Java Streams: Execution, Laziness, Spliterator, and Memory Efficiency

1■■ How Stream Operations Get Executed

A stream pipeline has three parts:

1 Source → where data comes from (List, Set, Array, I/O, etc.)
2 Intermediate operations → transformations (filter, map, sorted, etc.) - Lazy
3 Terminal operation → triggers execution (forEach, collect, reduce)

Example:

```
List<String> words = List.of("apple", "bat", "carrot");
words.stream()
.filter(w -> w.length() > 3)
.map(String::toUpperCase)
.forEach(System.out::println);
```

2■■ Why Streams Are Lazy

Intermediate operations don't execute immediately. They create a pipeline description. Only when a terminal operation is invoked, execution begins.
Example:

```
Stream<String> s = Stream.of("one", "two", "three")
.filter(w -> {
System.out.println("Filtering " + w);
return w.length() > 3;
});
System.out.println("Stream defined, nothing executed yet!");

s.forEach(System.out::println);
```

3■■ What is Spliterator

Spliterator = Split-able Iterator. It traverses and optionally splits elements for parallel processing.
Example:

```
List<String> words = List.of("apple", "bat", "carrot");
Spliterator<String> spliterator = words.spliterator();
while(spliterator.tryAdvance(System.out::println)) { }
```

4■■ How Streams Are Memory Efficient

Streams don't store transformed values. Each element is processed on the fly and passed to the terminal operation.
Example without streams (extra memory used):

```
List<String> temp = new ArrayList<>();
for (String w : words) {
if (w.length() > 3) {
temp.add(w.toUpperCase());
}
}
temp.forEach(System.out::println);
```

Example with streams (memory efficient):

```
words.stream()
.filter(w -> w.length() > 3)
```

```
.map(String::toUpperCase)
.forEach(System.out::println);
```

■ Summary

1 Execution → Streams pull elements one by one from source through pipeline.
2 Lazy → Intermediate ops don’t execute until terminal op.
3 Spliterator → Engine that traverses/splits elements.
4 Memory efficient → No storage of intermediate results.