Java Most Asked Stream API Coding Questions

1. Filter Even Numbers

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

Solution:

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

Explanation: The `filter` method is used to apply a condition that keeps only even numbers. The `collect` method gathers the results into a new list.

2. Find Maximum

Problem: Find the maximum value in a list of integers.

Solution:

```
Optional<Integer> max = numbers.stream()
                               .max(Integer::compare);
```

Explanation: The `max` method takes a comparator and returns the maximum element wrapped in an `Optional`.

3. Sum of Elements

Problem: Calculate the sum of elements in a list of integers.

Solution:

```
int sum = numbers.stream()
                 .mapToInt(Integer::intValue)
                 .sum();
```

Explanation: `mapToInt` converts the stream to an `IntStream`, which provides the `sum` method to get the total.

4. List of Names to Uppercase

Problem: Convert all strings in a list to uppercase.

Solution:

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

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

Explanation: The `map` function applies `String::toUpperCase` to each element, transforming them to uppercase.

5. Sort List

Problem: Sort a list of integers in ascending order.

Solution:

```
List<Integer> sortedNumbers = numbers.stream()
                                     .sorted()
                                     .collect(Collectors.toList());
```

Explanation: The `sorted` method sorts the elements of the stream in natural order.

6. Count Elements

Problem: Count the number of elements in a list that are greater than 5.

Solution:

```
long count = numbers.stream()
                    .filter(n -> n > 5)
                    .count();
```

Explanation: The `filter` method removes elements that don't satisfy the condition, and `count` returns the number of elements remaining.

7. Get Distinct Elements

Problem: Get a list of distinct elements from a list of integers.

Solution:

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

Explanation: The `distinct` method filters the stream to include only unique elements.

8. Reduce to Sum

Problem: Reduce a list of integers to their sum.

Solution:

```
int total = numbers.stream()
                   .reduce(0, Integer::sum);
```

Explanation: The `reduce` method takes an identity (0 in this case) and an accumulator function (`Integer::sum`) to calculate the total.

9. Find Any

Problem: Return any element from a list of integers.

Solution:

```
Optional<Integer> anyElement = numbers.stream()
                                      .findAny();
```

Explanation: `findAny` potentially returns any element from the stream, wrapped in an `Optional`.

10. List First Names

Problem: Extract first names from a list of full names.

Solution:

```
List<String> fullNames = Arrays.asList("Alice Johnson", "Bob Harris",
"Charlie Lou");
List<String> firstNames = fullNames.stream()
                                   .map(name -> name.split(" ")[0])
                                   .collect(Collectors.toList());
```

Explanation: The `map` function splits each name string and selects the first part.

11. All Match

Problem: Check if all numbers in a list are positive.

Solution:

```
boolean allPositive = numbers.stream()
                             .allMatch(n -> n > 0);
```

Explanation: `allMatch` returns `true` if every element in the stream matches the given predicate.

12. None Match

Problem: Check if there are no negative numbers in a list.

Solution:

```
boolean noneNegative = numbers.stream()
                              .noneMatch(n -> n < 0);
```

Explanation: `noneMatch` checks that no elements match the negative condition.

13. Find First

Problem: Find the first element in a list of integers.

Solution:

```
Optional<Integer> first = numbers.stream()
                                 .findFirst();
```

Explanation: `findFirst` returns the first element of the stream, wrapped in an `Optional`.

14. FlatMap for Nested Lists

Problem: Flatten a nested list structure.

Solution:

```
List<List<Integer>> nestedNumbers = Arrays.asList(Arrays.asList(1, 2),
Arrays.asList(3, 4, 5));
List<Integer> flatList = nestedNumbers.stream()
                                      .flatMap(List::stream)
                                      .collect(Collectors.toList());
```

Explanation: `flatMap` converts each element into its own stream and then merges them into a single stream.

15. Grouping Elements

Problem: Group users by age.

Solution:

```
Map<Integer, List<User>> usersByAge = users.stream()

.collect(Collectors.groupingBy(User::getAge));
```

Explanation: The `groupingBy` collector groups elements based on the age property, creating a map where each key is an age and each value is a list of users with that age.

16. Peek Elements

Problem: Print elements of a stream during processing without altering the stream.

Solution:

```
List<Integer> peekedAtNumbers = numbers.stream()
                                       .peek(System.out::println)
                                       .collect(Collectors.toList());
```

Explanation: `peek` is used for debugging or performing actions without changing the stream. It prints each element before passing it along the stream.

17. Limit Stream

Problem: Limit the output to the first 3 elements of the list.

Solution:

```
List<Integer> limited = numbers.stream()
                              .limit(3)
                              .collect(Collectors.toList());
```

Explanation: `limit` truncates the stream to be no longer than the specified size.

18. Skip Elements

Problem: Skip the first 2 elements of a list and return the rest.

Solution:

```
List<Integer> skipped = numbers.stream()
                              .skip(2)
                              .collect(Collectors.toList());
```

Explanation: `skip` discards the first n elements of the stream.

19. Convert to Set

Problem: Convert a list of integers to a set to remove duplicates.

Solution:

```
Set<Integer> uniqueNumbers = numbers.stream()
                                    .collect(Collectors.toSet());
```

Explanation: Collecting the stream into a `Set` automatically removes duplicates.

20. Summarizing Statistics

Problem: Get summary statistics for a list of integers.

Solution:

```
IntSummaryStatistics stats = numbers.stream()
                                    .mapToInt(Integer::intValue)
                                    .summaryStatistics();
```

Explanation: `summaryStatistics` provides a summary (max, min, average, sum, count) for a stream of integers.