegalStateException`.

```
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

- **Keep the chain small and readable.** Putting one operation per line (`filter` on its own line, `map` on its own line) makes a chain easy to scan.
- **Apply `filter()` as early as possible.** Placing a cheap `filter()` before an expensive `map()` means `map()` runs on fewer elements.
- **Use `peek()` only for observation/debugging**, not as part of production logic -- the next section spells out why.
- **Use a stream once, then let it go.** Don't store a stream in a variable and try to reuse it across multiple terminal operations; instead, build a fresh stream from the source (`list.stream()`) whenever you need one.

Common Mistakes

- **Using `peek()` to produce a side effect.** The documentation explicitly describes `peek()` as "primarily for debugging" -- JVM optimizations may skip `peek()` calls in some situations, so relying on it as a dependable side-effect mechanism is fragile.

- **Confusing `map()` and `flatMap()`.** If the transformation function returns a `Stream`/`List` and you used `map()`, you're left with a useless "stream of streams" -- what you needed was `flatMap()`.
- **Trying to reuse an already-consumed stream.** Applying another operation to the same `Stream` reference after a terminal operation has run throws `IllegalStateException` -- get a fresh stream from the source whenever you need one.
- **Calling `Stream.iterate()` without `limit()`.** Failing to bound a generation rule that has no natural end causes the pipeline to run forever (or until memory runs out).

Summary, Cheat Sheet, and Glossary

A `Stream` is a single-pass pipeline over a source that doesn't store data: a **source** (`collection.stream()`, `Stream.of()`, `Arrays.stream()`), zero or more **intermediate operations** (`filter`, `map`, `flatMap`, `distinct`, `sorted`, `peek`, `limit`, `skip` -- all lazy, all returning a `Stream`), and exactly one **terminal operation**. A stream is single-use; it can't be reused once consumed.

Quick reference:

```
list.stream()
    .filter(x -> ...)     // filter, stream can get shorter
    .map(x -> ...)          // transform, length unchanged
    .flatMap(x -> ...)      // transform + flatten
    .distinct()               // remove duplicates
    .sorted()                  // order elements
    .peek(x -> ...)             // observe, no change
    .limit(n)                    // first n elements
    .skip(n)                      // skip first n elements
```

Glossary

Stream — A single-use pipeline that processes elements from a data source in sequence, without storing data.

Source — Where a stream pipeline gets its data from (`collection.stream()`, `Stream.of()`, and similar).

Intermediate operation — A lazily-evaluated pipeline step that takes a `Stream` and returns a `Stream` (`filter`, `map`, `flatMap`, `distinct`, `sorted`, `peek`, `limit`, `skip`).

Terminal operation — The step that triggers the pipeline and produces a result, consuming the stream; covered in the next lesson.

Lazy evaluation — Intermediate operations doing no work until a terminal operation is called.

flatMap — An operation that turns each element into a stream and merges those streams into a single flat stream.