Java Stream API Coding Interview Questions - Intermediate Level (31–70)

31) Group a list of strings by their length

Approach 1 — `groupingBy` (common):

```
List<String> words = List.of("apple","banana","kiwi","pear");
Map<Integer, List<String>> byLen = words.stream()
    .collect(Collectors.groupingBy(String::length));
```

Approach 2 — with downstream `toSet` (unique values):

```
Map<Integer, Set<String>> byLenSet = words.stream()
    .collect(Collectors.groupingBy(String::length,
Collectors.toSet()));
```

Explanation: `Collectors.groupingBy(classifier)` buckets elements by key; downstream collectors let you control value type.

---

32) Count the frequency of each element in a list

Approach 1 — `groupingBy` + `counting` (idiomatic):

```
List<String> items = List.of("a","b","a","c","b","a");
Map<String, Long> freq = items.stream()
    .collect(Collectors.groupingBy(Function.identity(),
Collectors.counting()));
```

Approach 2 — `toMap` with merge function:

```
Map<String, Integer> freq2 = items.stream()
    .collect(Collectors.toMap(Function.identity(), v -> 1,
Integer::sum));
```

Explanation: Both build frequency maps; `groupingBy` easily returns Long counts.

---

33) Find the longest string in a list

Approach 1 — `max` with comparator:

```
Optional<String> longest = words.stream()
    .max(Comparator.comparingInt(String::length));
```

Approach 2 — `reduce` pairwise:

```
Optional<String> longest2 = words.stream()
    .reduce((a, b) -> a.length() >= b.length() ? a : b);
```

Explanation: `max` is clearer; `reduce` demonstrates functional reasoning.

---

34) Find the shortest string in a list

Approach 1 — `min` with comparator:

```
Optional<String> shortest = words.stream()
    .min(Comparator.comparingInt(String::length));
```

Approach 2 — sort and take first (less efficient):

```
Optional<String> shortest2 = words.stream()
    .sorted(Comparator.comparingInt(String::length))
    .findFirst();
```

Explanation: `min` is O(n); sorting is O(n log n) but sometimes simpler.

---

35) Implement pagination using `skip()` and `limit()`

Approach 1 — slice a list:

```
int page = 3, size = 10;
List<T> pageData = list.stream()
    .skip((long)(page - 1) * size)
    .limit(size)
    .collect(Collectors.toList());
```

Approach 2 — using `IntStream` index slicing:

```
List<T> pageData2 = IntStream.range(0, list.size())
```

```
    .filter(i -> i >= (page-1)*size && i < page*size)
    .mapToObj(list::get)
    .collect(Collectors.toList());
```

Explanation: `skip`/`limit` is straightforward; index-based approach helps when computing offsets with boundaries.

---

36) Sort a list of objects by a field

Approach 1 — `Comparator.comparing`:

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

Approach 2 — lambda comparator:

```
List<Employee> sorted2 = employees.stream()
    .sorted((a,b) -> a.getName().compareTo(b.getName()))
    .toList();
```

Explanation: `Comparator.comparing` is more readable and null-safe variants are available.

---

37) Sort a list of objects by multiple fields

Approach 1 — `thenComparing`:

```
employees.stream()
    .sorted(Comparator.comparing(Employee::getDept)
        .thenComparing(Employee::getName))
    .toList();
```

Approach 2 — combined lambda comparator:

```
employees.stream()
    .sorted((a,b) -> {
       int r = a.getDept().compareTo(b.getDept());
       return r != 0 ? r : a.getName().compareTo(b.getName());
    })
    .toList();
```

Explanation: Chain comparators with `thenComparing` for clarity; handle nulls with `Comparator.nullsFirst/Last`.

---

38) Find the second-highest salary from a list of employees

Approach 1 — map, distinct, sort desc, skip:

```
Optional<Integer> second = employees.stream()
    .map(Employee::getSalary)
    .distinct()
    .sorted(Comparator.reverseOrder())
    .skip(1)
    .findFirst();
```

Approach 2 — use `TreeSet` (unique sorted set):

```
TreeSet<Integer> set = employees.stream()
    .map(Employee::getSalary)
    .collect(Collectors.toCollection(() -> new
TreeSet<>(Comparator.reverseOrder())));
Integer second2 = set.stream().skip(1).findFirst().orElse(null);
```

Explanation: Distinct + reverse sort is concise; TreeSet avoids additional sorting on repeated queries.

---

39) Find the top N elements from a list

Approach 1 — sort and `limit`:

```
int N = 5;
List<Integer> topN = nums.stream()
    .sorted(Comparator.reverseOrder())
    .limit(N)
    .toList();
```

Approach 2 — maintain a min-heap (efficient for large streams):

```
PriorityQueue<Integer> pq = new PriorityQueue<>();
nums.forEach(x -> { pq.offer(x); if (pq.size() > N) pq.poll(); });
List<Integer> top =
pq.stream().sorted(Comparator.reverseOrder()).toList();
```

Explanation: Sorting is easy; heap approach is O(n log N) and better for streaming large inputs.

---

40) Merge two lists using streams

Approach 1 — `Stream.concat`:

```
List<T> merged = Stream.concat(a.stream(),
b.stream()).collect(Collectors.toList());
```

Approach 2 — `Stream.of` + `flatMap` (extendable to many lists):

```
List<T> merged2 = Stream.of(a, b)
    .flatMap(Collection::stream)
    .toList();
```

Explanation: `concat` merges two streams; `flatMap` generalizes to combine many collections.

---

41) Flatten a list of arrays into a single list

Approach 1 — `flatMap` with `Arrays::stream`:

```
List<String> flat = listOfArrays.stream()
    .flatMap(Arrays::stream)
    .collect(Collectors.toList());
```

Approach 2 — primitive arrays with `flatMapToInt`:

```
int[] flat = listOfIntArrays.stream()
    .flatMapToInt(Arrays::stream)
    .toArray();
```

Explanation: Use `flatMap` for object streams; primitive `flatMapToX` for performance.

---

42) Remove duplicates based on a specific field in objects

Approach 1 — `toMap` keyed by field (keeps first):

```
Collection<Employee> unique = employees.stream()
    .collect(Collectors.toMap(Employee::getId, e -> e, (e1,e2) -> e1,
LinkedHashMap::new))
    .values();
```

Approach 2 — `TreeSet` with comparator on field (keeps unique per comparator):

```
List<Employee> unique2 = employees.stream()
    .collect(Collectors.collectingAndThen(
        Collectors.toCollection(() -> new
TreeSet<>(Comparator.comparing(Employee::getId))),
        ArrayList::new));
```

Explanation: `toMap` dedupes by key; `TreeSet` enforces ordering and uniqueness via comparator.

---

43) Count words in a string using streams

Approach 1 — split + `groupingBy`:

```
Map<String, Long> counts = Arrays.stream(text.split("\\s+"))
    .map(String::toLowerCase)
    .collect(Collectors.groupingBy(Function.identity(),
Collectors.counting()));
```

Approach 2 — regex `Matcher.results()` (Unicode-aware, Java 9+):

```
Pattern p = Pattern.compile("\\p{L}+");
Map<String, Long> counts2 = p.matcher(text).results()
    .map(m -> m.group().toLowerCase())
    .collect(Collectors.groupingBy(Function.identity(),
Collectors.counting()));
```

Explanation: Splitting is simple; regex captures word tokens more robustly (punctuation etc).

---

44) Reverse a list using streams

Approach 1 — index-based mapping:

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

Approach 2 — non-stream `Collections.reverse` (recommended for mutability):

```
List<T> copy = new ArrayList<>(list);
Collections.reverse(copy);
```

Explanation: Streams can reverse by index; for in-place reversal `Collections.reverse` is simpler and faster.

---

45) Create an infinite stream of numbers and limit the output

Approach 1 — `Stream.iterate`:

```
List<Integer> first10 = Stream.iterate(0, n -> n +
1).limit(10).collect(Collectors.toList());
```

Approach 2 — primitive `IntStream.iterate`:

```
int[] first10 = IntStream.iterate(0, n -> n + 1).limit(10).toArray();
```

Explanation: `iterate` generates an infinite sequence; always use `limit` to bound it.

---

46) Map a list of numbers to their cubes

Approach 1 — object stream, compute cube:

```
List<Integer> cubes = nums.stream().map(n -> n * n * n).toList();
```

Approach 2 — primitive stream for performance:

```
int[] cubes = nums.stream().mapToInt(n -> n * n * n).toArray();
```

Explanation: Primitive streams avoid boxing and are faster for numeric transformations.

---

47) Get the frequency of each word from a paragraph

Approach 1 — split + grouping:

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

Approach 2 — regex tokenization with `Pattern`:

```
Pattern p = Pattern.compile("\\p{L}+");
Map<String, Long> freq2 = p.matcher(para).results()
    .map(m -> m.group().toLowerCase())
    .collect(Collectors.groupingBy(Function.identity(),
Collectors.counting()));
```

Explanation: Similar to Q43; prefer regex for robust token extraction.

---

48) Filter employees earning more than a certain salary

Approach 1 — simple `filter`:

```
List<Employee> rich = employees.stream()
    .filter(e -> e.getSalary() > threshold)
    .toList();
```

Approach 2 — `Collectors.filtering` downstream (Java 9+):

```
List<Employee> rich2 = employees.stream()
    .collect(Collectors.filtering(e -> e.getSalary() > threshold,
Collectors.toList()));
```

Explanation: Both yield same result; `filtering` can be handy inside larger collectors (e.g., grouping).

---

49) Find the oldest person in a list of people

Approach 1 — `max` by age:

```
Optional<Person> oldest = people.stream()
    .max(Comparator.comparingInt(Person::getAge));
```

Approach 2 — `reduce` to keep older:

```
Optional<Person> oldest2 = people.stream()
    .reduce((p1, p2) -> p1.getAge() >= p2.getAge() ? p1 : p2);
```

Explanation: `max` is idiomatic; reduce shows manual fold logic.

---

50) Find the youngest person in a list of people

Approach 1 — `min` by age:

```
Optional<Person> youngest = people.stream()
    .min(Comparator.comparingInt(Person::getAge));
```

Approach 2 — sorted then first (less efficient):

```
Optional<Person> youngest2 = people.stream()
    .sorted(Comparator.comparingInt(Person::getAge))
    .findFirst();
```

Explanation: `min` is O(n), sorting is O(n log n).

---

51) Group employees by department

Approach 1 — `groupingBy` to list:

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

Approach 2 — map dept -> set of names:

```
Map<String, Set<String>> namesByDept = employees.stream()
    .collect(Collectors.groupingBy(Employee::getDepartment,
        Collectors.mapping(Employee::getName, Collectors.toSet())));
```

Explanation: Downstream mapping extracts and collects only the desired property.

---

52) Group employees by department and count them

Approach 1 — `groupingBy` + `counting`:

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

Approach 2 — `toMap` accumulation:

```
Map<String, Integer> counts2 = employees.stream()
    .collect(Collectors.toMap(Employee::getDepartment, e -> 1,
Integer::sum));
```

Explanation: `groupingBy` is clear; `toMap` provides integer counts when preferred.

---

53) Find duplicate elements in a list

Approach 1 — build frequency map then filter:

```
Set<T> duplicates = list.stream()
    .collect(Collectors.groupingBy(Function.identity(),
Collectors.counting()))
    .entrySet().stream()
    .filter(e -> e.getValue() > 1)
    .map(Map.Entry::getKey)
    .collect(Collectors.toSet());
```

Approach 2 — single-pass `Set` tracking (non-stream but O(n)):

```
Set<T> seen = new HashSet<>(), dups = new HashSet<>();
for (T x : list) if (!seen.add(x)) dups.add(x);
```

Explanation: Frequency map is functional; set tracking is efficient and simple.

---

54) Remove duplicates while maintaining order

Approach 1 — `distinct()` (preserves encounter order for ordered streams):

```
List<T> unique = list.stream().distinct().toList();
```

Approach 2 — `LinkedHashSet` roundtrip (non-stream):

```
List<T> unique2 = new ArrayList<>(new LinkedHashSet<>(list));
```

Explanation: `distinct()` uses encounter order of source; LinkedHashSet preserves insertion order.

---

55) Find the element with the maximum frequency in a list

Approach 1 — frequency map then max entry:

```
String mode = items.stream()
```

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

Approach 2 — manual `merge` then max:

```
Map<String, Long> m = new HashMap<>();
items.forEach(s -> m.merge(s, 1L, Long::sum));
String mode2 =
m.entrySet().stream().max(Map.Entry.comparingByValue()).map(Map.Entry::
getKey).orElse(null);
```

Explanation: Build counts then pick max; ties resolved by map ordering unless otherwise specified.

---

56) Get the last element of a list using streams

Approach 1 — `reduce` to last:

```
T last = list.stream().reduce((a, b) -> b).orElse(null);
```

Approach 2 — index-based stream:

```
T last2 = IntStream.range(0, list.size())
    .mapToObj(list::get)
    .skip(list.size() - 1)
    .findFirst()
    .orElse(null);
```

Explanation: `reduce((a,b)->b)` returns the last observed element; index approach is explicit.

---

57) Check if a list is sorted

Approach 1 — pairwise index check:

```
boolean sorted = IntStream.range(1, list.size())
    .allMatch(i -> list.get(i-1).compareTo(list.get(i)) <= 0);
```

Approach 2 — compare with sorted copy:

```
boolean sorted2 = list.stream().sorted().toList().equals(list);
```

Explanation: Pairwise check is O(n); comparing with sorted copy is simpler but O(n log n).

---

58) Convert a list of integers to a formatted string

Approach 1 — `map` + `joining`:

```
String s =
nums.stream().map(String::valueOf).collect(Collectors.joining(", "));
```

Approach 2 — template formatting per element:

```
String s2 = nums.stream().map(n -> String.format("[%d]",
n)).collect(Collectors.joining(" "));
```

Explanation: `joining` easily formats sequences; `String.format` for custom templates.

---

59) Get a list of prime numbers from a range

Approach 1 — trial division up to sqrt(n):

```
List<Integer> primes = IntStream.rangeClosed(2, end)
    .filter(n -> IntStream.rangeClosed(2,
(int)Math.sqrt(n)).noneMatch(d -> n % d == 0))
    .boxed().toList();
```

Approach 2 — simpler (but slower) check up to n-1:

```
List<Integer> primes2 = IntStream.rangeClosed(2, end)
    .filter(n -> IntStream.rangeClosed(2, n-1).allMatch(d -> n % d !=
0))
    .boxed().toList();
```

Explanation: Use sqrt(n) bound for efficiency.

---

60) Generate Fibonacci numbers using streams

Approach 1 — pair tuple with `iterate`:

```
List<Integer> fib = Stream.iterate(new int[]{0,1}, a -> new int[]{a[1],
a[0]+a[1]})
    .limit(n)
    .map(a -> a[0])
    .toList();
```

Approach 2 — using `Long` tuple for larger sequences:

```
List<Long> fib2 = Stream.iterate(new long[]{0,1}, a -> new long[]{a[1],
a[0]+a[1]})
    .limit(n)
    .map(a -> a[0])
    .toList();
```

Explanation: Use two-value state in `iterate` to carry previous and current.

---

61) Implement custom collectors (example: sum of squares)

Approach 1 — `Collector.of`:

```
Collector<Integer, int[], Integer> sumSquares = Collector.of(
    () -> new int[1],
    (a, t) -> a[0] += t * t,
    (a, b) -> { a[0] += b[0]; return a; },
    a -> a[0]
);
int res = Stream.of(1,2,3).collect(sumSquares);
```

Approach 2 — use `mapToInt` + `sum` (simpler):

```
int res2 = Stream.of(1,2,3).mapToInt(x -> x * x).sum();
```

Explanation: Custom collectors give control; often primitive streams give simpler/faster solutions.

---

62) Use `reduce()` to concatenate strings

Approach 1 — `reduce` with identity (less efficient due to repeated concatenation):

```
String s = parts.stream().reduce("", (a,b)-> a.isEmpty() ? b : a + " "
+ b);
```

Approach 2 — `Collectors.joining` (preferred):

```
String s2 = parts.stream().collect(Collectors.joining(" "));
```

Explanation: `joining` uses efficient `StringBuilder` internally; avoid naive `reduce` for many strings.

---

63) Use `reduce()` to multiply all numbers in a list

Approach 1 — `reduce` with identity `1`:

```
int product = nums.stream().reduce(1, (a,b) -> a * b);
```

Approach 2 — `mapToLong` and reduce to long to reduce overflow risk:

```
long product2 = nums.stream().mapToLong(Integer::longValue).reduce(1L,
(a,b) -> a * b);
```

Explanation: Choose numeric type carefully to avoid overflow.

---

64) Use `reduce()` to find the max number without `max()`

Approach 1 — `reduce(Integer::max)`:

```
Optional<Integer> max = nums.stream().reduce(Integer::max);
```

Approach 2 — `reduce` with identity and ternary comparator:

```
int max2 = nums.stream().reduce(Integer.MIN_VALUE, (a,b) -> a > b ? a :
b);
```

Explanation: Both implement fold to largest element; ensure correct identity for empty lists.

---

65) Convert a map's values to a list using streams

Approach 1 — stream over `values()`:

```
List<V> vals = map.values().stream().collect(Collectors.toList());
```

Approach 2 — stream entries then map to value:

```
List<V> vals2 =
map.entrySet().stream().map(Map.Entry::getValue).toList();
```

Explanation: Both obtain values; `entrySet` useful if you also need keys.

---

66) Convert a map's keys to a list using streams

Approach 1 — stream over `keySet()`:

```
List<K> keys = map.keySet().stream().toList();
```

Approach 2 — stream entries then map to key:

```
List<K> keys2 =
map.entrySet().stream().map(Map.Entry::getKey).toList();
```

Explanation: Symmetric to values conversion.

---

67) Sort a map by its values using streams

Approach 1 — collect to `LinkedHashMap` to preserve ordering:

```
LinkedHashMap<K,V> sorted = map.entrySet().stream()
    .sorted(Map.Entry.comparingByValue())
    .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue,
        (a,b)->a, LinkedHashMap::new));
```

Approach 2 — collect to sorted `List` of entries (if map not required):

```
List<Map.Entry<K,V>> list = map.entrySet().stream()
    .sorted(Map.Entry.comparingByValue(Comparator.reverseOrder()))
    .toList();
```

Explanation: Use `LinkedHashMap` to retain sorted iteration order when a map is needed.

---

68) Find intersection of two lists using streams

Approach 1 — hash lookup (fast):

```
Set<T> setB = new HashSet<>(b);
List<T> inter = a.stream().filter(setB::contains).distinct().toList();
```

Approach 2 — `filter(b::contains)` (simpler but O(n*m)):

```
List<T> inter2 = a.stream().filter(b::contains).distinct().toList();
```

Explanation: Use a `HashSet` of the smaller collection for O(n) behavior.

---

69) Find union of two lists using streams

Approach 1 — concat + distinct:

```
List<T> union = Stream.concat(a.stream(),
b.stream()).distinct().toList();
```

Approach 2 — set union preserving order:

```
Set<T> unionSet = new LinkedHashSet<>(a);
unionSet.addAll(b);
List<T> union2 = new ArrayList<>(unionSet);
```

Explanation: `distinct` dedupes combined stream; LinkedHashSet retains insertion order.

---

70) Find difference between two lists using streams

Approach 1 — A \ B with hash lookup:

```
Set<T> setB = new HashSet<>(b);
List<T> diff = a.stream().filter(x -> !setB.contains(x)).toList();
```

Approach 2 — `removeAll` on a copy (non-stream):

```
List<T> copy = new ArrayList<>(a);
copy.removeAll(b); // copy now contains a minus b
```

Explanation: Hash-based filter is efficient; `removeAll` is simple and mutates copy.