List of Java 8 Interview questions

# Conceptual Questions

## What are the main features introduced in Java 8?

* + **Lambda Expressions**: Enable functional programming by writing functions inline.
  + **Stream API**: Process collections in a functional style.
  + **Functional Interfaces**: Interfaces with a single abstract method (e.g., Predicate, Function).
  + **Optional**: Avoid NullPointerException.
  + **Default Methods**: Add default implementations in interfaces.
  + **Date and Time API**: Improved handling of dates and times.
  + **Method References**: Simplified syntax for calling methods.

## What are functional interfaces?

* + Functional interfaces have exactly one abstract method.
  + Support lambda expressions and method references.
  + Examples:
    - Runnable (void run())
    - Predicate<T> (boolean test(T t))
    - Function<T, R> (R apply(T t))

## Explain Stream and its key methods.

* + A Stream represents a sequence of elements for processing.
  + **Intermediate Operations** (return a Stream):
    - filter(): Filter elements based on a condition.
    - map(): Transform elements.
    - sorted(): Sort elements.
  + **Terminal Operations** (consume the Stream):
    - collect(): Convert to a collection.
    - forEach(): Perform an action.
    - reduce(): Aggregate elements.

## What is the difference between map() and flatMap()?

* + **map()**: Transforms each element, returning a stream of streams.
  + **flatMap()**: Transforms and flattens nested structures into a single stream.

**What is Optional in Java 8?**

* + Used to avoid NullPointerException.
  + Methods:
    - of(value): Create an Optional with a non-null value.
    - empty(): Create an empty Optional.
    - ifPresent(): Perform an action if a value is present.

## How do default methods work in interfaces?

* + Add new methods to interfaces with a default implementation.

Example:

interface MyInterface { default void show() {

System.out.println("Default Method");

}

}

## What is the purpose of Collectors?

* + **Collectors** is a utility for reducing streams.
  + Common collectors:
    - toList(), toSet(): Convert to a list or set.
    - joining(): Concatenate strings.
    - groupingBy(): Group elements by a key.
    - partitioningBy(): Partition elements into two groups.

## How does the Date and Time API differ from java.util.Date?

* + Immutable and thread-safe classes: LocalDate, LocalTime, LocalDateTime.
  + DateTimeFormatter for parsing and formatting.
  + Zone-aware classes like ZonedDateTime.

## What are method references in Java 8?

* + A shorthand for lambda expressions.
  + Types:
    - Static methods: Class::methodName
    - Instance methods: instance::methodName
    - Constructors: ClassName::new

## What is parallelStream() in Java 8?

* + Processes elements in parallel for better performance in large datasets.

Example:

List<Integer> numbers = Arrays.asList(1, 2, 3); numbers.parallelStream().map(n -> n \* 2).forEach(System.out::println);

# Coding Problems with Solutions

## Print a list using Lambda Expressions.

List<String> names = Arrays.asList("Alice", "Bob", "Charlie"); names.forEach(name -> System.out.println(name));

1. **Filter even numbers from a list using Streams.** List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6); List<Integer> evens = numbers.stream()

.filter(n -> n % 2 == 0)

.collect(Collectors.toList()); System.out.println(evens); // Output: [2, 4, 6]

1. **Find the maximum value in a list using Streams.** List<Integer> numbers = Arrays.asList(10, 20, 30, 40); int max = numbers.stream()

.max(Integer::compare)

.orElse(0); System.out.println(max); // Output: 40

1. **Convert a list of strings to uppercase.** List<String> names = Arrays.asList("alice", "bob"); List<String> upperNames = names.stream()

.map(String::toUpperCase)

.collect(Collectors.toList()); System.out.println(upperNames); // Output: [ALICE, BOB]

1. **Group strings by their length using groupingBy().** List<String> names = Arrays.asList("Alice", "Bob", "Charlie"); Map<Integer, List<String>> grouped = names.stream()

.collect(Collectors.groupingBy(String::length));

System.out.println(grouped); // Output: {3=[Bob], 5=[Alice], 7=[Charlie]}

1. **Find the sum of numbers using reduce().** List<Integer> numbers = Arrays.asList(1, 2, 3, 4); int sum = numbers.stream()

.reduce(0, Integer::sum);

System.out.println(sum); // Output: 10

1. **Count word occurrences in a list using groupingBy().** List<String> words = Arrays.asList("apple", "banana", "apple"); Map<String, Long> wordCount = words.stream()

.collect(Collectors.groupingBy(w -> w, Collectors.counting()));

System.out.println(wordCount); // Output: {apple=2, banana=1}

## Concatenate strings using joining().

List<String> words = Arrays.asList("Java", "is", "awesome");

String sentence = words.stream()

.collect(Collectors.joining(" ")); System.out.println(sentence); // Output: Java is awesome

## Sort employees by salary.

class Employee { String name; int salary;

Employee(String name, int salary) { this.name = name;

this.salary = salary;

}

public String toString() { return name + ": " + salary;

}

}

List<Employee> employees = Arrays.asList( new Employee("Alice", 5000),

new Employee("Bob", 3000),

new Employee("Charlie", 4000)

);

List<Employee> sorted = employees.stream()

.sorted(Comparator.comparingInt(e -> e.salary))

.collect(Collectors.toList());

System.out.println(sorted); // Output: [Bob: 3000, Charlie: 4000, Alice: 5000]

## Find the first non-repeated character in a string.

String input = "swiss"; Character result = input.chars()

.mapToObj(c -> (char) c)

.filter(ch -> input.indexOf(ch) == input.lastIndexOf(ch))

.findFirst()

.orElse(null); System.out.println(result); // Output: w

## What is the difference between Stream.findFirst() and Stream.findAny()?

* + **findFirst()**:
    - Returns the first element of the Stream.
    - Suitable for ordered Streams.

## findAny():

* + - Returns any element of the Stream.
    - Suitable for parallel Streams where order doesn't matter.

## What are the different types of Streams in Java 8?

* + **Sequential Stream**:
    - Processes elements sequentially in a single thread.

## Parallel Stream:

* + - Processes elements in multiple threads for faster computation.

## Can we use multiple filters in a single Stream?

Yes, you can chain multiple filters:

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5); numbers.stream()

.filter(n -> n > 2)

.filter(n -> n % 2 == 0)

.forEach(System.out::println); // Output: 4

●

## Explain reduce() in Java 8 Streams with an example.

* + **reduce()** is used for aggregation, like summing or concatenating elements.

Example:

List<Integer> numbers = Arrays.asList(1, 2, 3, 4); int sum = numbers.stream()

.reduce(0, Integer::sum); // Start with 0 System.out.println(sum); // Output: 10

## How does Java 8 handle default methods in case of multiple inheritance?

* + If multiple interfaces provide the same default method:
    - The class must override the method to resolve the conflict.

Example:

interface A {

default void display() { System.out.println("A");

}

}

interface B {

default void display() { System.out.println("B");

}

}

class C implements A, B { public void display() {

A.super.display(); // Choose A's display method

}

}

## What are some best practices for using Streams in Java 8?

* + Avoid using Streams for small collections (traditional loops are better).
  + Use **parallelStream()** only when working with large datasets.
  + Prefer **method references** over complex lambda expressions for readability.
  + Use terminal operations (collect, reduce) to consume the Stream.

# Coding Problems

1. **Use Stream.distinct() to remove duplicates from a list.** List<Integer> numbers = Arrays.asList(1, 2, 2, 3, 4, 4, 5); List<Integer> distinctNumbers = numbers.stream()

.distinct()

.collect(Collectors.toList()); System.out.println(distinctNumbers); // Output: [1, 2, 3, 4, 5]

## Find all elements starting with "A" in a list.

List<String> names = Arrays.asList("Alice", "Bob", "Annie", "Alex"); List<String> filteredNames = names.stream()

.filter(name -> name.startsWith("A"))

.collect(Collectors.toList()); System.out.println(filteredNames); // Output: [Alice, Annie, Alex]

1. **Sort a list of strings alphabetically and in reverse order.** List<String> names = Arrays.asList("Charlie", "Alice", "Bob"); List<String> sortedNames = names.stream()

.sorted() // Ascending

.collect(Collectors.toList()); System.out.println(sortedNames); // Output: [Alice, Bob, Charlie]

List<String> reversedNames = names.stream()

.sorted(Comparator.reverseOrder()) // Descending

.collect(Collectors.toList()); System.out.println(reversedNames); // Output: [Charlie, Bob, Alice]

## Flatten a list of lists using flatMap().

List<List<Integer>> nestedList = Arrays.asList( Arrays.asList(1, 2, 3),

Arrays.asList(4, 5),

Arrays.asList(6, 7, 8)

);

List<Integer> flatList = nestedList.stream()

.flatMap(List::stream)

.collect(Collectors.toList()); System.out.println(flatList); // Output: [1, 2, 3, 4, 5, 6, 7, 8]

## Use Collectors.partitioningBy() to separate even and odd numbers.

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6); Map<Boolean, List<Integer>> partitioned = numbers.stream()

.collect(Collectors.partitioningBy(n -> n % 2 == 0));

System.out.println(partitioned); // Output: {false=[1, 3, 5], true=[2, 4, 6]}

1. **Find the second highest number in a list.** List<Integer> numbers = Arrays.asList(10, 20, 30, 40, 50); int secondHighest = numbers.stream()

.sorted(Comparator.reverseOrder())

.skip(1) // Skip the highest

.findFirst()

.orElseThrow(() -> new RuntimeException("No second highest found")); System.out.println(secondHighest); // Output: 40

## Count the frequency of characters in a string using Streams.

String input = "java";

Map<Character, Long> frequency = input.chars()

.mapToObj(c -> (char) c)

.collect(Collectors.groupingBy(c -> c, Collectors.counting())); System.out.println(frequency); // Output: {a=2, j=1, v=1}

## Generate an infinite Stream of even numbers and limit it to 10 elements.

List<Integer> evenNumbers = Stream.iterate(0, n -> n + 2)

.limit(10)

.collect(Collectors.toList()); System.out.println(evenNumbers); // Output: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

## Check if all elements in a list are greater than a given number.

List<Integer> numbers = Arrays.asList(10, 20, 30, 40); boolean allGreater = numbers.stream()

.allMatch(n -> n > 5); System.out.println(allGreater); // Output: true

1. **Find the average of a list of numbers.** List<Integer> numbers = Arrays.asList(10, 20, 30, 40); double average = numbers.stream()

.mapToInt(Integer::intValue)

.average()

.orElse(0.0); System.out.println(average); // Output: 25.0

1. **Generate the Fibonacci series using Stream.iterate().**

Stream.iterate(new int[]{0, 1}, f -> new int[]{f[1], f[0] + f[1]})

.limit(10)

.map(f -> f[0])

.forEach(System.out::print); // Output: 01123581321

1. **Group employees by department using Collectors.groupingBy().**

class Employee { String name; String department;

Employee(String name, String department) { this.name = name;

this.department = department;

}

}

List<Employee> employees = Arrays.asList(

new Employee("Alice", "HR"),

new Employee("Bob", "IT"), new Employee("Charlie", "HR"), new Employee("David", "IT")

);

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

.collect(Collectors.groupingBy(emp ->

emp.department));

groupedByDepartment.forEach((dept, emps) -> { System.out.println(dept + ": " + emps.stream().map(e ->

e.name).collect(Collectors.toList()));

});

1. **Count occurrences of each word in a sentence.**

String sentence = "Java is fun and Java is powerful";

Map<String, Long> wordCount = Arrays.stream(sentence.split(" "))

.collect(Collectors.groupingBy(word -> word, Collectors.counting()));

System.out.println(wordCount); // Output: {Java=2, is=2, fun=1, and=1, powerful=1}

1. **Find the longest word in a list.**

List<String> words = Arrays.asList("apple", "banana", "cherry", "date"); String longestWord = words.stream()

.max(Comparator.comparingInt(String::length))

.orElse(null); System.out.println(longestWord); // Output: banana

1. **Merge two lists into a single list using flatMap().**

List<Integer> list1 = Arrays.asList(1, 2, 3);

List<Integer> list2 = Arrays.asList(4, 5, 6); List<Integer> mergedList = Stream.of(list1, list2)

.flatMap(List::stream)

.collect(Collectors.toList()); System.out.println(mergedList); // Output: [1, 2, 3, 4, 5, 6]

1. **Find the first element in a Stream greater than 10.**

List<Integer> numbers = Arrays.asList(5, 8, 12, 3, 20); int first = numbers.stream()

.filter(n -> n > 10)

.findFirst()

.orElse(-1); System.out.println(first); // Output: 12

1. **Find the minimum value in a list using Streams.**

List<Integer> numbers = Arrays.asList(10, 20, 5, 15); int min = numbers.stream()

.min(Integer::compareTo)

.orElseThrow(() -> new RuntimeException("No minimum value

found"));

System.out.println(min); // Output: 5

1. **Use Stream.generate() to create a list of random numbers.**

List<Double> randomNumbers = Stream.generate(Math::random)

.limit(5)

.collect(Collectors.toList()); System.out.println(randomNumbers);

1. **Find duplicate elements in a list using Streams.**

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 2, 3); Set<Integer> duplicates = numbers.stream()

.filter(n -> Collections.frequency(numbers, n) > 1)

.collect(Collectors.toSet()); System.out.println(duplicates); // Output: [2, 3]

1. **Partition a list into prime and non-prime numbers.**

List<Integer> numbers = Arrays.asList(2, 3, 4, 5, 6, 7, 8, 9, 10); Map<Boolean, List<Integer>> partitioned = numbers.stream()

.collect(Collectors.partitioningBy(num ->

isPrime(num)));

System.out.println(partitioned); static boolean isPrime(int num) {

if (num <= 1) return false;

return IntStream.rangeClosed(2, (int) Math.sqrt(num)).noneMatch(n -> num % n == 0);

}

1. **Use Stream.flatMap() to process nested collections.**

List<List<String>> nestedList = Arrays.asList( Arrays.asList("Alice", "Bob"), Arrays.asList("Charlie", "David")

);

List<String> flatList = nestedList.stream()

.flatMap(List::stream)

.collect(Collectors.toList()); System.out.println(flatList); // Output: [Alice, Bob, Charlie, David]

1. **Calculate the factorial of a number using Streams.**

int number = 5;

int factorial = IntStream.rangeClosed(1, number)

.reduce(1, (a, b) -> a \* b); System.out.println(factorial); // Output: 120

1. **Use Stream.skip() and Stream.limit() to extract sublists.**

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7); List<Integer> sublist = numbers.stream()

.skip(2) // Skip the first 2 elements

.limit(3) // Take the next 3 elements

.collect(Collectors.toList()); System.out.println(sublist); // Output: [3, 4, 5]

1. **Use Collectors.teeing() to compute two operations on a Stream.**

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5); Map<String, Double> result = numbers.stream()

.collect(Collectors.teeing( Collectors.summingDouble(n -> n), Collectors.averagingDouble(n -> n),

(sum, avg) -> Map.of("Sum", sum, "Average", avg)

));

System.out.println(result); // Output: {Sum=15.0, Average=3.0}

1. **Find all palindromic strings in a list.**

List<String> words = Arrays.asList("madam", "racecar", "java", "level", "hello");

List<String> palindromes = words.stream()

.filter(word -> word.equals(new StringBuilder(word).reverse().toString()))

.collect(Collectors.toList()); System.out.println(palindromes); // Output: [madam, racecar, level]