50 Java Stream Scenario based Programs with Explanations

1. Find the k most frequent elements in a list

```
int k = 3;

List<Integer> topK = list.stream()

    .collect(Collectors.groupingBy(Function.identity(), Collectors.counting()))

    .entrySet().stream()

    .sorted(Map.Entry.<Integer, Long>comparingByValue().reversed())

    .limit(k)

    .map(Map.Entry::getKey)

    .collect(Collectors.toList());
```

Explanation:

This problem demonstrates how to use Java Streams for: find the k most frequent elements in a list.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

2. Implement a custom collector to get summary statistics of string lengths

```
Collector<String, ?, IntSummaryStatistics> summaryCollector =

    Collectors.summarizingInt(String::length);

IntSummaryStatistics stats = list.stream().collect(summaryCollector);
```

Explanation:

This problem demonstrates how to use Java Streams for: implement a custom collector to get summary statistics of string lengths.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on

the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

3. Find all distinct permutations of characters in each string

```
List<String> permutations = list.stream()

    .flatMap(s -> permute(s).stream())

    .distinct()

    .collect(Collectors.toList());


// Assume permute is a custom method returning all permutations of a string
```

Explanation:

This problem demonstrates how to use Java Streams for: find all distinct permutations of characters in each string.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

4. Batch process a large stream in fixed-size chunks (e.g., 100 items each)

```
AtomicInteger counter = new AtomicInteger();

Map<Integer, List<String>> batches = list.stream()

    .collect(Collectors.groupingBy(i -> counter.getAndIncrement() / 100));
```

Explanation:

This problem demonstrates how to use Java Streams for: batch process a large stream in fixed-size chunks (e.g., 100 items each).

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

5. Create a time-windowed stream from timestamped events

```
Map<Long, List<Event>> windows = events.stream()
    .collect(Collectors.groupingBy(e -> e.getTimestamp() / 60000)); // Group by minute
```

Explanation:

This problem demonstrates how to use Java Streams for: create a time-windowed stream from timestamped events.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

6. Find the longest increasing subsequence using Stream (complex logic)

```
List<Integer> lis = new ArrayList<>();
for (Integer num : list) {
    int i = Collections.binarySearch(lis, num);
    if (i < 0) i = -(i + 1);
    if (i == lis.size()) lis.add(num);
    else lis.set(i, num);
}
```

Explanation:

This problem demonstrates how to use Java Streams for: find the longest increasing subsequence using stream (complex logic).

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

7. Group by multiple fields (e.g., department and designation)

```
Map<List<String>, List<Employee>> grouped = employees.stream()

                .collect(Collectors.groupingBy(e     ->     Arrays.asList(e.getDepartment(),

e.getDesignation())));
```

Explanation:

This problem demonstrates how to use Java Streams for: group by multiple fields (e.g., department and designation).

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

8. Flatten a hierarchical tree structure into a list using recursion and stream

```
List<Node> flatten(Node node) {

    return Stream.concat(Stream.of(node),

            node.getChildren().stream().flatMap(child -> flatten(child).stream()))

            .collect(Collectors.toList());

}
```

Explanation:

This problem demonstrates how to use Java Streams for: flatten a hierarchical tree structure into a list using recursion and stream.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

9. Parallel stream processing with thread-safe accumulation

```
ConcurrentMap<String, Long> result = list.parallelStream()
```

```
                .collect(Collectors.groupingByConcurrent(Function.identity(),
Collectors.counting()));
```

Explanation:

This problem demonstrates how to use Java Streams for: parallel stream processing with thread-safe accumulation.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

10. Custom stream-based DSL for filtering dynamic criteria

```
Predicate<Employee> isManager = e -> e.getRole().equals("Manager");
Predicate<Employee> earnsAbove50k = e -> e.getSalary() > 50000;
List<Employee> filtered = employees.stream()
    .filter(isManager.and(earnsAbove50k))
    .collect(Collectors.toList());
```

Explanation:

This problem demonstrates how to use Java Streams for: custom stream-based dsl for filtering dynamic criteria.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

11. Find common elements from two lists using streams

```
List<Integer> common = list1.stream()
    .filter(list2::contains)
    .collect(Collectors.toList());
```

Explanation:

This problem demonstrates how to use Java Streams for: find common elements from two lists using streams.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

12. Convert a list of strings to a map with string lengths as values

```
Map<String, Integer> map = list.stream()

    .collect(Collectors.toMap(Function.identity(), String::length));
```

Explanation:

This problem demonstrates how to use Java Streams for: convert a list of strings to a map with string lengths as values.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

13. Calculate the product of all integers in a list

```
int product = list.stream()

    .reduce(1, (a, b) -> a * b);
```

Explanation:

This problem demonstrates how to use Java Streams for: calculate the product of all integers in a list.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

14. Partition students based on pass/fail using score > 40

```
Map<Boolean, List<Student>> result = students.stream()
    .collect(Collectors.partitioningBy(s -> s.getScore() > 40));
```

Explanation:

This problem demonstrates how to use Java Streams for: partition students based on pass/fail using score > 40.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

15. Sort a list of employees by department then salary descending

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

Explanation:

This problem demonstrates how to use Java Streams for: sort a list of employees by department then salary descending.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

16. Get all manager names reporting to a particular department

```
List<String> managers = employees.stream()
    .filter(e -> e.getDepartment().equals("Sales") && e.getRole().equals("Manager"))
    .map(Employee::getName)
```

```
    .collect(Collectors.toList());
```

Explanation:

This problem demonstrates how to use Java Streams for: get all manager names reporting to a particular department.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

17. Group books by author and then by genre

```
Map<String, Map<String, List<Book>>> grouped = books.stream()
    .collect(Collectors.groupingBy(Book::getAuthor,
        Collectors.groupingBy(Book::getGenre)));
```

Explanation:

This problem demonstrates how to use Java Streams for: group books by author and then by genre.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

18. Count files by file extension

```
Map<String, Long> countByExt = files.stream()
                    .collect(Collectors.groupingBy(f      ->      getExtension(f.getName()),
Collectors.counting()));
```

Explanation:

This problem demonstrates how to use Java Streams for: count files by file extension.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on

the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

19. Find customers with more than 3 purchases

```
List<Customer> loyal = purchases.stream()
    .collect(Collectors.groupingBy(Purchase::getCustomer, Collectors.counting()))
    .entrySet().stream()
    .filter(e -> e.getValue() > 3)
    .map(Map.Entry::getKey)
    .collect(Collectors.toList());
```

Explanation:

This problem demonstrates how to use Java Streams for: find customers with more than 3 purchases.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

20. Convert a list of dates to strings in dd-MM-yyyy format

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd-MM-yyyy");
List<String> formatted = dates.stream()
    .map(d -> d.format(formatter))
    .collect(Collectors.toList());
```

Explanation:

This problem demonstrates how to use Java Streams for: convert a list of dates to strings in dd-mm-yyyy format.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

21. Filter out non-prime numbers using streams

```
List<Integer> primes = list.stream()
    .filter(n -> IntStream.rangeClosed(2, (int)Math.sqrt(n))
        .allMatch(i -> n % i != 0))
    .collect(Collectors.toList());
```

Explanation:

This problem demonstrates how to use Java Streams for: filter out non-prime numbers using streams.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

22. Find the longest string in a list

```
String longest = list.stream()
    .max(Comparator.comparingInt(String::length))
    .orElse("");
```

Explanation:

This problem demonstrates how to use Java Streams for: find the longest string in a list.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

23. Calculate moving average with window size 3

```
List<Double> movingAvg = IntStream.range(0, list.size() - 2)
    .mapToDouble(i -> (list.get(i) + list.get(i+1) + list.get(i+2)) / 3.0)
```

```
    .boxed().collect(Collectors.toList());
```

Explanation:

This problem demonstrates how to use Java Streams for: calculate moving average with window size 3.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

24. Find all palindromes in a list

```
List<String> palindromes = list.stream()
    .filter(s -> s.equalsIgnoreCase(new StringBuilder(s).reverse().toString()))
    .collect(Collectors.toList());
```

Explanation:

This problem demonstrates how to use Java Streams for: find all palindromes in a list.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

25. Group employees by city and then by team size

```
Map<String, Map<Integer, List<Employee>>> grouped = employees.stream()
    .collect(Collectors.groupingBy(Employee::getCity,
        Collectors.groupingBy(e -> e.getTeam().size())));
```

Explanation:

This problem demonstrates how to use Java Streams for: group employees by city and then by team size.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

26. Normalize and deduplicate emails

```
List<String> cleanEmails = emails.stream()

    .map(String::toLowerCase)

    .map(e -> e.replaceAll("\s", ""))

    .distinct()

    .collect(Collectors.toList());
```

Explanation:

This problem demonstrates how to use Java Streams for: normalize and deduplicate emails.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

27. Get all students whose names start and end with vowels

```
List<String> result = students.stream()

    .map(Student::getName)

    .filter(n -> n.matches("(?i)^[aeiou].*[aeiou]$"))

    .collect(Collectors.toList());
```

Explanation:

This problem demonstrates how to use Java Streams for: get all students whose names start and end with vowels.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

28. Sum the values of a nested list structure

```
int sum = nestedList.stream()
    .flatMap(Collection::stream)
    .mapToInt(Integer::intValue)
    .sum();
```

Explanation:

This problem demonstrates how to use Java Streams for: sum the values of a nested list structure.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

29. Get the earliest and latest transaction dates

```
Optional<LocalDate> min = transactions.stream()
    .map(Transaction::getDate)
    .min(LocalDate::compareTo);
Optional<LocalDate> max = transactions.stream()
    .map(Transaction::getDate)
    .max(LocalDate::compareTo);
```

Explanation:

This problem demonstrates how to use Java Streams for: get the earliest and latest transaction dates.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

30. Convert list of objects to comma-separated values for a field

```
String csv = objects.stream()
```

```
    .map(Object::toString)
    .collect(Collectors.joining(", "));
```

Explanation:

This problem demonstrates how to use Java Streams for: convert list of objects to comma-separated values for a field.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

31. Generate stream of all substrings of a string

```
List<String> substrings = IntStream.range(0, str.length())
    .boxed()
        .flatMap(i -> IntStream.range(i + 1, str.length() + 1).mapToObj(j -> str.substring(i, j)))
    .collect(Collectors.toList());
```

Explanation:

This problem demonstrates how to use Java Streams for: generate stream of all substrings of a string.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

32. Create a frequency map of digit counts in a list of numbers

```
Map<Integer, Long> digitCount = list.stream()
    .flatMap(n -> String.valueOf(n).chars().mapToObj(c -> c - '0'))
    .collect(Collectors.groupingBy(Function.identity(), Collectors.counting()));
```

Explanation:

This problem demonstrates how to use Java Streams for: create a frequency map of digit counts in a list of numbers.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

33. Partition strings by length: short (<5), medium (5-10), long (>10)

```
Map<String, List<String>> partitioned = list.stream()
     .collect(Collectors.groupingBy(s -> s.length() < 5 ? "short" : s.length() <= 10 ?
"medium" : "long"));
```

Explanation:

This problem demonstrates how to use Java Streams for: partition strings by length: short (<5), medium (5-10), long (>10).

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

34. Find repeating characters in a string

```
Map<Character, Long> freq = str.chars()
    .mapToObj(c -> (char)c)
    .collect(Collectors.groupingBy(Function.identity(), Collectors.counting()));
List<Character> repeating = freq.entrySet().stream()
    .filter(e -> e.getValue() > 1)
    .map(Map.Entry::getKey)
    .collect(Collectors.toList());
```

Explanation:

This problem demonstrates how to use Java Streams for: find repeating characters in a string.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

35. Zip two lists into a Map using index

```
Map<String, Integer> zipped = IntStream.range(0, list1.size())
    .boxed()
    .collect(Collectors.toMap(list1::get, list2::get));
```

Explanation:

This problem demonstrates how to use Java Streams for: zip two lists into a map using index.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

36. Find the first recurring element in a list

```
Set<Integer> seen = new HashSet<>();
Optional<Integer> firstRecurring = list.stream()
    .filter(n -> !seen.add(n))
    .findFirst();
```

Explanation:

This problem demonstrates how to use Java Streams for: find the first recurring element in a list.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

37. Extract domain names from a list of emails

```
List<String> domains = emails.stream()
    .map(email -> email.substring(email.indexOf("@") + 1))
    .distinct()
    .collect(Collectors.toList());
```

Explanation:

This problem demonstrates how to use Java Streams for: extract domain names from a list of emails.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

38. Find longest common prefix of a list of strings

```
String prefix = list.stream()
    .reduce((a, b) -> {
        int i = 0;
        while (i < a.length() && i < b.length() && a.charAt(i) == b.charAt(i)) i++;
        return a.substring(0, i);
    }).orElse("");
```

Explanation:

This problem demonstrates how to use Java Streams for: find longest common prefix of a list of strings.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

39. Calculate the variance of a list of integers

```
double avg = list.stream().mapToDouble(i -> i).average().orElse(0);
```

```
double variance = list.stream()
    .mapToDouble(i -> Math.pow(i - avg, 2))
    .average().orElse(0);
```

Explanation:

This problem demonstrates how to use Java Streams for: calculate the variance of a list of integers.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

40. Find the top 3 longest words in a paragraph

```
List<String> top3Words = Arrays.stream(paragraph.split("\s+"))
    .sorted(Comparator.comparingInt(String::length).reversed())
    .limit(3)
    .collect(Collectors.toList());
```

Explanation:

This problem demonstrates how to use Java Streams for: find the top 3 longest words in a paragraph.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

41. Detect circular references in a list of parent-child relationships

```
Set<String> visited = new HashSet<>();
boolean hasCycle = relations.stream()
    .anyMatch(r -> !visited.add(r.getParent() + "->" + r.getChild()));
```

Explanation:

This problem demonstrates how to use Java Streams for: detect circular references in a list of parent-child relationships.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

42. Calculate median of a list using streams

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

double median = list.size() % 2 == 0 ?

    (sorted.get(list.size()/2 - 1) + sorted.get(list.size()/2)) / 2.0 :

    sorted.get(list.size()/2);
```

Explanation:

This problem demonstrates how to use Java Streams for: calculate median of a list using streams.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

43. Create a map of initials to concatenated names

```
Map<Character, String> map = names.stream()

    .collect(Collectors.groupingBy(n -> n.charAt(0),

        Collectors.mapping(Function.identity(), Collectors.joining(", "))));
```

Explanation:

This problem demonstrates how to use Java Streams for: create a map of initials to concatenated names.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

44. Find the second most frequent element

```
String secondMost = list.stream()
    .collect(Collectors.groupingBy(Function.identity(), Collectors.counting()))
    .entrySet().stream()
    .sorted(Map.Entry.<String, Long>comparingByValue().reversed())
    .skip(1).findFirst().map(Map.Entry::getKey).orElse(null);
```

Explanation:

This problem demonstrates how to use Java Streams for: find the second most frequent element.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

45. Identify anagrams in a list of strings

```
Map<String, List<String>> anagrams = list.stream()
    .collect(Collectors.groupingBy(s -> s.chars().sorted()
                        .collect(StringBuilder::new,    StringBuilder::appendCodePoint,
StringBuilder::append).toString()));
```

Explanation:

This problem demonstrates how to use Java Streams for: identify anagrams in a list of strings.

The key Stream operations involved include mapping, filtering, grouping, reducing, or flatMapping based on the use case.

This is useful in real-world applications like analytics, processing, or building backend logic for services.

46. Create a suffix tree-like structure from list of strings

```
Map<String, List<String>> suffixMap = list.stream()
    .flatMap(word -> IntStream.range(0, word.length())
```