Java Streams Overview

---

1. Why Streams are Introduced

Streams were introduced in Java 8 to simplify collection processing and enable functional programming patterns. The goals were:

- **Declarative style:** Focus on *what* to do, not *how* to do it.
- **Lazy evaluation:** Process elements on-the-fly, saving memory.
- **Easy parallelism:** Simplify concurrent processing via `parallelStream()`.
- **Functional programming support:** Promote pure functions and chainable operations.
- **Reduce boilerplate:** Less code compared to traditional loops for filtering, mapping, and collecting data.

Example:

```
List<String> words = List.of("apple", "bat", "carrot");
List<String> result = words.stream()
                           .filter(w -> w.length() > 3)
                           .map(String::toUpperCase)
                           .toList();
System.out.println(result); // [APPLE, CARROT]
```

---

2. Pros and Cons of Streams

Pros: 1. **Declarative and readable** - simpler than traditional loops. 2. **Encourages immutability** - less chance of side-effects. 3. **Chainable and composable** - filter → map → reduce pipelines. 4. **Parallel-friendly** - `parallelStream()` allows concurrent processing easily. 5. **Memory-efficient** - lazy evaluation avoids storing intermediate results.

Cons: 1. **Harder to debug** - complex pipelines can be tricky. 2. **Overhead for small datasets** - streams can be slightly slower than loops. 3. **Limited flexibility** - breaking early from a loop or modifying elements inline can be difficult. 4. **Learning curve** - functional programming concepts may take time. 5. **Readability issues** - very long chains may become cryptic.

---

3. How Streams Are Immutable

Streams do **not modify the source collection** or the elements unless explicitly programmed. They operate as a **pipeline of transformations** that produce a view of the data.

Example:

```java
List<String> words = new ArrayList<>(List.of("apple", "bat", "carrot"));
Stream<String> stream = words.stream()
                             .filter(w -> w.length() > 3)
                             .map(String::toUpperCase);

System.out.println(words); // [apple, bat, carrot]
```

- `words` list remains unchanged.
- Intermediate operations (`filter`, `map`) create a **new pipeline**, not a new list.
- Streams are **single-use**; once a terminal operation (`collect`, `forEach`) is called, the stream is consumed.
- Functional immutability ensures elements are not altered unless explicitly done in operations like `peek()`.

Additional Notes

- **Lazy evaluation:** Streams only process elements when a terminal operation is invoked. This saves memory and improves performance.
- **Spliterator:** Internal component that splits elements for sequential or parallel processing.
- **Parallel processing:** Streams can process multiple elements concurrently without manual thread management.

Example of lazy evaluation:

```java
Stream<String> s = words.stream().filter(w -> {
    System.out.println("Filtering: " + w);
    return w.length() > 3;
});
// No output yet
s.forEach(System.out::println); // Filtering happens here, lazily
```

Conclusion: Streams provide a **clean, efficient, and parallel-ready way** to process collections in Java, with emphasis on immutability, functional style, and reduced boilerplate.