Java Stream API — Advanced Level (71–100)

This document provides coding approaches, explanations and Java Stream API snippets for advanced interview questions (71–100). Each item typically includes 1–2 stream-centric approaches and brief notes on when to use them. Replace placeholder classes (e.g., Employee, Transaction) with your project classes when running code.

71. Parallel stream to improve performance in large datasets

Approach 1:

```
List<Integer> nums = IntStream.rangeClosed(1, 1_000_000).boxed().collect(Collectors.toList());
int sum = nums.parallelStream()
        .filter(n -> n % 2 == 0)
        .mapToInt(Integer::intValue)
        .sum();
```

Why it works / Notes: Parallelizes processing across available cores. Good for CPU-heavy operations and large data; beware of shared mutable state and order requirements.

Approach 2 (Alternative):

```
int sum2 = IntStream.rangeClosed(1, 1_000_000).parallel().filter(n -> n % 2 == 0).sum();
```

Why it works / Notes: Using primitive stream parallel() avoids boxing overhead and can be faster.

72. Custom comparator with null handling in sorting

Approach 1:

```
List<String> sorted = names.stream()
   .sorted(Comparator.nullsLast(Comparator.naturalOrder()))
   .collect(Collectors.toList());
```

Why it works / Notes: Comparator.nullsLast/First handle nulls cleanly. Use when list may contain nulls and order must be predictable.

Approach 2 (Alternative):

```
List<String> sorted2 = names.stream()
   .sorted(Comparator.comparing(Function.identity(),
Comparator.nullsFirst(Comparator.naturalOrder())))
   .collect(Collectors.toList());
```

Why it works / Notes: Explicit comparing with null-safe comparator offers same behavior but more explicit.

73. Implement Collector to sum even numbers only

Approach 1:

```
Collector<Integer, int[], Integer> evenSumCollector =
   Collector.of(
      () -> new int[1],
      (acc, n) -> { if (n % 2 == 0) acc[0] += n; },
      (a, b) -> { a[0] += b[0]; return a; },
      acc -> acc[0]
   );

int sumEven = Stream.of(1,2,3,4,5,6).collect(evenSumCollector);
```

Why it works / Notes: Custom Collector.of gives full control over accumulation and parallel combination. Use for custom reduction behaviours.

Approach 2 (Alternative):

```
int sumEven2 = Stream.of(1,2,3,4,5,6).filter(n -> n%2==0).mapToInt(Integer::intValue).sum();
```

Why it works / Notes: Often simpler to filter + mapToInt + sum unless custom state is required.

74. Merge multiple lists of employees into a single sorted list

Approach 1:

```
List<Employee> merged = Stream.of(list1, list2, list3)
   .flatMap(Collection::stream)
   .sorted(Comparator.comparing(Employee::getSalary).reversed())
   .collect(Collectors.toList());
```

Why it works / Notes: Stream.of + flatMap easily merges many lists. Sorting afterward gives global order.

Approach 2 (Alternative):

```
List<Employee> merged2 = Stream.concat(list1.stream(), Stream.concat(list2.stream(),
list3.stream()))
   .sorted(Comparator.comparing(Employee::getSalary).reversed())
   .collect(Collectors.toList());
```

Why it works / Notes: Stream.concat is fine for two lists at a time; Stream.of scales better for many lists.

75. Group employees by department and then by designation

Approach 1:

```
Map<String, Map<String, List<Employee>>> grouped =
   employees.stream()
           .collect(Collectors.groupingBy(Employee::getDepartment,
                   Collectors.groupingBy(Employee::getDesignation)));
```

Why it works / Notes: Nested groupingBy creates a two-level map: dept -> (designation -> list). Useful for multi-key aggregation.

Approach 2 (Alternative):

```
Map<String, Map<String, Long>> counts =
   employees.stream()
           .collect(Collectors.groupingBy(Employee::getDepartment,
                   Collectors.groupingBy(Employee::getDesignation, Collectors.counting())));
```

Why it works / Notes: Downstream collectors can compute aggregates (counts, sums) inside nested grouping.

76. Convert CSV data into a list of objects using streams

Approach 1:

```
List<Person> people = Files.lines(Path.of("data.csv"))
   .skip(1) // skip header
   .map(line -> line.split(","))
   .map(cols -> new Person(cols[0], Integer.parseInt(cols[1]), cols[2]))
   .collect(Collectors.toList());
```

Why it works / Notes: Files.lines returns a stream of lines; skip header, split, map to objects. Handle IOExceptions and malformed rows.

Approach 2 (Alternative):

```
try (Stream<String> lines = Files.lines(path)) {
   List<Person> list = lines.skip(1).map(line -> parseCsv(line)).collect(Collectors.toList());
}
```

Why it works / Notes: Use try-with-resources to close file stream and consider using CSV libraries (OpenCSV) for robust parsing.

77. Find top N frequent words from a paragraph

Approach 1:

```
Map<String, Long> freq = Arrays.stream(text.toLowerCase().split("\W+"))
   .filter(s -> !s.isEmpty())
   .collect(Collectors.groupingBy(Function.identity(), Collectors.counting()));

List<String> topN = freq.entrySet().stream()
   .sorted(Map.Entry.<String,Long>comparingByValue().reversed())
   .limit(N)
   .map(Map.Entry::getKey)
   .collect(Collectors.toList());
```

Why it works / Notes: Compute frequency map then sort entries by count. For large texts, consider using a min-heap to keep top N in O(m log N) where m = unique words.

Approach 2 (Alternative):

```
List<String> topN2 = Arrays.stream(text.toLowerCase().split("\W+"))
   .filter(s -> !s.isEmpty())
   .collect(Collectors.groupingBy(Function.identity(), Collectors.counting()))
   .entrySet().stream()
   .sorted((e1,e2)->Long.compare(e2.getValue(), e1.getValue()))
   .limit(N).map(Map.Entry::getKey).toList();
```

Why it works / Notes: Same as above; chaining streams keeps pipeline concise.

78. Create a stream from a file and filter lines

Approach 1:

```
try (Stream<String> lines = Files.lines(Path.of("log.txt"))) {
   List<String> errors = lines.filter(l -> l.contains("ERROR")).collect(Collectors.toList());
}
```

Why it works / Notes: Files.lines gives lazy stream of lines. Be careful to close stream (try-with-resources).

Approach 2 (Alternative):

```
List<String> nonEmpty = Files.lines(path).filter(l -> !l.isBlank()).collect(Collectors.toList());
```

Why it works / Notes: Don't forget to handle IO exceptions; for very large files process line-by-line and avoid collecting all lines.

79. Count lines in a file containing a specific word

Approach 1:

```
long count = Files.lines(Path.of("file.txt"))
    .filter(l -> l.contains("TODO"))
    .count();
```

Why it works / Notes: Similar to above but uses terminal count(). For case-insensitive search convert to lower-case.

Approach 2 (Alternative):

```
long count2 = Files.lines(path).parallel().filter(l -> l.toLowerCase().contains("todo")).count();
```

Why it works / Notes: Parallel file streams may or may not help depending on I/O; benchmark and be careful with underlying file system.

80. Read CSV file and map rows to objects using streams

Approach 1:

```
List<Record> records = Files.lines(path)
    .skip(1)
    .map(line -> line.split(","))
    .map(cols -> new Record(cols[0], cols[1], Integer.parseInt(cols[2])))
    .collect(Collectors.toList());
```

Why it works / Notes: Same as Q76; consider trimming, quoting, escaped commas. For production prefer CSV parsers.

Approach 2 (Alternative):

```
CSVReader reader = new CSVReader(new FileReader("data.csv"));
List<String[]> all = reader.readAll();
List<Record> recs = all.stream().skip(1).map(cols -> new Record(cols[0], cols[1],
Integer.parseInt(cols[2]))).collect(Collectors.toList());
```

Why it works / Notes: OpenCSV or Commons CSV handle quoting and edge cases.

81. Find duplicate words in a text file

Approach 1:

```
Set<String> dups = Files.lines(path)
    .flatMap(l -> Arrays.stream(l.toLowerCase().split("\W+")))
    .filter(s -> !s.isEmpty())
    .collect(Collectors.groupingBy(Function.identity(), Collectors.counting()))
    .entrySet().stream()
    .filter(e -> e.getValue() > 1)
```

```
    .map(Map.Entry::getKey)
    .collect(Collectors.toSet());
```

Why it works / Notes: Group and count then filter counts>1. For streaming large files prefer incremental counting with a mutable map.

Approach 2 (Alternative):

```
Map<String, Long> counts = new HashMap<>();
Files.lines(path).forEach(line -> {
    Arrays.stream(line.toLowerCase().split("\W+")).filter(s->!s.isEmpty()).forEach(w ->
counts.merge(w,1L,Long::sum));
});
Set<String> dups2 = counts.entrySet().stream().filter(e-
>e.getValue()>1).map(Map.Entry::getKey).collect(Collectors.toSet());
```

Why it works / Notes: Mutable accumulation is more memory efficient if you need final counts anyway.

82. Use peek() for debugging in stream pipelines

Approach 1:

```
List<Integer> result = Stream.of(1,2,3,4,5)
    .peek(n -> System.out.println("before filter: " + n))
    .filter(n -> n % 2 == 0)
    .peek(n -> System.out.println("after filter: " + n))
    .map(n -> n * n)
    .collect(Collectors.toList());
```

Why it works / Notes: peek() is an intermediate operation useful for debugging; avoid relying on it for side-effects in production pipelines.

Approach 2 (Alternative):

```
Stream.of(1,2,3).map(n -> n*2).peek(System.out::println).count();
```

Why it works / Notes: Remember streams are lazy — peek will only run if a terminal operation exists.

83. Measure execution time of stream operations

Approach 1:

```
long start = System.nanoTime();
long sum = IntStream.rangeClosed(1, 1_000_000).parallel().filter(n->n%2==0).sum();
```

```
long end = System.nanoTime();
System.out.println("Elapsed ms: " + (end-start)/1_000_000);
```

Why it works / Notes: Use System.nanoTime() for fine-grained measurement; run multiple iterations and warm up JVM before measuring.

Approach 2 (Alternative):

```
Instant t0 = Instant.now();
IntStream.rangeClosed(1,1_000_000).filter(n->n%2==0).sum();
Duration d = Duration.between(t0, Instant.now());
System.out.println(d.toMillis());
```

Why it works / Notes: Use JMH for rigorous benchmarking; microbenchmarks are easy to get wrong without warmup.

84. Use parallel streams for CPU-intensive tasks

Approach 1:

```
long result = IntStream.range(0, 10_000_000).parallel()
   .map(n -> expensiveComputation(n))
   .sum();
```

Why it works / Notes: Parallel streams can utilize multiple cores; ensure expensiveComputation is CPU-bound and thread-safe, avoid shared mutable data.

Approach 2 (Alternative):

```
ForkJoinPool customPool = new ForkJoinPool(8);
int sum = customPool.submit(() -> IntStream.range(0, 1_000_000).parallel().sum()).join();
```

Why it works / Notes: Consider custom ForkJoinPool if you need explicit control of parallelism level.

85. Convert a list of enums to a map

Approach 1:

```
Map<MyEnum, String> map = Arrays.stream(MyEnum.values())
   .collect(Collectors.toMap(Function.identity(), e -> e.getDisplayName()));
```

Why it works / Notes: Use enum.values() stream and toMap to build lookups. If keys can collide use a merge function.

Approach 2 (Alternative):

```
Map<String, MyEnum> byCode = Arrays.stream(MyEnum.values())
   .collect(Collectors.toMap(MyEnum::getCode, Function.identity()));
```

Why it works / Notes: Useful for reverse lookups from property to enum.

86. Implement pagination logic for search results

Approach 1:

```
int page = 2, size = 20;
List<Result> pageData = results.stream()
   .skip((long)(page-1)*size)
   .limit(size)
   .collect(Collectors.toList());
```

Why it works / Notes: skip/limit works for in-memory collections. For large datasets or DB-backed searches prefer database pagination with OFFSET/LIMIT or cursor-based paging.

Approach 2 (Alternative):

```
// Cursor-style: assume results sorted by id
List<Result> page2 = results.stream().filter(r -> r.getId() >
lastSeenId).limit(size).collect(Collectors.toList());
```

Why it works / Notes: Cursor-based pagination avoids OFFSET performance penalties on large data.

87. Group transactions by currency and sum their values

Approach 1:

```
Map<String, Double> totals = transactions.stream()
   .collect(Collectors.groupingBy(Transaction::getCurrency,
Collectors.summingDouble(Transaction::getAmount)));
```

Why it works / Notes: Use groupingBy with summingDouble to aggregate numeric fields.

Approach 2 (Alternative):

```
Map<String, BigDecimal> totals2 = transactions.stream()
   .collect(Collectors.groupingBy(Transaction::getCurrency,
      Collectors.reducing(BigDecimal.ZERO, Transaction::getAmount, BigDecimal::add)));
```

Why it works / Notes: Use BigDecimal for money to avoid floating point errors.

88. Partition transactions into high and low value

Approach 1:

```
Map<Boolean, List<Transaction>> parts = transactions.stream()
   .collect(Collectors.partitioningBy(t -> t.getAmount().compareTo(threshold) >= 0));
```

Why it works / Notes: partitioningBy is optimized for boolean classification and returns exactly two buckets.

Approach 2 (Alternative):

```
Map<Boolean, Long> counts = transactions.stream()
   .collect(Collectors.partitioningBy(t -> t.getAmount() > threshold, Collectors.counting()));
```

Why it works / Notes: Downstream collectors let you compute summaries per partition.

89. Process nested JSON data using streams

Approach 1:

```
// Using Jackson to parse and stream nested arrays
ObjectMapper mapper = new ObjectMapper();
JsonNode root = mapper.readTree(jsonString);
List<String> ids = StreamSupport.stream(root.get("records").spliterator(), false)
   .flatMap(r -> StreamSupport.stream(r.get("items").spliterator(), false))
   .map(item -> item.get("id").asText())
   .collect(Collectors.toList());
```

Why it works / Notes: Use Jackson's tree model and StreamSupport to convert Iterable/Iterator into streams. For huge JSON use streaming parser (JsonParser) to avoid loading all into memory.

Approach 2 (Alternative):

```
JsonParser parser = mapper.getFactory().createParser(new File("data.json"));
// iterate tokens and extract required fields incrementally
```

Why it works / Notes: Streaming parsers are memory-efficient for very large JSON files.

90. Convert a Set to a Map using streams

Approach 1:

```
Map<K,V> map = set.stream().collect(Collectors.toMap(k -> k.getKey(), k -> k.getValue()));
```

Why it works / Notes: Use toMap with appropriate key/value mapping functions. If duplicate keys possible provide merge function and a Map supplier (e.g., LinkedHashMap).

Approach 2 (Alternative):

```
LinkedHashMap<K,V> map2 = set.stream()
   .collect(Collectors.toMap(k->k.getKey(), k->k.getValue(), (a,b)->a, LinkedHashMap::new));
```

Why it works / Notes: Choose Map implementation if ordering matters.

91. Sort a list of employees by salary and then name

Approach 1:

```
List<Employee> sorted = employees.stream()
   .sorted(Comparator.comparing(Employee::getSalary).reversed()
      .thenComparing(Employee::getName))
   .collect(Collectors.toList());
```

Why it works / Notes: Chain comparators; reverse salary order then tie-break by name.

Approach 2 (Alternative):

```
List<Employee> sorted2 = employees.stream()
   .sorted(Comparator.comparing(Employee::getSalary, Comparator.reverseOrder())
      .thenComparing(Employee::getName, Comparator.nullsLast(Comparator.naturalOrder())))
   .toList();
```

Why it works / Notes: Use null-safe comparators when fields may be null.

92. Find the employee with nth highest salary

Approach 1:

```
int n = 3;
Optional<Employee> nth = employees.stream()
   .map(Employee::getSalary)
   .distinct()
   .sorted(Comparator.reverseOrder())
   .skip(n-1)
   .findFirst()
   .flatMap(sal -> employees.stream().filter(e -> e.getSalary().equals(sal)).findFirst());
```

Why it works / Notes: Map salaries, deduplicate, sort descending, skip n-1 then find employees with that salary. Adjust tie handling as needed.

Approach 2 (Alternative):

```
Employee nth2 = employees.stream()
   .sorted(Comparator.comparing(Employee::getSalary).reversed())
   .skip(n-1).findFirst().orElse(null);
```

Why it works / Notes: Second approach returns nth by ordering including duplicates (i.e., stable by employee ordering).

93. Create a map of department to highest-paid employee

Approach 1:

```
Map<String, Optional<Employee>> best = employees.stream()
   .collect(Collectors.groupingBy(Employee::getDepartment,
      Collectors.maxBy(Comparator.comparing(Employee::getSalary))));
```

Why it works / Notes: groupingBy + maxBy returns Optional<Employee> per department. Use collectingAndThen to unwrap Optional.

Approach 2 (Alternative):

```
Map<String, Employee> best2 = employees.stream()
   .collect(Collectors.groupingBy(Employee::getDepartment,

Collectors.collectingAndThen(Collectors.maxBy(Comparator.comparing(Employee::getSalary)),
Optional::get)));
```

Why it works / Notes: collectingAndThen(Optional::get) will throw if a group is empty; ensure non-empty groups or handle safely.

94. Split a string into words and sort alphabetically

Approach 1:

```
List<String> words = Arrays.stream(text.split("\W+"))
   .filter(s -> !s.isBlank())
   .map(String::toLowerCase)
   .sorted()
   .collect(Collectors.toList());
```

Why it works / Notes: Split on non-word chars, normalize case, sort. For Unicode-aware tokenization use regex \p{L}+.

Approach 2 (Alternative):

```
List<String> words2 = Pattern.compile("\\p{L}+").matcher(text).results().map(m-
>m.group().toLowerCase()).sorted().toList();
```

Why it works / Notes: Pattern.results() (Java 9+) returns a stream of matches; good for robust tokenization.

95. Convert a range of numbers into a comma-separated string

Approach 1:

```
String s = IntStream.rangeClosed(1, 10).mapToObj(String::valueOf).collect(Collectors.joining(",
"));
```

Why it works / Notes: mapToObj + joining creates a single string efficiently. For very large ranges consider streaming directly to Writer.

Approach 2 (Alternative):

```
String s2 =
IntStream.rangeClosed(1,10).boxed().collect(Collectors.toCollection(ArrayList::new)).stream().m
ap(String::valueOf).collect(Collectors.joining(","));
```

Why it works / Notes: Direct joining avoids intermediate list; prefer mapToObj + joining.

96. Find employees who joined in the current year

Approach 1:

```
int year = Year.now().getValue();
List<Employee> joined = employees.stream()
    .filter(e -> e.getJoinDate().getYear() == year)
    .collect(Collectors.toList());
```

Why it works / Notes: Use java.time.LocalDate and getYear(). For different timezones consider ZonedDateTime.

Approach 2 (Alternative):

```
List<Employee> joined2 = employees.stream().filter(e ->
e.getJoinDate().isAfter(LocalDate.now().withDayOfYear(1).minusDays(1))).toList();
```

Why it works / Notes: Comparing with first day of year handles edge cases; ensure joinDate not null.

97. Implement custom predicate filters with streams

Approach 1:

```
Predicate<Employee> highSalary = e -> e.getSalary() > 100_000;
Predicate<Employee> inDept = e -> "IT".equals(e.getDepartment());
List<Employee> filtered = employees.stream().filter(highSalary.and(inDept)).toList();
```

Why it works / Notes: Compose predicates for reusable, readable filters. Use Predicate.and/or/negate.

Approach 2 (Alternative):

```
List<Employee> filtered2 = employees.stream().filter(Predicate.not(e -> e.isContract())).toList();
```

Why it works / Notes: Predicate.not is available Java 11+. Avoid heavy logic in predicates to keep pipelines readable.

98. Find the longest repeating sequence in a list

Approach 1:

```
// Example: find longest run length and the element
List<Integer> list = Arrays.asList(1,1,1,2,2,3,3,3,3,2);
int maxLen = 0; int curLen = 0; Integer curVal = null; Integer bestVal = null;
for (Integer v : list) {
   if (!v.equals(curVal)) { curVal = v; curLen = 1; }
   else curLen++;
   if (curLen > maxLen) { maxLen = curLen; bestVal = curVal; }
}
```

Why it works / Notes: Classic one-pass algorithm; streams are not ideal for stateful sequential run-length detection — use a loop or custom collector if you need stream style.

Approach 2 (Alternative):

```
// Custom collector outline (advanced) - maintain current and best state across elements (left as exercise)
```

Why it works / Notes: Stateful operations are tricky in streams — prefer imperative loops for clarity and performance.

99. Convert a list of dates to sorted LocalDate objects

Approach 1:

```
List<LocalDate> sortedDates = dates.stream()
   .map(LocalDate::parse) // if strings
   .sorted()
   .collect(Collectors.toList());
```

Why it works / Notes: LocalDate implements Comparable so sorting works directly. For custom formats use DateTimeFormatter in map step.

Approach 2 (Alternative):

```
DateTimeFormatter fmt = DateTimeFormatter.ofPattern("dd-MM-yyyy");
List<LocalDate> s2 = dateStrings.stream().map(d -> LocalDate.parse(d, fmt)).sorted().toList();
```

Why it works / Notes: Be defensive about parsing exceptions; consider filter/flatMap for malformed rows.

100. Implement batch processing of records using streams

Approach 1:

```
int batchSize = 1000;
List<Record> all = getAllRecords();
AtomicInteger counter = new AtomicInteger();
```

```
all.stream()
  .collect(Collectors.groupingBy(r -> counter.getAndIncrement() / batchSize))
  .values().forEach(batch -> processBatch(batch));
```

Why it works / Notes: Grouping by index using counter partitions list into batches. For very large datasets avoid collecting all records; process via iterator and flush per batch.

Approach 2 (Alternative):

```
Iterator<Record> it = recordIterator();
List<Record> batch = new ArrayList<>(batchSize);
while (it.hasNext()) {
   batch.add(it.next());
   if (batch.size() == batchSize) { processBatch(batch); batch.clear(); }
}
if (!batch.isEmpty()) processBatch(batch);
```

Why it works / Notes: Iterator-based batching is memory efficient; the stream grouping approach needs entire collection loaded and uses additional memory.

Final Notes

• Always be careful with parallel streams: thread-safety, shared mutable state, ordering guarantees, and the cost of splitting/merging.

• Prefer primitive streams for numeric heavy work to avoid boxing: IntStream, LongStream, DoubleStream.

• For IO-heavy tasks (file reading, DB calls) parallel streams may not help; prefer asynchronous IO or batching.

• Use established libraries (Jackson, OpenCSV) for parsing JSON/CSV reliably in production.