Java Stream API — Basic Level (1–30) — Coding Questions with Multiple Approaches

Each problem includes at least two Stream-centric approaches, plus brief explanations and notes.

1) Find all even numbers from a list

Approach 1:

```
import java.util.*;
import java.util.stream.*;

List<Integer> list = Arrays.asList(1,2,3,4,5,6);
List<Integer> evens = list.stream()
                .filter(n -> n % 2 == 0)
                .collect(Collectors.toList());
System.out.println(evens);
```

Why it works: Use filter with a modulo predicate and collect to List.

Approach 2 (Alternative):

```
import java.util.*;
import java.util.stream.*;

List<Integer> list = Arrays.asList(1,2,3,4,5,6);
List<Integer> evens = list.stream()
                .collect(Collectors.filtering(n -> n % 2 == 0,
Collectors.toList()));
System.out.println(evens);
```

Why it works: Using Collectors.filtering (Java 9+) pushes the predicate into the downstream collector.

Notes: Performance tip: prefer primitive streams when doing heavy numeric work.

2) Convert a list of strings to uppercase

Approach 1:

```
import java.util.*;
import java.util.stream.*;

List<String> names = Arrays.asList("java","stream","api");
List<String> upper = names.stream()
                 .map(String::toUpperCase)
                 .collect(Collectors.toList());
System.out.println(upper);
```

Why it works: map with method reference for clarity.

Approach 2 (Alternative):

```
import java.util.*;
import java.util.stream.*;

List<String> names = Arrays.asList("java","stream","api");
List<String> upper = names.stream()
                 .collect(Collectors.mapping(String::toUpperCase,
Collectors.toList()));
System.out.println(upper);
```

Why it works: Collectors.mapping (Java 9+) performs the transform in the collector.

3) Find the sum of all numbers in a list

Approach 1:

```
import java.util.*;
import java.util.stream.*;

List<Integer> nums = Arrays.asList(1,2,3,4);
int sum = nums.stream().mapToInt(Integer::intValue).sum();
System.out.println(sum);
```

Why it works: mapToInt creates IntStream enabling sum().

Approach 2 (Alternative):

```
import java.util.*;
import java.util.stream.*;

List<Integer> nums = Arrays.asList(1,2,3,4);
int sum = nums.stream().reduce(0, Integer::sum);
System.out.println(sum);
```

Why it works: reduce with identity & accumulator keeps boxing but is concise.

Notes: Prefer primitive streams (mapToInt) to avoid boxing.

4) Find the maximum number in a list

Approach 1:

```
import java.util.*;
import java.util.stream.*;

List<Integer> nums = Arrays.asList(3,7,2,9,5);
int max =
nums.stream().mapToInt(Integer::intValue).max().orElseThrow();
System.out.println(max);
```

Why it works: Use IntStream.max + orElseThrow for empty handling.

Approach 2 (Alternative):

```
import java.util.*;
import java.util.stream.*;

List<Integer> nums = Arrays.asList(3,7,2,9,5);
int max = nums.stream().reduce(Integer::max).orElseThrow();
System.out.println(max);
```

Why it works: reduce with Integer::max avoids converting to primitive stream.

5) Find the minimum number in a list

Approach 1:

```
import java.util.*;
import java.util.stream.*;

List<Integer> nums = Arrays.asList(3,7,2,9,5);
int min =
nums.stream().mapToInt(Integer::intValue).min().orElse(Integer.MI
N_VALUE);
System.out.println(min);
```

Why it works: IntStream.min with fallback.

Approach 2 (Alternative):

```
import java.util.*;
import java.util.stream.*;

List<Integer> nums = Arrays.asList(3,7,2,9,5);
int min =
nums.stream().min(Comparator.naturalOrder()).orElseThrow();
System.out.println(min);
```

Why it works: Stream.min with Comparator.

6) Count the number of strings starting with a specific letter

Approach 1:

```
import java.util.*;
import java.util.stream.*;

List<String> words =
Arrays.asList("apple","banana","apricot","cherry");
long count = words.stream()
        .filter(s -> s.startsWith("a"))
        .count();
System.out.println(count);
```

Why it works: Filter then count.

Approach 2 (Alternative):

```
import java.util.*;
import java.util.stream.*;

List<String> words =
Arrays.asList("apple","banana","apricot","cherry");
long count = words.stream()
        .collect(Collectors.filtering(s -> s.startsWith("a"),
Collectors.counting()));
System.out.println(count);
```

Why it works: Use Collectors.filtering + counting.

7) Remove duplicate elements from a list

Approach 1:

```
import java.util.*;
import java.util.stream.*;
```

```
List<Integer> nums = Arrays.asList(1,2,2,3,3,3,4);
List<Integer> distinct =
nums.stream().distinct().collect(Collectors.toList());
System.out.println(distinct);
```

Why it works: distinct uses equals/hashCode.

Approach 2 (Alternative):

```
import java.util.*;
import java.util.stream.*;

List<Integer> nums = Arrays.asList(1,2,2,3,3,3,4);
List<Integer> distinct = new ArrayList<>(new
LinkedHashSet<>(nums));
System.out.println(distinct);
```

Why it works: Alternative not strictly streams: LinkedHashSet preserves order then back to list.

8) Sort a list in ascending order

Approach 1:

```
import java.util.*;
import java.util.stream.*;

List<Integer> nums = Arrays.asList(4,1,3,2);
List<Integer> sorted =
nums.stream().sorted().collect(Collectors.toList());
System.out.println(sorted);
```

Why it works: sorted() natural order.

Approach 2 (Alternative):

```
import java.util.*;
import java.util.stream.*;

List<Integer> nums = Arrays.asList(4,1,3,2);
List<Integer> sorted =
nums.stream().sorted(Comparator.naturalOrder()).collect(Collector
s.toList());
System.out.println(sorted);
```

Why it works: Explicit comparator for readability or generic code.

9) Sort a list in descending order

Approach 1:

```
import java.util.*;
import java.util.stream.*;

List<Integer> nums = Arrays.asList(4,1,3,2);
List<Integer> desc =
nums.stream().sorted(Comparator.reverseOrder()).collect(Collectors.toList());
System.out.println(desc);
```

Why it works: Reverse natural order.

Approach 2 (Alternative):

```
import java.util.*;
import java.util.stream.*;

List<Integer> nums = Arrays.asList(4,1,3,2);
List<Integer> desc = nums.stream()
                .sorted((a,b) -> Integer.compare(b,a))
                .collect(Collectors.toList());
System.out.println(desc);
```

Why it works: Custom comparator, equivalent.

10) Find the first element of a list

Approach 1:

```
import java.util.*;
import java.util.stream.*;

List<String> list = Arrays.asList("a","b","c");
String first = list.stream().findFirst().orElse(null);
System.out.println(first);
```

Why it works: findFirst returns Optional; choose default.

Approach 2 (Alternative):

```
import java.util.*;
import java.util.stream.*;

List<String> list = Arrays.asList("a","b","c");
String first =
list.stream().limit(1).collect(Collectors.collectingAndThen(Collectors
.toList(), l -> l.isEmpty()? null : l.get(0)));
System.out.println(first);
```

Why it works: Limit + collectingAndThen shows collector post-processing.

11) Check if any element in the list matches a condition

Approach 1:

```
import java.util.*;
import java.util.stream.*;

List<Integer> nums = Arrays.asList(1,3,5,8);
boolean hasEven = nums.stream().anyMatch(n -> n % 2 == 0);
System.out.println(hasEven);
```

Why it works: anyMatch short-circuits on first match.

Approach 2 (Alternative):

```
import java.util.*;
import java.util.stream.*;

List<Integer> nums = Arrays.asList(1,3,5,8);
boolean hasEven = nums.stream().filter(n -> n % 2 ==
0).findAny().isPresent();
System.out.println(hasEven);
```

Why it works: Filter + findAny equivalent but less efficient.

12) Check if all elements match a condition

Approach 1:

```
import java.util.*;
import java.util.stream.*;

List<Integer> nums = Arrays.asList(2,4,6);
boolean allEven = nums.stream().allMatch(n -> n % 2 == 0);
System.out.println(allEven);
```

Why it works: allMatch checks all with short-circuiting.

Approach 2 (Alternative):

```
import java.util.*;
import java.util.stream.*;

List<Integer> nums = Arrays.asList(2,4,6);
boolean allEven = !nums.stream().anyMatch(n -> n % 2 != 0);
System.out.println(allEven);
```

Why it works: Logical negation of anyMatch.

13) Check if no elements match a condition

Approach 1:

```
import java.util.*;
import java.util.stream.*;

List<String> words = Arrays.asList("cat","dog");
boolean noneLongerThan5 = words.stream().noneMatch(s ->
s.length() > 5);
System.out.println(noneLongerThan5);
```

Why it works: noneMatch is the negation of anyMatch.

Approach 2 (Alternative):

```
import java.util.*;
import java.util.stream.*;

List<String> words = Arrays.asList("cat","dog");
boolean noneLongerThan5 = words.stream().allMatch(s ->
s.length() <= 5);
System.out.println(noneLongerThan5);
```

Why it works: Equivalent using allMatch.

14) Filter null values from a list

Approach 1:

```
import java.util.*;
import java.util.stream.*;
```

```
List<String> list = Arrays.asList("a", null, "b", null, "c");
List<String> nonNull =
list.stream().filter(Objects::nonNull).collect(Collectors.toList());
System.out.println(nonNull);
```

Why it works: Objects::nonNull is idiomatic.

Approach 2 (Alternative):

```
import java.util.*;
import java.util.stream.*;

List<String> list = Arrays.asList("a", null, "b", null, "c");
List<String> nonNull = list.stream().flatMap(s -> s == null ?
Stream.empty() : Stream.of(s)).collect(Collectors.toList());
System.out.println(nonNull);
```

Why it works: flatMap to drop nulls.

15) Convert a list of integers to their square values

Approach 1:

```
import java.util.*;
import java.util.stream.*;

List<Integer> nums = Arrays.asList(1,2,3,4);
List<Integer> squares = nums.stream().map(n ->
n*n).collect(Collectors.toList());
System.out.println(squares);
```

Why it works: map applies a pure function.

Approach 2 (Alternative):

```
import java.util.*;
import java.util.stream.*;

List<Integer> nums = Arrays.asList(1,2,3,4);
int[] squares = nums.stream().mapToInt(n -> n*n).toArray();
System.out.println(Arrays.toString(squares));
```

Why it works: Primitive stream to array.

16) Collect stream results into a Set instead of a List

Approach 1:

```
import java.util.*;
import java.util.stream.*;

List<Integer> nums = Arrays.asList(1,2,2,3);
Set<Integer> set = nums.stream().collect(Collectors.toSet());
System.out.println(set);
```

Why it works: Collectors.toSet uses HashSet by default (no order).

Approach 2 (Alternative):

```
import java.util.*;
import java.util.stream.*;

List<Integer> nums = Arrays.asList(1,2,2,3);
Set<Integer> set =
nums.stream().collect(Collectors.toCollection(LinkedHashSet::new));
System.out.println(set);
```

Why it works: Use toCollection to choose a Set implementation and preserve insertion order.

17) Join a list of strings into a single comma-separated string

Approach 1:

```
import java.util.*;
import java.util.stream.*;

List<String> parts = Arrays.asList("a","b","c");
String csv = parts.stream().collect(Collectors.joining(","));
System.out.println(csv);
```

Why it works: Collectors.joining with delimiter.

Approach 2 (Alternative):

```
import java.util.*;
import java.util.stream.*;

List<String> parts = Arrays.asList("a","b","c");
```

```
String csv = String.join(",", parts);
System.out.println(csv);
```

Why it works: Alternative using String.join (not a stream but idiomatic).

18) Find the average of a list of numbers

Approach 1:

```
import java.util.*;
import java.util.stream.*;

List<Integer> nums = Arrays.asList(1,2,3,4);
double avg =
nums.stream().mapToInt(Integer::intValue).average().orElse(0.0);
System.out.println(avg);
```

Why it works: IntStream.average returns OptionalDouble.

Approach 2 (Alternative):

```
import java.util.*;
import java.util.stream.*;

List<Integer> nums = Arrays.asList(1,2,3,4);
double avg =
nums.stream().collect(Collectors.averagingInt(Integer::intValue));
System.out.println(avg);
```

Why it works: Use averaging collector for readability.

19) Convert a list of objects to a list of one of their fields

Approach 1:

```
import java.util.*;
import java.util.stream.*;

record User(int id, String name) {}
List<User> users = Arrays.asList(new User(1,"A"), new User(2,"B"));
List<String> names =
users.stream().map(User::name).collect(Collectors.toList());
System.out.println(names);
```

Why it works: Method reference to accessor.

Approach 2 (Alternative):

```
import java.util.*;
import java.util.stream.*;

class User { int id; String name; User(int i,String n){id=i;name=n;}
String getName(){return name;} }
List<User> users = Arrays.asList(new User(1,"A"), new User(2,"B"));
Set<String> names =
users.stream().collect(Collectors.mapping(User::getName,
Collectors.toSet()));
System.out.println(names);
```

Why it works: Collectors.mapping to a Set.

20) Skip the first N elements in a stream

Approach 1:

```
import java.util.*;
import java.util.stream.*;

List<Integer> nums = Arrays.asList(10,20,30,40,50);
List<Integer> after2 =
nums.stream().skip(2).collect(Collectors.toList());
System.out.println(after2);
```

Why it works: skip(n) discards the first n elements.

Approach 2 (Alternative):

```
import java.util.*;
import java.util.stream.*;

List<Integer> nums = Arrays.asList(10,20,30,40,50);
List<Integer> after2 = IntStream.range(0, nums.size())
                .filter(i -> i >= 2)
                .mapToObj(nums::get)
                .collect(Collectors.toList());
System.out.println(after2);
```

Why it works: Index-based filter as an alternative.

21) Limit a stream to the first N elements

Approach 1:

```
import java.util.*;
import java.util.stream.*;

List<Integer> nums = Arrays.asList(10,20,30,40,50);
List<Integer> first3 =
nums.stream().limit(3).collect(Collectors.toList());
System.out.println(first3);
```

Why it works: limit(n) short-circuits after n elements.

Approach 2 (Alternative):

```
import java.util.*;
import java.util.stream.*;

List<Integer> nums = Arrays.asList(10,20,30,40,50);
List<Integer> first3 = IntStream.range(0, Math.min(3, nums.size()))
                      .mapToObj(nums::get)
                      .collect(Collectors.toList());
System.out.println(first3);
```

Why it works: Index slicing alternative.

22) Convert a primitive array to a stream and process it

Approach 1:

```
import java.util.*;
import java.util.stream.*;

int[] arr = {1,2,3,4};
int sum = Arrays.stream(arr).filter(n -> n%2==0).sum();
System.out.println(sum);
```

Why it works: Arrays.stream for primitive arrays yields IntStream.

Approach 2 (Alternative):

```
import java.util.*;
import java.util.stream.*;

int[] arr = {1,2,3,4};
long count = IntStream.of(arr).filter(n -> n%2==0).count();
System.out.println(count);
```

Why it works: IntStream.of is equivalent for int[].

23) Use mapToInt() to sum a list of numbers

Approach 1:

```
import java.util.*;
import java.util.stream.*;

List<Integer> nums = Arrays.asList(5,6,7);
int sum = nums.stream().mapToInt(Integer::intValue).sum();
System.out.println(sum);
```

Why it works: Straightforward primitive stream sum.

Approach 2 (Alternative):

```
import java.util.*;
import java.util.stream.*;

List<Integer> nums = Arrays.asList(5,6,7);
int sum =
nums.stream().collect(Collectors.summingInt(Integer::intValue));
System.out.println(sum);
```

Why it works: Use summingInt collector.

24) Convert a list of lists into a single list (flatMap)

Approach 1:

```
import java.util.*;
import java.util.stream.*;

List<List<Integer>> lol = Arrays.asList(Arrays.asList(1,2),
Arrays.asList(3,4));
List<Integer> flat =
lol.stream().flatMap(List::stream).collect(Collectors.toList());
System.out.println(flat);
```

Why it works: flatMap flattens nested streams.

Approach 2 (Alternative):

```
import java.util.*;
import java.util.stream.*;
```

```
List<List<Integer>> lol = Arrays.asList(Arrays.asList(1,2),
Arrays.asList(3,4));
Set<Integer> flatDistinct = lol.stream()
                          .flatMap(Collection::stream)

.collect(Collectors.toCollection(LinkedHashSet::new));
System.out.println(flatDistinct);
```

Why it works: Flatten + collect with a specific collection type.

25) Filter a list of strings based on length

Approach 1:

```
import java.util.*;
import java.util.stream.*;

List<String> words = Arrays.asList("a","abcd","xyz","hello");
List<String> longOnes = words.stream().filter(s -> s.length() >=
3).collect(Collectors.toList());
System.out.println(longOnes);
```

Why it works: Basic filter on property.

Approach 2 (Alternative):

```
import java.util.*;
import java.util.stream.*;

List<String> words = Arrays.asList("a","abcd","xyz","hello");
List<String> longOnes = words.stream()
                        .collect(Collectors.filtering(s -> s.length() >= 3,
Collectors.toList()));
System.out.println(longOnes);
```

Why it works: Collectors.filtering variant.

26) Generate a list of random numbers using streams

Approach 1:

```
import java.util.*;
import java.util.stream.*;

List<Double> rnd =
Stream.generate(Math::random).limit(5).collect(Collectors.toList());
System.out.println(rnd);
```

Why it works: Stream.generate supplier + limit.

Approach 2 (Alternative):

```
import java.util.*;
import java.util.stream.*;

Random r = new Random();
List<Integer> ints = r.ints(5, 1,
101).boxed().collect(Collectors.toList());
System.out.println(ints);
```

Why it works: Use Random.ints to produce bounded IntStream.

27) Find distinct characters from a string using streams

Approach 1:

```
import java.util.*;
import java.util.stream.*;

String s = "banana";
List<Character> chars = s.chars().mapToObj(c ->
(char)c).distinct().collect(Collectors.toList());
System.out.println(chars);
```

Why it works: chars() -> IntStream, then box to Character.

Approach 2 (Alternative):

```
import java.util.*;
import java.util.stream.*;

String s = "banana";
String unique = s.chars()
        .distinct()
        .collect(StringBuilder::new,
StringBuilder::appendCodePoint, StringBuilder::append)
        .toString();
System.out.println(unique);
```

Why it works: Collect distinct code points back into a String.

28) Partition a list into even and odd numbers

Approach 1:

```
import java.util.*;
import java.util.stream.*;

List<Integer> nums = Arrays.asList(1,2,3,4,5,6);
Map<Boolean, List<Integer>> parts = nums.stream()
                    .collect(Collectors.partitioningBy(n -> n % 2 ==
0));
System.out.println(parts);
```

Why it works: partitioningBy yields two buckets (true/false).

Approach 2 (Alternative):

```
import java.util.*;
import java.util.stream.*;

List<Integer> nums = Arrays.asList(1,2,3,4,5,6);
Map<Boolean, Set<Integer>> parts = nums.stream()
  .collect(Collectors.partitioningBy(n -> n % 2 == 0,
Collectors.toCollection(LinkedHashSet::new)));
System.out.println(parts);
```

Why it works: Downstream collector to control collection type.

29) Remove empty strings from a list

Approach 1:

```
import java.util.*;
import java.util.stream.*;

List<String> words = Arrays.asList("a","","b","  ","c");
List<String> nonEmpty = words.stream().filter(s ->
!s.isEmpty()).collect(Collectors.toList());
System.out.println(nonEmpty);
```

Why it works: Filter by String::isEmpty negation.

Approach 2 (Alternative):

```
import java.util.*;
import java.util.stream.*;
```

```
List<String> words = Arrays.asList("a","","b","  ","c");
List<String> trimmedNonBlank = words.stream()
                    .map(String::trim)
                    .filter(s -> !s.isBlank())
                    .collect(Collectors.toList());
System.out.println(trimmedNonBlank);
```

Why it works: Trim first, then remove blanks (Java 11 String::isBlank).

30) Get the second-largest number in a list

Approach 1:

```
import java.util.*;
import java.util.stream.*;

List<Integer> nums = Arrays.asList(5,9,1,9,3,7);
int second = nums.stream()
         .distinct()
         .sorted(Comparator.reverseOrder())
         .skip(1)
         .findFirst()
         .orElseThrow();
System.out.println(second);
```

Why it works: Distinct to avoid duplicates, reverse sort, skip first.

Approach 2 (Alternative):

```
import java.util.*;
import java.util.stream.*;

List<Integer> nums = Arrays.asList(5,9,1,9,3,7);
int second = nums.stream()
         .collect(Collectors.collectingAndThen(
           Collectors.toCollection(() -> new
TreeSet<>(Comparator.reverseOrder())),
           set -> set.stream().skip(1).findFirst().orElseThrow()
         ));
System.out.println(second);
```

Why it works: Collect to a reversed TreeSet (unique + ordered), then take the second.