1. **Filter and Collect**: Given a list of integers, filter out all odd numbers and collect the even numbers into a new list.
2. **Map and Collect**: Transform a list of strings to a list of their lengths.
3. **Find First**: Given a list of strings, find the first string that starts with the letter 'A'.
4. **FlatMap**: Given a list of lists of integers, flatten the list into a single list of integers.
5. **Reduce**: Calculate the sum of a list of integers using the reduce method.
6. **Group By**: Group a list of strings by their length.
7. **Partition By**: Partition a list of integers into even and odd numbers.
8. **Limit and Collect**: Get the first 5 even numbers from a list of integers.
9. **Skip and Collect**: Skip the first 3 elements of a list and collect the remaining elements.
10. **Distinct and Collect**: Remove duplicates from a list of integers.
11. **Peek**: Print each element of a list while performing other operations.
12. **Sort**: Sort a list of strings in reverse alphabetical order.
13. **Match**: Check if all elements in a list of integers are positive.
14. **Count**: Count the number of strings in a list that have more than 5 characters.
15. **Min**: Find the minimum value in a list of integers.
16. **Max**: Find the maximum value in a list of integers.
17. **Optional Handling**: Find any element in a list of integers that is greater than 10.
18. **Stream of Objects**: Given a list of employees, find the employee with the highest salary.
19. **Stream of Primitives**: Calculate the average of a list of integers.
20. **Infinite Streams**: Generate a stream of the first 10 even numbers.
21. **Collectors.joining()**: Concatenate a list of strings with a comma separator.
22. **Custom Collector**: Implement a custom collector to collect elements into a TreeSet.
23. **CollectingAndThen**: Collect a list of integers into an unmodifiable list.
24. **ToMap**: Convert a list of strings to a map where the key is the string and the value is its length.
25. **Summarizing**: Get statistics (count, sum, min, average, max) of a list of integers.
26. **Grouping By Multiple Fields**: Group a list of employees by department and then by age.
27. **Partitioning By with Predicate**: Partition a list of students into those who passed and those who failed based on their grades.
28. **Finding nth Largest Element**: Find the second largest element in a list of integers.
29. **Concatenating Streams**: Concatenate two streams into a single stream.
30. **Parallel Stream**: Use parallel stream to improve performance for a large list of numbers by filtering out primes.
31. **Custom Comparator**: Sort a list of employees first by department, then by age.
32. **Filtering Nulls**: Filter out null values from a list of strings.
33. **Stream Builder**: Use Stream.builder() to build a custom stream of integers.
34. **Generating Fibonacci Sequence**: Generate the first 10 Fibonacci numbers using Stream.iterate().
35. **Iterate and Limit**: Generate a sequence of powers of 2 up to 2^10.
36. **Stream from Array**: Create a stream from an array of strings and collect it to a list.
37. **Stream from File**: Read lines from a file and filter out empty lines.
38. **Stream of String**: Convert a list of words into a list of their characters.
39. **Nested Streams**: Process a stream of streams of strings and collect them into a single list of strings.
40. **BiPredicate**: Filter a list of employees where age > 30 and salary > 50000.
41. **Binary Operator**: Combine two lists of integers into one list without duplicates.
42. **Mapping Nested Properties**: Given a list of orders, get a list of all products ordered.
43. **Grouping and Counting**: Group a list of words by their length and count the number of words in each group.
44. **Find the Most Frequent Element**: Find the most frequent element in a list of strings.
45. **Custom Stream Filter**: Implement a custom filter to exclude elements based on a predicate.
46. **Stream of Map Entries**: Convert a map to a stream of entries and collect them into a list.
47. **Creating a Range**: Create a stream of integers from 1 to 100.
48. **Fibonacci with Pair Class**: Generate the first 10 Fibonacci numbers using a custom Pair class.
49. **Recursive Streams**: Use a recursive method to process a stream.
50. **Performance Measurement**: Measure the performance difference between sequential and parallel streams.
51. **Find Duplicates**: Identify duplicate elements in a list of integers.
52. **Longest String**: Find the longest string in a list.
53. **Second Smallest Element**: Find the second smallest element in a list of integers.
54. **Removing Duplicates**: Remove duplicates from a list of objects based on a specific property.
55. **Stream of Future**: Process a list of CompletableFuture objects.
56. **Stream of Optionals**: Flatten a list of Optional objects to a list of non-null values.
57. **Concatenate Strings**: Concatenate a list of strings into a single string with spaces in between.
58. **Summing Int with Map**: Sum the values of a map.
59. **Parallel Streams with ForkJoinPool**: Customize the ForkJoinPool for parallel streams.
60. **Exception Handling in Streams**: Handle checked exceptions in stream operations.
61. **Creating Stream from List of Objects**: Create a stream from a list of custom objects and perform operations.
62. **Using Custom Predicate**: Use a custom predicate to filter a stream.
63. **Stream of Pairs**: Generate a stream of pairs from two lists.
64. **BiFunction**: Apply a function to pairs of elements from two lists.
65. **Combining Predicates**: Combine multiple predicates using and(), or(), and negate().
66. **Finding Top N Elements**: Find the top 3 highest salaries from a list of employees.
67. **Processing Stream of Lines**: Process lines of text from a file stream.
68. **Stream of Indices**: Process a stream of indices of a list.
69. **Stream of Dates**: Generate a stream of dates from a start date to an end date.
70. **Stream of Random Numbers**: Generate a stream of random numbers.
71. **Collect to Array**: Collect a stream to an array.
72. **Reducing with Custom Binary Operator**: Implement a custom binary operator for reducing a stream.
73. **Stream of Bytes**: Process a stream of bytes from an InputStream.
74. **Zipping Streams**: Zip two streams into one stream of pairs.
75. **Stream from CSV File**: Process a CSV file and collect the data into a list of objects.
76. **Stream with BufferedReader**: Use BufferedReader to process a file stream.
77. **Stream of Tuple**: Process a stream of tuples from two lists.
78. **Filtering Objects by Property**: Filter a list of objects based on a specific property value.
79. **Finding the Median**: Find the median of a list of integers.
80. **Grouping by Function Result**: Group a list of numbers by the result of a function applied to them.
81. **Stream of Dates with Intervals**: Generate a stream of dates with a specific interval.
82. **Stream of Prime Numbers**: Generate a stream of prime numbers.
83. **Stream of Factors**: Generate a stream of factors for a given number.
84. **Generating Permutations**: Generate permutations of a list of elements.
85. **Processing Stream of Events**: Process a stream of events in real-time.
86. **Splitting Stream**: Split a stream into multiple sub-streams.
87. **Creating Stream from Supplier**: Use a Supplier to create a stream of objects.
88. **Stream with Multiple Filters**: Apply multiple filters to a stream and collect the result.
89. **Stream of String Tokens**: Tokenize a string and process the tokens using a stream.
90. **Grouping and Reducing**: Group a list of transactions by type and sum the amounts.
91. **Stream from Collection with Custom Class**: Process a collection of a custom class with Stream API.
92. **Lazy Evaluation**: Demonstrate lazy evaluation in a stream pipeline.
93. **Intermediate vs Terminal Operations**: Explain the difference between intermediate and terminal operations in streams.
94. **Transforming Data Structure**: Convert a list of maps to a map of lists.
95. **Stream of Integers from String**: Convert a comma-separated string of numbers to a list of integers.
96. **Stream of Objects to Primitive**: Convert a stream of objects to a stream of primitives.
97. **Stream of Strings to Integers**: Convert a stream of strings to a stream of integers and handle parsing errors.
98. **Stream of Elements with Index**: Process elements of a stream along with their indices.
99. **Using Collector to Combine Streams**: Combine multiple streams into a single stream using a custom collector.
100. **Processing Stream with State**: Maintain state in a stream pipeline to compute cumulative sum.

Java 8 Stream API offers a wide variety of methods that are commonly used for processing collections of objects. Here are some of the most important and frequently used methods:

**Intermediate Operations**

1. **filter(Predicate<? super T> predicate)**
   * Used to select elements based on a condition.
   * Example: **stream.filter(x -> x > 10)**
2. **map(Function<? super T, ? extends R> mapper)**
   * Used to transform each element of the stream.
   * Example: **stream.map(String::toUpperCase)**
3. **flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)**
   * Used to flatten a stream of collections into a single stream.
   * Example: **stream.flatMap(Collection::stream)**
4. **distinct()**
   * Used to remove duplicate elements from the stream.
   * Example: **stream.distinct()**
5. **sorted()**
   * Used to sort the elements in natural order.
   * Example: **stream.sorted()**
6. **sorted(Comparator<? super T> comparator)**
   * Used to sort the elements using a comparator.
   * Example: **stream.sorted(Comparator.reverseOrder())**
7. **limit(long maxSize)**
   * Used to truncate the stream to the given number of elements.
   * Example: **stream.limit(5)**
8. **skip(long n)**
   * Used to skip the first **n** elements of the stream.
   * Example: **stream.skip(2)**

**Terminal Operations**

1. **forEach(Consumer<? super T> action)**
   * Used to perform an action for each element of the stream.
   * Example: **stream.forEach(System.out::println)**
2. **collect(Collector<? super T, A, R> collector)**
   * Used to collect the elements of the stream into a collection.
   * Example: **stream.collect(Collectors.toList())**
3. **toArray()**
   * Used to convert the stream to an array.
   * Example: **stream.toArray()**
4. **reduce(BinaryOperator<T> accumulator)**
   * Used to reduce the elements of the stream to a single value.
   * Example: **stream.reduce(Integer::sum)**
5. **reduce(T identity, BinaryOperator<T> accumulator)**
   * Used to reduce the elements of the stream with an initial value.
   * Example: **stream.reduce(0, Integer::sum)**
6. **count()**
   * Used to count the number of elements in the stream.
   * Example: **stream.count()**
7. **anyMatch(Predicate<? super T> predicate)**
   * Used to check if any elements match a given predicate.
   * Example: **stream.anyMatch(x -> x > 10)**
8. **allMatch(Predicate<? super T> predicate)**
   * Used to check if all elements match a given predicate.
   * Example: **stream.allMatch(x -> x > 10)**
9. **noneMatch(Predicate<? super T> predicate)**
   * Used to check if no elements match a given predicate.
   * Example: **stream.noneMatch(x -> x > 10)**
10. **findFirst()**
    * Used to find the first element of the stream.
    * Example: **stream.findFirst()**
11. **findAny()**
    * Used to find any element of the stream.
    * Example: **stream.findAny()**

These methods form the core of the Stream API and are essential for processing streams in a functional programming style.