JAVA STREAMS

Where Code Flows and Logic Bends

◆ *Java Streams: Where Simplicity Meets Power*

Think you know Java? Streams will challenge that.

With just a few lines of code, Java Streams can replace complex loops, unlock parallelism, and turn data processing into elegant expressions. It's not just about shorter code — it's about **thinking differently**.

Get ready to write logic that's clean, fast, and functional. From chaining operations to embracing immutability and lazy evaluation, Streams reward those who master their flow — and punish those who don't.

Simple in syntax. Brutal in depth.
Welcome to the thinking developer's API.

📜 Java Stream Methods

Streams are introduced in **Java 8**. They allow processing of collections in a **functional style** chaining multiple operations together.

- Streams **don't store** data; they **process** data.
- Streams are **consumed** once — you cannot reuse a stream after a terminal operation.
- Stream operations can be **chained**.
- Prefer **parallel streams** only when it can **truly improve** performance (large data + non-thread blocking code).

Main Interfaces:

- `Stream<T>`
- `IntStream`, `LongStream`, `DoubleStream`

1. Creation of Streams

Method	Description	Example
`stream()`	Converts a collection into a sequential stream	`list.stream()`
`parallelStream()`	Converts collection into a parallel stream	`list.parallelStream()`
`Stream.of(...)`	Creates stream from values	`Stream.of(1, 2, 3)`
`Arrays.stream(array)`	Creates stream from an array	`Arrays.stream(new int[]{1,2,3})`

2. **Intermediate Operations** (returns a new Stream, **lazy evaluation**)

Method	Description	Example
`filter(Predicate)`	Select elements matching a condition	stream.filter(x -> x > 5)
`map(Function)`	Transform elements	stream.map(String::toUpperCase)
`flatMap(Function)`	Flattens nested structures	stream.flatMap(list -> list.stream())
`distinct()`	Removes duplicates (based on equals())	stream.distinct()
`sorted()`	Sorts elements (natural order)	stream.sorted()
`sorted(Comparator)`	Custom sorting	stream.sorted(Comparator.reverseOrder())
`limit(n)`	Limits stream to n elements	stream.limit(5)
`skip(n)`	Skips first n elements	stream.skip(3)
`peek(Consumer)`	Perform action without consuming	stream.peek(System.out::println)

Note: Intermediate operations are lazy — no processing happens until a terminal operation is called.

3. **Terminal Operations** (triggers stream processing)

Method	Description	Example
`collect(Collector)`	Collects elements into a collection	`stream.collect(Collectors.toList())`
`forEach(Consumer)`	Performs an action for each element	`stream.forEach(System.out::println)`
`toArray()`	Converts stream into array	`stream.toArray()`
`reduce(BinaryOperator)`	Combines elements into a single result	`stream.reduce(0, Integer::sum)`
`count()`	Counts number of elements	`stream.count()`
`min(Comparator)`	Smallest element based on comparator	`stream.min(Comparator.naturalOrder())`
`max(Comparator)`	Largest element based on comparator	`stream.max(Comparator.naturalOrder())`
`anyMatch(Predicate)`	True if **any** element matches	`stream.anyMatch(x -> x > 10)`
`allMatch(Predicate)`	True if **all** elements match	`stream.allMatch(x -> x > 0)`
`noneMatch(Predicate)`	True if **no** element matches	`stream.noneMatch(x -> x < 0)`
`findFirst()`	Returns first element (Optional)	`stream.findFirst()`
`findAny()`	Returns any element (useful in parallel)	`stream.findAny()`

4. **Collectors** (for collect())

- **Collectors.toList()** → Collects into a List
- **Collectors.toSet()** → Collects into a Set
- **Collectors.toMap(keyMapper, valueMapper)** → Collects into a Map
- **Collectors.groupingBy(Function)** → Groups elements by a key
- **Collectors.partitioningBy(Predicate)** → Partitions elements into two groups (true/false)

```
List<String> names = List.of("Alice", "Bob", "Charlie");
Map<Integer, List<String>> groupedByLength = names.stream()
    .collect(Collectors.groupingBy(String::length));
```

5. **Special Stream Types**

`IntStream`, `LongStream`, `DoubleStream`	**Streams for primitives (no boxing)**	**`IntStream.range(1,5)`**

Methods like `sum()`, `average()`, `min()`, `max()` are available directly on primitive streams.

🆚 Difference Between stream() and parallelStream()

Feature	stream()	parallelStream()
Processing	Sequential: one element at a time, in one thread (usually main thread).	Parallel: splits data into multiple chunks and processes them **simultaneously** using **multiple threads** (ForkJoinPool).
Speed	Good for small or simple datasets.	Can be faster for **large** datasets if system has multiple cores.
Threading	Sin6gle thread.	Multiple threads.
Order	Preserves the original order of elements.	Order is **not guaranteed** unless forced (e.g., forEachOrdered).
Performance	Simple and low overhead.	Adds overhead due to splitting and combining — benefits only when heavy work is done.
Usage Example	list.stream().filter(x -> x > 5).collect(...)	list.parallelStream().filter(x -> x > 5).collect(...)
Ideal Use Case	Small datasets, operations where order matters, I/O operations.	Large datasets, CPU-intensive operations, when order doesn't matter much.
Underlying Mechanism	Iterates items one by one.	Uses ForkJoinPool.commonPool internally to divide tasks.

Java Stream Practice Questions (with Solutions)

Q1: Given a list of integers, return a list of only even numbers.

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6);
```

Sol

```
List<List<Integer>> pairs = nums.stream()
    .flatMap(i -> nums.stream()
        .filter(j -> i < j && i + j == target)
        .map(j -> List.of(i, j)))
    .collect(Collectors.toList());

System.out.println(pairs); // Output: [[2, 8], [3, 7], [4, 6]]
```

Q2: From a list, find all pairs that sum to a given number (e.g., 10).

```
List<Integer> nums = List.of(1, 2, 3, 7, 5, 8, 6, 4);
int target = 10;
```

Sol

```
List<String> upperNames = names.stream()
    .map(String::toUpperCase)
    .collect(Collectors.toList());

System.out.println(upperNames); // Output: [ALICE, BOB, CHARLIE]
```

Q3: Find the first string that starts with letter "C".

```
List<String> names = List.of("Alice", "Bob", "Charlie", "David");
```

Sol

```
Optional<String> firstNameStartingWithC = names.stream()
    .filter(name -> name.startsWith("C"))
    .findFirst();

firstNameStartingWithC.ifPresent(System.out::println); // Output:
Charlie
```

Q4: Find the sum of squares of numbers in a list.

```
List<Integer> numbers = List.of(1, 2, 3, 4);
```

Sol

```
int sumOfSquares = numbers.stream()
    .map(n -> n * n)
    .reduce(0, Integer::sum);

System.out.println(sumOfSquares); // Output: 30 (1+4+9+16)
```

Q5: Sort a list of strings in descending (reverse alphabetical) order.

```
List<String> fruits = List.of("apple", "banana", "cherry", "date");
```

Sol

```
List<String> sortedFruits = fruits.stream()
    .sorted(Comparator.reverseOrder())
    .collect(Collectors.toList());

System.out.println(sortedFruits); // Output: [date, cherry, banana, apple]
```

Q6: Group words by their length.

```
List<String> words = List.of("one", "two", "three", "four", "five");
```

Sol

```
Map<Integer, List<String>> groupedByLength = words.stream()
    .collect(Collectors.groupingBy(String::length));

System.out.println(groupedByLength);
// Output: {3=[one, two], 5=[three], 4=[four, five]}
```

Q7: Find the maximum number in a list.

```
List<Integer> numbers = List.of(10, 20, 5, 80, 30);
```

Sol

```
Optional<Integer> maxNumber = numbers.stream()
    .max(Integer::compare);
maxNumber.ifPresent(System.out::println); // Output: 80
```

Q8: **Count how many strings start with "A".**

```
List<String> names = List.of("Alice", "Arnold", "Bob", "Charlie",
"Andrew");
```

Sol

```
long count = names.stream()
    .filter(name -> name.startsWith("A"))
    .count();

System.out.println(count); // Output: 3
```

Q9: **Given a list of strings, group them by anagram sets.**

```
List<String> words = List.of("listen", "silent", "enlist", "rat",
"tar", "art");
```

Sol

```
Map<String, List<String>> anagramGroups = words.stream()
    .collect(Collectors.groupingBy(
        word -> word.chars()
                    .sorted()
                    .mapToObj(c -> String.valueOf((char)c))
                    .collect(Collectors.joining())
    ));

// Output: {eilnst=[listen, silent, enlist], art=[rat, tar, art]}
```

Q10: Convert a list of lists into a single list.

```
List<List<String>> nestedList = List.of(
    List.of("a", "b"),
    List.of("c", "d"),
    List.of("e", "f")
);
```

Sol

```
List<String> flatList = nestedList.stream()
    .flatMap(Collection::stream)
    .collect(Collectors.toList());

System.out.println(flatList); // Output: [a, b, c, d, e, f]
```

Q11: **Given a list of integers, return a list of strings `"even"` or `"odd"` depending on whether the number is even or odd.**

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5);
```

Sol

```
List<String> evenOrOdd = numbers.stream()
    .map(n -> n % 2 == 0 ? "even" : "odd")
    .collect(Collectors.toList());

System.out.println(evenOrOdd); // Output: [odd, even, odd, even, odd]
```

Q12: **Given a list of sentences, count the frequency of each word (case-insensitive).**

```
List<String> sentences = List.of("Java is fun", "Streams are
powerful", "Java is powerful");
```

Sol

```
Map<String, Long> wordFreq = sentences.stream()
    .flatMap(sentence ->
Arrays.stream(sentence.toLowerCase().split("\\s+")))
    .collect(Collectors.groupingBy(word -> word,
Collectors.counting()));

// Output: {java=2, is=2, fun=1, streams=1, are=1, powerful=2}
```

Q13: **From a list of integers, find the duplicate numbers and how many times they occur.**

```
List<Integer> nums = List.of(1, 2, 3, 2, 3, 4, 5, 3);
```

Sol

```
Map<Integer, Long> duplicates = nums.stream()
    .collect(Collectors.groupingBy(Function.identity(),
Collectors.counting()))
    .entrySet().stream()
    .filter(e -> e.getValue() > 1)
    .collect(Collectors.toMap(Map.Entry::getKey,
Map.Entry::getValue));

// Output: {2=2, 3=3}
```

Q14: **Flatten a `Map<String, List<List<Integer>>>` into a `List<Integer>`.**

```
Map<String, List<List<Integer>>> map = Map.of(
    "a", List.of(List.of(1, 2), List.of(3)),
    "b", List.of(List.of(4), List.of(5, 6))
);
```

Sol

```
List<Integer> flatList = map.values().stream()
    .flatMap(List::stream)
    .flatMap(List::stream)
    .collect(Collectors.toList());

// Output: [1, 2, 3, 4, 5, 6]
```

Q15:

Return the common elements between two lists using streams.

Sol

```
List<Integer> common = list1.stream()
    .filter(list2::contains)
    .collect(Collectors.toList());
```

Q16: **Remove duplicate integers from a list.**
List<Integer> numbers = List.of(1, 2, 2, 3, 4, 4, 5);

Sol

```
List<Integer> uniqueNumbers = numbers.stream()
    .distinct()
    .collect(Collectors.toList());

System.out.println(uniqueNumbers); // Output: [1, 2, 3, 4, 5]
```

Q17: **Given "hello world", count the frequency of each character.**

Sol

```
Map<Character, Long> charFreq = str.chars()
    .mapToObj(c -> (char) c)
    .filter(c -> c != ' ')
    .collect(Collectors.groupingBy(Function.identity(),
Collectors.counting()));
```

Q18: Given a list of strings, find the element that occurs most frequently.

List<String> input = List.of("apple", "banana", "apple", "orange", "banana", "apple");

Sol

```
String mostFrequent = input.stream()
    .collect(Collectors.groupingBy(Function.identity(),
Collectors.counting()))
    .entrySet().stream()
    .max(Map.Entry.comparingByValue())
    .map(Map.Entry::getKey)
    .orElse(null);

System.out.println(mostFrequent); // Output: apple
```

Q19: Given a list of lowercase strings, return the list of characters that appear in every string.

List<String> words = List.of("bella", "label", "roller");

Sol

```
List<Character> commonChars = words.stream()
    .map(word -> word.chars()
                    .mapToObj(c -> (char) c)
                    .collect(Collectors.groupingBy(c -> c,    Collectors.counting())))
    .reduce((map1, map2) -> {
        map1.keySet().retainAll(map2.keySet());
        map1.replaceAll((k, v) -> Math.min(v, map2.get(k)));
        return map1;
    })
    .orElse(Map.of()).entrySet().stream()
    .flatMap(e -> Collections.nCopies(e.getValue().intValue(), e.getKey()).stream())
    .collect(Collectors.toList());

// Output: [e, l, l]
```

Q20: Reverse a list of elements using streams only.

Sol

```
List<Integer> reversed = IntStream.range(0, list.size())
    .mapToObj(i -> list.get(list.size() - i - 1))
    .collect(Collectors.toList());
```