**I need java stream 300 problems for interviews and knowledge purpose. Give Beginner, Intermediate And Advanced level questions.**

**Thought for 4 seconds**

**300 Java Stream Problems for Interviews**

**Introduction to Java Streams**

**Java Streams, introduced in Java 8, provide a powerful way to process collections of objects. They enable functional-style operations on streams of elements, such as map-reduce transformations on collections.**

**Incident🡪log check ->lower env replicate->issue root cause finds in lower env -> sync code with production**

**Production env fail-> env issue->**

**Dev team will reach**

**Change request ->script request rollback roll out->central advice approve the request**

**Problem issue same-> problem task assign ->code issue**

**Db query ->joins inner query outer query 2nd highest salary**

**Core java basic**

**Java8 feature**

**## Beginner Level Problems (100)**

**### Basic Stream Creation and Simple Operations**

**1. \*\*Create a Stream of integers from 1 to 10 and print each element.\*\***

**IntStream.rangeClosed(1, 10).forEach(System.out::println);**

**```**

**2. \*\*Convert an array of strings to a Stream and print each element.\*\***

**```java**

**String[] array = {"Java", "Stream", "API"};**

**Arrays.stream(array).forEach(System.out::println);**

**```**

**### 300 Java Stream Problems for Interviews**

**## Introduction to Java Streams**

**Java Streams, introduced in Java 8, provide a powerful way to process collections of objects. They enable functional-style operations on streams of elements, such as map-reduce transformations on collections.**

**## Beginner Level Problems (100)**

**### Basic Stream Creation and Simple Operations**

**1. \*\*Create a Stream of integers from 1 to 10 and print each element.\*\***

**```java**

**IntStream.rangeClosed(1, 10).forEach(System.out::println);**

**```**

**2. \*\*Convert an array of strings to a Stream and print each element.\*\***

**```java**

**String[] array = {"Java", "Stream", "API"};**

**Arrays.stream(array).forEach(System.out::println);**

**```**

**3. \*\*Create an empty Stream and check if it has any elements.\*\***

**```java**

**boolean isEmpty = Stream.empty().findAny().isEmpty();**

**System.out.println("Stream is empty: " + isEmpty);**

**```**

**4. \*\*Create a Stream from a List and convert it back to a List.\*\***

**```java**

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

**List<String> result = list.stream().collect(Collectors.toList());**

**```**

**5. \*\*Create a Stream of 5 random integers between 1 and 100.\*\***

**```java**

**Random random = new Random();**

**random.ints(5, 1, 101).forEach(System.out::println);**

**```**

**6. \*\*Create a Stream from a String, splitting it by spaces.\*\***

**```java**

**String str = "Java Stream API";**

**Arrays.stream(str.split(" ")).forEach(System.out::println);**

**```**

**7. \*\*Create a Stream of the first 10 even numbers.\*\***

**```java**

**IntStream.iterate(2, n -> n + 2).limit(10).forEach(System.out::println);**

**```**

**8. \*\*Create a Stream of characters from a String.\*\***

**```java**

**String str = "stream";**

**str.chars().mapToObj(c -> (char) c).forEach(System.out::println);**

**```**

**9. \*\*Create a Stream from a file and print each line.\*\***

**```java**

**try (Stream<String> lines = Files.lines(Paths.get("file.txt"))) {**

**lines.forEach(System.out::println);**

**} catch (IOException e) {**

**e.printStackTrace();**

**}**

**```**

**10. \*\*Create a Stream from a Set and print each element.\*\***

**```java**

**Set<String> set = new HashSet<>(Arrays.asList("a", "b", "c"));**

**set.stream().forEach(System.out::println);**

**```**

**### Filtering and Mapping**

**11. \*\*Filter a list of integers to get only even numbers.\*\***

**```java**

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

**List<Integer> evenNumbers = numbers.stream()**

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

**.collect(Collectors.toList());**

**```**

**12. \*\*Filter a list of strings to get only those with length greater than 3.\*\***

**```java**

**List<String> words = Arrays.asList("hi", "hello", "bye", "goodbye");**

**List<String> longWords = words.stream()**

**.filter(s -> s.length() > 3)**

**.collect(Collectors.toList());**

**```**

**13. \*\*Convert a list of strings to uppercase.\*\***

**```java**

**List<String> words = Arrays.asList("java", "stream", "api");**

**List<String> upperCaseWords = words.stream()**

**.map(String::toUpperCase)**

**.collect(Collectors.toList());**

**```**

**14. \*\*Extract the lengths of a list of strings.\*\***

**```java**

**List<String> words = Arrays.asList("java", "stream", "api");**

**List<Integer> lengths = words.stream()**

**.map(String::length)**

**.collect(Collectors.toList());**

**```**

**15. \*\*Filter out null values from a list.\*\***

**```java**

**List<String> list = Arrays.asList("a", null, "b", null, "c");**

**List<String> nonNullList = list.stream()**

**.filter(Objects::nonNull)**

**.collect(Collectors.toList());**

**```**

**16. \*\*Filter a list of integers to get numbers greater than 5.\*\***

**```java**

**List<Integer> numbers = Arrays.asList(2, 5, 8, 10, 3, 1);**

**List<Integer> greaterThan5 = numbers.stream()**

**.filter(n -> n > 5)**

**.collect(Collectors.toList());**

**```**

**17. \*\*Filter a list of strings to get only those starting with 'a'.\*\***

**```java**

**List<String> words = Arrays.asList("apple", "banana", "apricot", "cherry");**

**List<String> aWords = words.stream()**

**.filter(s -> s.startsWith("a"))**

**.collect(Collectors.toList());**

**```**

**18. \*\*Map a list of integers to their squares.\*\***

**```java**

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

**List<Integer> squares = numbers.stream()**

**.map(n -> n \* n)**

**.collect(Collectors.toList());**

**```**

**19. \*\*Filter a list of integers to get only odd numbers and then double them.\*\***

**```java**

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

**List<Integer> doubledOdds = numbers.stream()**

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

**.map(n -> n \* 2)**

**.collect(Collectors.toList());**

**```**

**20. \*\*Filter a list of strings to get only those containing the letter 'a'.\*\***

**```java**

**List<String> words = Arrays.asList("java", "stream", "api", "lambda");**

**List<String> aWords = words.stream()**

**.filter(s -> s.contains("a"))**

**.collect(Collectors.toList());**

**```**

**### Sorting and Limiting**

**21. \*\*Sort a list of integers in ascending order.\*\***

**```java**

**List<Integer> numbers = Arrays.asList(5, 3, 8, 1, 9, 2);**

**List<Integer> sortedNumbers = numbers.stream()**

**.sorted()**

**.collect(Collectors.toList());**

**```**

**22. \*\*Sort a list of strings by length.\*\***

**```java**

**List<String> words = Arrays.asList("java", "stream", "api", "lambda");**

**List<String> sortedByLength = words.stream()**

**.sorted(Comparator.comparing(String::length))**

**.collect(Collectors.toList());**

**```**

**23. \*\*Sort a list of strings in reverse alphabetical order.\*\***

**```java**

**List<String> words = Arrays.asList("java", "stream", "api", "lambda");**

**List<String> reverseSorted = words.stream()**

**.sorted(Comparator.reverseOrder())**

**.collect(Collectors.toList());**

**```**

**24. \*\*Get the first 3 elements from a list.\*\***

**```java**

**List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);**

**List<Integer> firstThree = numbers.stream()**

**.limit(3)**

**.collect(Collectors.toList());**

**```**

**25. \*\*Skip the first 3 elements from a list and get the rest.\*\***

**```java**

**List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);**

**List<Integer> afterThree = numbers.stream()**

**.skip(3)**

**.collect(Collectors.toList());**

**```**

**26. \*\*Get the 3 smallest numbers from a list.\*\***

**```java**

**List<Integer> numbers = Arrays.asList(5, 3, 8, 1, 9, 2);**

**List<Integer> smallestThree = numbers.stream()**

**.sorted()**

**.limit(3)**

**.collect(Collectors.toList());**

**```**

**27. \*\*Get the 3 longest strings from a list.\*\***

**```java**

**List<String> words = Arrays.asList("java", "stream", "api", "lambda", "functional");**

**List<String> longestThree = words.stream()**

**.sorted(Comparator.comparing(String::length).reversed())**

**.limit(3)**

**.collect(Collectors.toList());**

**```**

**28. \*\*Sort a list of integers in descending order.\*\***

**```java**

**List<Integer> numbers = Arrays.asList(5, 3, 8, 1, 9, 2);**

**List<Integer> descendingOrder = numbers.stream()**

**.sorted(Comparator.reverseOrder())**

**.collect(Collectors.toList());**

**```**

**29. \*\*Sort a list of strings by their last character.\*\***

**```java**

**List<String> words = Arrays.asList("java", "stream", "api", "lambda");**

**List<String> sortedByLastChar = words.stream()**

**.sorted(Comparator.comparing(s -> s.charAt(s.length() - 1)))**

**.collect(Collectors.toList());**

**```**

**30. \*\*Get the elements from a list after skipping the first 2 and limiting to the next 3.\*\***

**```java**

**List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);**

**List<Integer> result = numbers.stream()**

**.skip(2)**

**.limit(3)**

**.collect(Collectors.toList());**

**```**

**### Finding and Matching**

**31. \*\*Check if a list contains any even number.\*\***

**```java**

**List<Integer> numbers = Arrays.asList(1, 3, 5, 7, 8, 9);**

**boolean hasEven = numbers.stream().anyMatch(n -> n % 2 == 0);**

**```**

**32. \*\*Check if all numbers in a list are positive.\*\***

**```java**

**List<Integer> numbers = Arrays.asList(1, 3, 5, 7, 8, 9);**

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

**```**

**33. \*\*Check if no number in a list is divisible by 10.\*\***

**```java**

**List<Integer> numbers = Arrays.asList(1, 3, 5, 7, 8, 9);**

**boolean noDivisibleBy10 = numbers.stream().noneMatch(n -> n % 10 == 0);**

**```**

**34. \*\*Find the first even number in a list.\*\***

**```java**

**List<Integer> numbers = Arrays.asList(1, 3, 5, 6, 7, 8, 9);**

**Optional<Integer> firstEven = numbers.stream().filter(n -> n % 2 == 0).findFirst();**

**```**

**35. \*\*Find any string in a list that starts with 'a'.\*\***

**```java**

**List<String> words = Arrays.asList("banana", "apple", "cherry", "apricot");**

**Optional<String> anyA = words.stream().filter(s -> s.startsWith("a")).findAny();**

**```**

**36. \*\*Check if a list contains a specific string.\*\***

**```java**

**List<String> words = Arrays.asList("java", "stream", "api", "lambda");**

**boolean containsStream = words.stream().anyMatch("stream"::equals);**

**```**

**37. \*\*Find the first string in a list with length greater than 5.\*\***

**```java**

**List<String> words = Arrays.asList("java", "stream", "api", "lambda", "functional");**

**Optional<String> firstLong = words.stream().filter(s -> s.length() > 5).findFirst();**

**```**

**38. \*\*Check if all strings in a list have length less than 10.\*\***

**```java**

**List<String> words = Arrays.asList("java", "stream", "api", "lambda", "functional");**

**boolean allShort = words.stream().allMatch(s -> s.length() < 10);**

**```**

**39. \*\*Check if no string in a list is empty.\*\***

**```java**

**List<String> words = Arrays.asList("java", "stream", "api", "lambda");**

**boolean noEmpty = words.stream().noneMatch(String::isEmpty);**

**```**

**40. \*\*Find any number in a list that is greater than 10.\*\***

**```java**

**List<Integer> numbers = Arrays.asList(5, 3, 8, 12, 9, 2);**

**Optional<Integer> anyGreaterThan10 = numbers.stream().filter(n -> n > 10).findAny();**

**```**

**### Reduction and Aggregation**

**41. \*\*Find the sum of all integers in a list.\*\***

**```java**

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

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

**// Alternative: int sum = numbers.stream().reduce(0, Integer::sum);**

**```**

**42. \*\*Find the product of all integers in a list.\*\***

**```java**

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

**int product = numbers.stream().reduce(1, (a, b) -> a \* b);**

**```**

**43. \*\*Find the maximum value in a list of integers.\*\***

**```java**

**List<Integer> numbers = Arrays.asList(5, 3, 8, 1, 9, 2);**

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

**// Alternative: Optional<Integer> max = numbers.stream().reduce(Integer::max);**

**```**

**44. \*\*Find the minimum value in a list of integers.\*\***

**```java**

**List<Integer> numbers = Arrays.asList(5, 3, 8, 1, 9, 2);**

**Optional<Integer> min = numbers.stream().min(Integer::compare);**

**// Alternative: Optional<Integer> min = numbers.stream().reduce(Integer::min);**

**```**

**45. \*\*Calculate the average of a list of integers.\*\***

**```java**

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

**double average = numbers.stream().mapToInt(Integer::intValue).average().orElse(0.0);**

**```**

**46. \*\*Count the number of elements in a stream.\*\***

**```java**

**List<String> words = Arrays.asList("java", "stream", "api", "lambda");**

**long count = words.stream().count();**

**```**

**47. \*\*Concatenate all strings in a list with a comma separator.\*\***

**```java**

**List<String> words = Arrays.asList("java", "stream", "api", "lambda");**

**String result = words.stream().collect(Collectors.joining(", "));**

**```**

**48. \*\*Find the sum of the lengths of all strings in a list.\*\***

**```java**

**List<String> words = Arrays.asList("java", "stream", "api", "lambda");**

**int totalLength = words.stream().mapToInt(String::length).sum();**

**```**

**49. \*\*Find the longest string in a list.\*\***

**```java**

**List<String> words = Arrays.asList("java", "stream", "api", "lambda", "functional");**

**Optional<String> longest = words.stream().max(Comparator.comparing(String::length));**

**```**

**50. \*\*Find the shortest string in a list.\*\***

**```java**

**List<String> words = Arrays.asList("java", "stream", "api", "lambda", "functional");**

**Optional<String> shortest = words.stream().min(Comparator.comparing(String::length));**

**```**

**### Collecting Results**

**51. \*\*Collect a stream of integers into an array.\*\***

**```java**

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

**Integer[] array = numbers.stream().toArray(Integer[]::new);**

**```**

**52. \*\*Collect a stream of strings into a Set to remove duplicates.\*\***

**```java**

**List<String> words = Arrays.asList("java", "stream", "java", "api", "stream");**

**Set<String> uniqueWords = words.stream().collect(Collectors.toSet());**

**```**

**53. \*\*Collect a stream of strings into a LinkedList.\*\***

**```java**

**List<String> words = Arrays.asList("java", "stream", "api", "lambda");**

**LinkedList<String> linkedList = words.stream().collect(Collectors.toCollection(LinkedList::new));**

**```**

**54. \*\*Create a comma-separated string from a list of integers.\*\***

**```java**

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

**String result = numbers.stream()**

**.map(String::valueOf)**

**.collect(Collectors.joining(", "));**

**```**

**55. \*\*Group a list of strings by their length.\*\***

**```java**

**List<String> words = Arrays.asList("java", "stream", "api", "lambda", "functional");**

**Map<Integer, List<String>> groupedByLength = words.stream()**

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

**```**

**56. \*\*Partition a list of integers into even and odd numbers.\*\***

**```java**

**List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);**

**Map<Boolean, List<Integer>> evenAndOdd = numbers.stream()**

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

**```**

**57. \*\*Count occurrences of each character in a string.\*\***

**```java**

**String text = "hello world";**

**Map<Character, Long> charFrequency = text.chars()**

**.mapToObj(c -> (char) c)**

**.collect(Collectors.groupingBy(c -> c, Collectors.counting()));**

**```**

**58. \*\*Create a Map from a list of strings, with the string as key and its length as value.\*\***

**```java**

**List<String> words = Arrays.asList("java", "stream", "api", "lambda");**

**Map<String, Integer> lengthMap = words.stream()**

**.collect(Collectors.toMap(w -> w, String::length));**

**```**

**59. \*\*Find the frequency of each element in a list.\*\***

**```java**

**List<String> words = Arrays.asList("java", "stream", "java", "api", "stream");**

**Map<String, Long> frequency = words.stream()**

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

**```**

**60. \*\*Collect a stream of pairs into a Map.\*\***

**```java**

**record Pair(String key, Integer value) {}**

**List<Pair> pairs = Arrays.asList(**

**new Pair("a", 1),**

**new Pair("b", 2),**

**new Pair("c", 3)**

**);**

**Map<String, Integer> map = pairs.stream()**

**.collect(Collectors.toMap(Pair::key, Pair::value));**

**```**

**### Primitive Streams**

**61. \*\*Create an IntStream of numbers from 1 to 10 and calculate their sum.\*\***

**```java**

**int sum = IntStream.rangeClosed(1, 10).sum();**

**```**

**62. \*\*Create a DoubleStream of 5 random values and find their average.\*\***

**```java**

**double average = new Random().doubles(5).average().orElse(0.0);**

**```**

**63. \*\*Convert a list of strings to an IntStream of their lengths.\*\***

**```java**

**List<String> words = Arrays.asList("java", "stream", "api", "lambda");**

**IntStream lengthStream = words.stream().mapToInt(String::length);**

**```**

**64. \*\*Generate a LongStream of the first 10 powers of 2.\*\***

**```java**

**LongStream powers = LongStream.iterate(1, n -> n \* 2).limit(10);**

**```**

**65. \*\*Create an IntStream of even numbers from 1 to 100.\*\***

**```java**

**IntStream evenNumbers = IntStream.rangeClosed(1, 100).filter(n -> n % 2 == 0);**

**```**

**66. \*\*Convert an IntStream to a Stream of Integer objects.\*\***

**```java**

**Stream<Integer> boxedStream = IntStream.range(1, 10).boxed();**

**```**

**67. \*\*Find the sum of squares of numbers from 1 to 10.\*\***

**```java**

**int sumOfSquares = IntStream.rangeClosed(1, 10).map(n -> n \* n).sum();**

**```**

**68. \*\*Create a DoubleStream of 5 values and convert it to an array.\*\***

**```java**

**double[] array = DoubleStream.of(1.1, 2.2, 3.3, 4.4, 5.5).toArray();**

**```**

**69. \*\*Find the product of all numbers from 1 to 5 using an IntStream.\*\***

**```java**

**int product = IntStream.rangeClosed(1, 5).reduce(1, (a, b) -> a \* b);**

**```**

**70. \*\*Create a stream of characters from a string and count the vowels.\*\***

**```java**

**String str = "hello world";**

**long vowelCount = str.chars()**

**.mapToObj(c -> (char) c)**

**.filter(c -> "aeiouAEIOU".indexOf(c) >= 0)**

**.count();**

**```**

**### Practical Problems**

**71. \*\*Remove duplicate elements from a list.\*\***

**```java**

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

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

**```**

**72. \*\*Find all pairs of integers in a list whose sum is a given value.\*\***

**```java**

**List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);**

**int targetSum = 10;**

**List<String> pairs = IntStream.range(0, numbers.size())**

**.boxed()**

**.flatMap(i -> IntStream.range(i + 1, numbers.size())**

**.filter(j -> numbers.get(i) + numbers.get(j) == targetSum)**

**.mapToObj(j -> "(" + numbers.get(i) + "," + numbers.get(j) + ")"))**

**.collect(Collectors.toList());**

**```**

**73. \*\*Count the number of words in a string.\*\***

**```java**

**String text = "This is a sample text with several words";**

**long wordCount = Arrays.stream(text.split("\\s+")).count();**

**```**

**74. \*\*Find the most frequent word in a text.\*\***

**```java**

**String text = "this is a sample text this is another sample";**

**String mostFrequent = Arrays.stream(text.split("\\s+"))**

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

**.entrySet().stream()**

**.max(Map.Entry.comparingByValue())**

**.map(Map.Entry::getKey)**

**.orElse("");**

**```**

**75. \*\*Calculate the factorial of a number using a stream.\*\***

**```java**

**int n = 5;**

**long factorial = IntStream.rangeClosed(1, n).reduce(1, (a, b) -> a \* b);**

**```**

**76. \*\*Check if a string is a palindrome using streams.\*\***

**```java**

**String str = "radar";**

**boolean isPalindrome = IntStream.range(0, str.length() / 2)**

**.allMatch(i -> str.charAt(i) == str.charAt(str.length() - i - 1));**

**```**

**77. \*\*Find the second largest number in a list.\*\***

**```java**

**List<Integer> numbers = Arrays.asList(5, 9, 11, 2, 8, 21, 1);**

**Optional<Integer> secondLargest = numbers.stream()**

**.sorted(Comparator.reverseOrder())**

**.skip(1)**

**.findFirst();**

**```**

**78. \*\*Find the kth smallest element in a list.\*\***

**```java**

**List<Integer> numbers = Arrays.asList(5, 9, 11, 2, 8, 21, 1);**

**int k = 3; // Find the 3rd smallest**

**Optional<Integer> kthSmallest = numbers.stream()**

**.sorted()**

**.skip(k - 1)**

**.findFirst();**

**```**

**79. \*\*Check if a list contains only unique elements.\*\***

**```java**

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

**boolean hasOnlyUniqueElements = numbers.stream().distinct().count() == numbers.size();**

**```**

**80. \*\*Find common elements between two lists.\*\***

**```java**

**List<Integer> list1 = Arrays.asList(1, 2, 3, 4, 5);**

**List<Integer> list2 = Arrays.asList(4, 5, 6, 7, 8);**

**List<Integer> common = list1.stream()**

**.filter(list2::contains)**

**.collect(Collectors.toList());**

**```**

**### Error Handling and Optional**

**81. \*\*Safely convert a string to an integer using Optional.\*\***

**```java**

**String str = "123";**

**Optional<Integer> result = Optional.ofNullable(str)**

**.flatMap(s -> {**

**try {**

**return Optional.of(Integer.parseInt(s));**

**} catch (NumberFormatException e) {**

**return Optional.empty();**

**}**

**});**

**```**

**82. \*\*Handle null values in a stream using Optional.\*\***

**```java**

**List<String> list = Arrays.asList("a", null, "b", null, "c");**

**List<String> nonNullUppercase = list.stream()**

**.map(Optional::ofNullable)**

**.flatMap(Optional::stream) // Java 9+**

**.map(String::toUpperCase)**

**.collect(Collectors.toList());**

**```**

**83. \*\*Find the first non-null element in a list.\*\***

**```java**

**List<String> list = Arrays.asList(null, null, "first", "second", null);**

**Optional<String> firstNonNull = list.stream()**

**.filter(Objects::nonNull)**

**.findFirst();**

**```**

**84. \*\*Provide a default value if a stream is empty.\*\***

**```java**

**List<String> emptyList = Collections.emptyList();**

**String result = emptyList.stream()**

**.findFirst()**

**.orElse("Default Value");**

**```**

**85. \*\*Throw a custom exception if a stream is empty.\*\***

**```java**

**List<String> emptyList = Collections.emptyList();**

**String result = emptyList.stream()**

**.findFirst()**

**.orElseThrow(() -> new RuntimeException("Stream is empty"));**

**```**

**86. \*\*Map values in a stream and handle potential exceptions.\*\***

**```java**

**List<String> list = Arrays.asList("10", "20", "thirty", "40");**

**List<Integer> numbers = list.stream()**

**.map(s -> {**

**try {**

**return Optional.of(Integer.parseInt(s));**

**} catch (NumberFormatException e) {**

**return Optional.<Integer>empty();**

**}**

**})**

**.flatMap(Optional::stream) // Java 9+**

**.collect(Collectors.toList());**

**```**

**87. \*\*Use Optional to handle potentially missing values in a stream.\*\***

**```java**

**Map<String, String> map = Map.of("a", "1", "b", "2", "c", "3");**

**List<String> keys = Arrays.asList("a", "d", "c", "e");**

**List<Integer> values = keys.stream()**

**.map(map::get)**

**.map(Optional::ofNullable)**

**.flatMap(Optional::stream)**

**.map(Integer::parseInt)**

**.collect(Collectors.toList());**

**```**

**88. \*\*Chain multiple operations with Optional to handle potential errors.\*\***

**```java**

**String str = "123";**

**Optional<Double> result = Optional.ofNullable(str)**

**.flatMap(s -> {**

**try {**

**return Optional.of(Integer.parseInt(s));**

**} catch (NumberFormatException e) {**

**return Optional.empty();**

**}**

**})**

**.map(n -> n \* 2.5);**

**```**

**89. \*\*Use Optional to provide different behaviors for empty and non-empty streams.\*\***

**```java**

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

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

**max.ifPresentOrElse(**

**m -> System.out.println("Max value: " + m),**

**() -> System.out.println("The list is empty")**

**);**

**```**

**90. \*\*Combine multiple Optional values from a stream.\*\***

**```java**

**List<Optional<String>> optionals = Arrays.asList(**

**Optional.of("a"),**

**Optional.empty(),**

**Optional.of("b"),**

**Optional.empty(),**

**Optional.of("c")**

**);**

**List<String> result = optionals.stream()**

**.flatMap(Optional::stream) // Java 9+**

**.collect(Collectors.toList());**

**```**

**### Debugging and Peeking**

**91. \*\*Use peek() to debug a stream pipeline without affecting the result.\*\***

**```java**

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

**List<Integer> result = numbers.stream()**

**.peek(n -> System.out.println("Before filter: " + n))**

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

**.peek(n -> System.out.println("After filter: " + n))**

**.map(n -> n \* 2)**

**.peek(n -> System.out.println("After map: " + n))**

**.collect(Collectors.toList());**

**```**

**92. \*\*Count elements at different stages of a stream pipeline.\*\***

**```java**

**List<String> words = Arrays.asList("java", "stream", "api", "lambda", "functional");**

**AtomicInteger beforeFilter = new AtomicInteger();**

**AtomicInteger afterFilter = new AtomicInteger();**

**List<String> result = words.stream()**

**.peek(s -> beforeFilter.incrementAndGet())**

**.filter(s -> s.length() > 4)**

**.peek(s -> afterFilter.incrementAndGet())**

**.collect(Collectors.toList());**

**System.out.println("Before filter: " + beforeFilter.get());**

**System.out.println("After filter: " + afterFilter.get());**

**```**

**93. \*\*Use peek() to log intermediate results in a stream pipeline.\*\***

**```java**

**List<String> words = Arrays.asList("java", "stream", "api", "lambda");**

**List<String> result = words.stream()**

**.filter(s -> {**

**boolean condition = s.length() > 3;**

**System.out.println("Filtering: " + s + ", condition: " + condition);**

**return condition;**

**})**

**.map(String::toUpperCase)**

**.peek(s -> System.out.println("After map: " + s))**

**.collect(Collectors.toList());**

**```**

**94. \*\*Debug a stream pipeline by examining the state at each step.\*\***

**```java**

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

**List<Integer> result = numbers.stream()**

**.peek(n -> System.out.println("Original: " + n))**

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

**.peek(n -> System.out.println("Filtered (even): " + n))**

**.map(n -> n \* n)**

**.peek(n -> System.out.println("Mapped (squared): " + n))**

**.collect(Collectors.toList());**

**```**

**95. \*\*Use peek() to verify the behavior of a stream pipeline.\*\***

**```java**

**List<String> words = Arrays.asList("java", "stream", "api", "lambda");**

**Map<Integer, List<String>> groupedByLength = words.stream()**

**.peek(s -> System.out.println("Processing: " + s))**

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

**groupedByLength.forEach((length, list) ->**

**System.out.println("Length " + length + ": " + list));**

**```**

**96. \*\*Identify performance bottlenecks in a stream pipeline.\*\***

**```java**

**List<Integer> numbers = IntStream.rangeClosed(1, 1000000).boxed().collect(Collectors.toList());**

**long startTime = System.currentTimeMillis();**

**List<Integer> result = numbers.stream()**

**.peek(n -> {**

**if (n % 100000 == 0) {**

**System.out.println("Processing: " + n + ", time: " +**

**(System.currentTimeMillis() - startTime) + "ms");**

**}**

**})**

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

**.map(n -> n \* 2)**

**.collect(Collectors.toList());**

**System.out.println("Total time: " + (System.currentTimeMillis() - startTime) + "ms");**

**```**

**97. \*\*Use peek() to validate transformations in a stream pipeline.\*\***

**```java**

**List<String> words = Arrays.asList("java", "stream", "api", "lambda");**

**List<String> result = words.stream()**

**.peek(s -> System.out.println("Original: " + s))**

**.map(String::toUpperCase)**

**.peek(s -> System.out.println("Uppercase: " + s))**

**.filter(s -> s.length() > 3)**

**.peek(s -> System.out.println("Filtered: " + s))**

**.collect(Collectors.toList());**

**```**

**98. \*\*Debug a complex stream pipeline with multiple operations.\*\***

**```java**

**List<String> words = Arrays.asList("java", "stream", "api", "lambda", "functional");**

**Map<Boolean, List<String>> result = words.stream()**

**.peek(s -> System.out.println("Original: " + s))**

**.filter(s -> s.length() > 3)**

**.peek(s -> System.out.println("After length filter: " + s))**

**.map(String::toUpperCase)**

**.peek(s -> System.out.println("After uppercase: " + s))**

**.collect(Collectors.partitioningBy(s -> s.contains("A")));**

**System.out.println("Contains 'A': " + result.get(true));**

**System.out.println("Does not contain 'A': " + result.get(false));**

**```**

**99. \*\*Use peek() to count elements by category in a stream pipeline.\*\***

**```java**

**List<String> words = Arrays.asList("java", "stream", "api", "lambda", "functional");**

**Map<String, AtomicInteger> counts = new HashMap<>();**

**counts.put("short", new AtomicInteger());**

**counts.put("medium", new AtomicInteger());**

**counts.put("long", new AtomicInteger());**

**List<String> result = words.stream()**

**.peek(s -> {**

**if (s.length() <= 3) counts.get("short").incrementAndGet();**

**else if (s.length() <= 6) counts.get("medium").incrementAndGet();**

**else counts.get("long").incrementAndGet();**

**})**

**.collect(Collectors.toList());**

**System.out.println("Short words: " + counts.get("short"));**

**System.out.println("Medium words: " + counts.get("medium"));**

**System.out.println("Long words: " + counts.get("long"));**

**```**

**100. \*\*Use peek() to track the execution order of a stream pipeline.\*\***

**```java**

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

**List<Integer> result = numbers.stream()**

**.peek(n -> System.out.println("Before filter: " + n))**

**.filter(n -> {**

**System.out.println("Filtering: " + n);**

**return n % 2 == 0;**

**})**

**.peek(n -> System.out.println("After filter: " + n))**

**.map(n -> {**

**System.out.println("Mapping: " + n);**

**return n \* 2;**

**})**

**.peek(n -> System.out.println("After map: " + n))**

**.collect(Collectors.toList());**

**```**

**## Intermediate Level Problems (100)**

**New Features in Java**

**How is Java Marching Forward?**[**#**](https://interview.in28minutes.com/interview-guides/java/new-features-in-java/#how-is-java-marching-forward%3F)

**📌 Breaking Myth**

* **Myth: "Java is old and unpopular"**
* **Truth: Java remains one of the most popular programming languages worldwide!**
  + **Java is one of the popular choices for building:**
    - **Modern cloud applications (Spring Boot, Quarkus, Micronaut)**
  + **IMHO Java is evolving faster than ever, staying modern, powerful, and future-proof!**

**📌 Continuous Innovation in Java Eco System**

* **Java 8 – Embraced functional programming with Lambda expressions and Streams.**
* **Java 9 – Introduced modularization to the Java Platform - enabling faster startup and less memory footprint for Java applications**
* **Java 10 to NOW – Started the six-month release cycle for faster updates and quick feedback.**
* **Java 11 to NOW – Reduced code verbosity**
  + **Switch Expressions – More concise conditional logic.**
  + **Text Blocks – Simplifies multi-line strings.**
  + **Records – Reduces boilerplate for data objects.**
  + **Pattern Matching – Enhances code readability and conciseness.**
* **Java 21 to NOW:**
  + **Project Loom – Virtual threads for high-performance concurrency.**
  + **Cloud & Container-Ready – Optimized for modern cloud-native applications.**

**What are the primary goals behind recent Java releases?**[**#**](https://interview.in28minutes.com/interview-guides/java/new-features-in-java/#what-are-the-primary-goals-behind-recent-java-releases%3F)

* **Make it concise – Reduce boilerplate and improve readability.**
  + **Records – No need for boilerplate code in data classes.**

**record Person(String name, int age) {}**

* + **Switch Expressions – More readable and concise switch statements.**
  + **String result = switch (day) {**
  + **case MONDAY, FRIDAY -> "Weekend is near!";**
  + **default -> "Regular day.";**

**};**

* **Cloud and container support – Improve compatibility with cloud-native environments.**
  + **Optimized JDK Container Images – Optimized JDK images for containers with small memory footprint and fast start up times.**
  + **GraalVM – Enables efficient native image compilation for lightweight, fast-running applications.**
* **Improve concurrency – Introduce Virtual Threads for scalable parallelism.**
  + **Virtual Threads – Handles millions of lightweight threads efficiently.**
  + **Thread.*startVirtualThread*(**
  + **() -> System.out.*println*(**

**"Hello from a virtual thread!"));**

* **Developer productivity – Provide better debugging tools and runtime efficiency.**
  + **JShell – Interactive REPL for quick testing.**

**jshell> System.out.*println*("Hello, Java!");**

* + **Simpler Main Without Class - Write just the method, skip the boilerplate**
  + ***//HelloWorld.java***
  + **void *main*() {**
  + **System.out.*println*("Hello, World!");**

**}**

* + **Easy Launch without Compilation - Run source files directly**

**$ java HelloWorld.java**

* **Reduce startup time – Optimize performance for faster application launch.**
* **Better memory management – Optimize Garbage Collection (GC) for lower latency.**
* **Backward compatibility – Ensure smooth migration from older versions.**

**What are some major Java versions and their most important features?**[**#**](https://interview.in28minutes.com/interview-guides/java/new-features-in-java/#what-are-some-major-java-versions-and-their-most-important-features%3F)

**📌 J2SE 5.0**

* **Enhanced For Loop – Simplifies iteration over collections and arrays.**
* **for (int num : numbers) {**
* ***//Do something***

**}**

* **Generics – Enables type safety in collections and methods.**

**public class HashMap<K,V> extends AbstractMap<K,V>**

* **Enums – Provides a type-safe way to define constants.**
* **enum Season {**
* **WINTER, SPRING, SUMMER, FALL**

**}**

* **Autoboxing – Automatically converts primitives to wrapper classes.**
* **Integer integer = 5;**

**int i = integer;**

**📌 Java SE 7**

* **Try-With-Resources: Automates resource management in try-catch blocks.**
* **try (BufferedReader br =**
* **new BufferedReader(**

**new FileReader("file.txt"))) {**

* **Strings in Switch: Allows using String values in switch statements.**
* **switch (day) {**
* **case "Monday": *//Do Something;***
* **case "Friday": *//Do Something;***
* **case "Sunday": *//Do Something;***
* **default: *//Do Something;***

**}**

**📌 Java SE 8**

* **Lambda Expressions – Allows writing concise, functional-style code.**
* **public static Predicate<Course>**
* ***createPredicate*(int cutoff) {**
* **return course ->**
* **course.*getReviewScore*() > cutoff;**

**}**

* **Streams API – Provides a powerful way to process collections.**
* **List<String> transformedNumbers = numbers.*stream*()**
* **.*map*(x -> x \* x) *// Step 1: Square each number***
* **.*map*(x -> x + 10) *// Step 2: Add 10***
* **.*map*(x -> "Value: " + x) *// Step 3: Convert***

**.*toList*(); *// Collect the result***

* **Functional Programming – Introduces higher-order functions and immutability.**
* **courses.*stream*()**
* **.*sorted*((c1, c2) -> c1.*length*() - c2.*length*())**

**.*forEach*(System.out::*println*);**

**📌 Java SE 9**

* **Modularization – Splits JDK into modules for better scalability.**
* **JShell (REPL) – Interactive command-line tool for testing Java code.**

**jshell> System.out.*println*("Hello, Java!");**

**📌 Java SE 10**

* **Local Variable Type Inference (var) – Allows type inference for local variables.**
* ***// ArrayList<String> numbers = new ArrayList<>();***
* ***// No need to repeat ArrayList<String>***
* **var numbers = new ArrayList<String>();**
* ***// Inferred as HashMap<Integer, String>***

**var map = new HashMap<Integer, String>();**

**📌 Java SE 14**

* **Switch Expressions – Enables switch to return values, reducing boilerplate code.**
* **String result = switch (day) {**
* **case MONDAY, FRIDAY -> "Weekend is near!";**
* **default -> "Regular day.";**

**};**

**📌 Java SE 15**

* **Text Blocks – Simplifies working with multi-line strings.**
* **String json = """**
* **{**
* **"name": "John",**
* **"age": 30**
* **}**
* **""";**

***//json ==> "{\n \"name\": \"John\",\n \"age\": 30\n}\n"***

**📌 Java SE 16**

* **Pattern Matching for instanceof - Simplify code making use of instanceof**
* **public void *process*(Object obj) {**
* **if (obj instanceof String s) {**
* ***// No explicit cast needed***
* **System.out.*println*("Message: " + s);**
* **}**

**}**

* **Record Classes – Provides a compact syntax for immutable data objects.**
* **public record Person(**
* **String name, String email,**

**String phoneNumber) {}**

**📌 Java SE 17**

* **Sealed Classes - Controlled inheritance of your class hierarchies.**
* **sealed class Vehicle permits Car, Truck, Bike {}**
* ***// No further subclassing***
* **final class Car extends Vehicle {}**
* ***// Can be extended freely***
* **non-sealed class Truck extends Vehicle {}**
* **sealed class Bike extends Vehicle**
* **permits ElectricBike {}**
* ***// No further subclassing***

**final class ElectricBike extends Bike {}**

**📌 Java SE 21**

* **Virtual Threads – Improves concurrency with lightweight threads.**
* **Thread.*startVirtualThread*(**
* **() -> System.out.*println*(**

**"Hello from a virtual thread!"));**

* **Sequenced Collections – Adds operations for ordered collections.**
* **interface SequencedCollection<E> extends Collection<E> {**
* **void *addFirst*(E element);**
* **void *addLast*(E element);**
* **E *getFirst*();**
* **E *getLast*();**
* **E *removeFirst*();**
* **E *removeLast*();**
* **SequencedCollection<E> *reversed*();**

**}**

* **Record Patterns - Supports deconstruction, including nested patterns**
* **record Transaction(**
* **String sender, String receiver, double amount) {}**
* **public void *processTransaction*(Object obj) {**
* **if (obj instanceof Transaction(**
* **String sender, String receiver, double amount)) {**
* **System.out.*println*("Processing transaction: "**
* **+ sender + " -> " + receiver + " : ₹" + amount);**
* **}**

**}**

* **Pattern Matching for Switch - More expressive and type-safe switch statements**
* **interface CustomerMessage{}**
* **record TextMessage(String text) implements CustomerMessage {}**
* **record Ticket(int ticketNumber) implements CustomerMessage {}**
* **CustomerMessage message = new Ticket(1050);**
* **String response = switch (message) {**
* **case TextMessage(String text)**
* **-> "Processing text message: "**
* **+ text;**
* **case Ticket(int ticketNumber)**
* **-> "Processing support ticket: #"**
* **+ ticketNumber;**
* **};**

**System.out.*println*(response);**

**📌 Java SE 22**

* **Unnamed Variables & Patterns – Allows ignoring unused variables in code.**
* **switch (obj) { *//I will not use the value!***
* **case String \_ -> System.out.*println*("String");**
* **case Integer \_ -> System.out.*println*("Integer");**
* **default -> System.out.*println*("Unknown type.");**

**}**

**📌 Java SE 23**

* **Markdown Documentation Comments – Enhances Javadoc with Markdown support.**
* ***/// This is the \*\*traditional\*\****
* ***/// \*Hello World!\* program.***
* ***/// This is the <strong>traditional</strong>***

***/// <em>Hello World!</em> program.***

**📌 Java SE 24**

* **Stream Gatherers – Introduces new ways to aggregate stream data.**
* ***//A new way to group,***
* ***//transform, and aggregate stream elements***
* **List<List<Integer>> windows = numbers.*stream*()**
* **.*gather*(Gatherers.*windowFixedSize*(3))**
* **.*toList*();**
* ***//[1, 2, 3]***
* ***//[4, 5, 6]***
* ***//[7, 8, 9]***

***//[10]***

**📌 Ongoing Improvements**

* **Performance Optimizations – Enhances start up speed, memory footprint and efficiency of Java applications.**
* **Garbage Collection Enhancements – Improves memory management and GC algorithms.**
* **API and Syntactic Improvements – Continuously refines Java syntax and library functions.**

**Exception Handling in Java**

**Why is Exception Handling Important?**[**#**](https://interview.in28minutes.com/interview-guides/java/exception-handling-in-java/#why-is-exception-handling-important%3F)

**Exceptions Happen Regularly**

* **Why? Complex code, changing environments, and human errors**
  + **Defects Are Inevitable – Even great developers miss edge cases**
  + **Runtime Surprises – Files missing, servers down, data corrupted**
  + **External Systems Fail – APIs, databases, and networks aren’t always reliable**
  + **Environment Changes – OS updates, memory limits, config issues, etc.**
* **Moral of the Story? Expect exceptions. Handle them gracefully.**

**Need to Handle Them Properly**

* **What? Handling exceptions isn't just about catching them**
  + **It's about solving them gracefully without scaring your end users!**
* **1. User Communication**
  + **Show friendly, non-technical error messages**
  + **Avoid app crashes or ugly stack traces**
  + **Display a unique error ID the user can report**
  + **Guide users on how to contact support**
* **2. Debugging Support**
  + **Log the full stack trace and context**
  + **Include the same error ID shown to the user**
  + **Capture input data, user actions, or environment details**
  + **Helps support/dev teams trace and fix the issue fast**

**Which Design Pattern Powers Java's Exception Flow?**[**#**](https://interview.in28minutes.com/interview-guides/java/exception-handling-in-java/#which-design-pattern-powers-java's-exception-flow%3F)

* **Chain of Responsibility: A behavioral pattern where a request moves through a chain of handlers**
* **Why?: Decouples sender and receiver — each handler decides to process or pass along**
* **How It Works:**
  + **Exception travels up the call stack method by method**
  + **Each method can catch or pass the exception higher**
  + **Stops when an appropriate handler is found**
* **Real-World Analogy:**
  + **Like a leave request going from employee → manager → HR**
  + **Each can approve or escalate the request**
* **public static void *main*(String[] args) {**
* ***method1*();**
* **}**
* **private static void *method1*() {**
* ***method2*();**
* **}**
* **private static void *method2*() {**
* **String str = null;**
* **str.*toString*(); *// Throws NullPointerException***
* **}**
* ***//Exception in thread "main"***
* ***//java.lang.NullPointerException at***
* ***//method2(ExceptionHandlingExample1.java:15)***
* ***//at method1(ExceptionHandlingExample1.java:10)***

***//at main(ExceptionHandlingExample1.java:6)***

**Can You Walk Through a Practical Example of Java Exception Handling?**[**#**](https://interview.in28minutes.com/interview-guides/java/exception-handling-in-java/#can-you-walk-through-a-practical-example-of-java-exception-handling%3F)

* **PaymentApp Tries to make a payment of $600 (more than balance) using PaymentService**
  + **Custom exception InsufficientFundsException extends Exception**
  + **PaymentService throws InsufficientFundsException if not enough money**
  + **PaymentApp Uses try-catch-finally to handle exceptions cleanly**
    - **Logs and prints appropriate error or success messages**
    - **Always prints “Transaction attempt completed” using finally**

**public class PaymentApp {**

**public static void *main*(String[] args) {**

**PaymentService paymentService = new PaymentService();**

**try {**

**paymentService.*makePayment*(600);**

**} catch (InsufficientFundsException e) {**

**System.err.*println*(**

**"Error: " + e.*getMessage*());**

**} catch (IllegalArgumentException e) {**

**System.err.*println*(**

**"Invalid Input: " + e.*getMessage*());**

**} catch (Exception e) {**

**System.err.*println*(**

**"Unexpected error: " + e.*getMessage*());**

**} finally {**

**System.out.*println*(**

**"Transaction attempt completed.");**

**}**

**}**

**}**

**class PaymentService {**

**private static final Logger logger**

**= Logger.*getLogger*(PaymentService.class.*getName*());**

**private double balance = 500.0; *// Initial balance***

**public void *makePayment*(double amount)**

**throws InsufficientFundsException {**

**logger.*info*("Processing payment of $" + amount);**

**if (amount <= 0) {**

**throw new IllegalArgumentException**

**("Payment amount must be greater than zero.");**

**}**

**if (amount > balance) {**

**throw new InsufficientFundsException**

**("Available balance: $"**

**+ balance);**

**}**

**balance -= amount;**

**System.out.println**

**("Payment of $" + amount**

**+ " successful. Remaining balance: $"**

**+ balance);**

**}**

**}**

**class InsufficientFundsException extends Exception {**

**public InsufficientFundsException(String message) {**

**super(message);**

**}**

**}**

**1. Try-Catch Block (Handling Exceptions Gracefully)**

* **What? Used to catch and handle exceptions to prevent program crashes.**
* **Where? In main(), different catch blocks handle specific exceptions.**
* **try {**
* **paymentService.*makePayment*(600);**
* **} catch (InsufficientFundsException e) {**
* **System.err.*println*("Error: "**
* **+ e.*getMessage*());**
* **} catch (IllegalArgumentException e) {**
* **System.err.*println*("Invalid Input: "**
* **+ e.*getMessage*());**
* **} catch (Exception e) {**
* **System.err.*println*("Unexpected error: "**
* **+ e.*getMessage*());**

**}**

* **Key Points:**
  + **Multiple catch Blocks: Each block handles a different exception type.**
  + **Order of Exceptions -> Specific to General: InsufficientFundsException (custom exception) comes before Exception (general).**

**2. Custom Exception (InsufficientFundsException)**

* **What? A user-defined exception for handling specific errors.**
* **Why? Provides meaningful error messages instead of generic exceptions.**
* **Extends Exception → Makes it a checked exception.**
* **class InsufficientFundsException extends Exception {**
* **public InsufficientFundsException(String message) {**
* **super(message);**
* **}**

**}**

**3. Throw and Throws (Propagating Exceptions)**

* **What? The throw statement generates exceptions, and throws declares them in the method signature.**
* **Where? Inside PaymentService.makePayment()**
* **public void *makePayment*(double amount)**
* **throws InsufficientFundsException {**
* **if (amount > balance) {**
* **throw new InsufficientFundsException**
* **("Available balance: $" + balance);**
* **}**

**}**

* **Key Points: throw vs throws**
  + **throw new InsufficientFundsException(...) → Generates the exception.**
  + **throws InsufficientFundsException → Informs the caller (main()) to handle it.**

**4. Finally Block (Ensuring Cleanup Code Executes)**

* **What? The finally block (almost) always executes whether an exception occurs or not.**
* **Why? Used for cleanup tasks like closing connections or logging actions.**
* **finally {**
* ***// Almost Always executes***
* ***// Clean up!***
* **System.out.*println*("Transaction attempt completed.");**

**}**

* **Key Points:**
  + **Runs even if an exception occurs.**
  + **Used for releasing resources, or performing essential operations.**

**5. Catch-All Exception Handling (Unexpected Errors)**

* **What? The last catch block catches any unexpected exceptions.**
* **Why? Ensures the program doesn’t crash due to an unhandled error.**
* **catch (Exception e) {**
* **System.err.*println*(**
* **"Unexpected error: " + e.*getMessage*());**

**}**

* **Key Points:**
  + **Catches any other runtime exceptions that are not explicitly handled.**
  + **Helps in debugging unexpected errors.**

**What Happens When You Swallow Exceptions — And Why Is It Bad?**[**#**](https://interview.in28minutes.com/interview-guides/java/exception-handling-in-java/#what-happens-when-you-swallow-exceptions-%E2%80%94-and-why-is-it-bad%3F)

* **What? An exception is caught but silently ignored**
* **Why It's Bad? Looks like everything is fine... but it's not**
* **No Logging = No Clues – Developer's can't trace what went wrong**
* **Program Continues – But state might be broken or inconsistent**
* **Result? Bugs hide, grow, and jump out later in weirder places**
* **❌ Example of Exception Swallowing (Bad Practice)**
* **try {**
* **int[] numbers = {1, 2};**
* ***// ❌ ArrayIndexOutOfBoundsException***
* **System.out.*println*(numbers[3]);**
* **} catch (Exception e) {**
* ***// ❌ Exception is caught but ignored***
* **}**

**System.out.*println*("Program continues...");**

* **Why Is This a Problem?**
  + **No Clue Something Broke – Developer can’t fix what they can’t see**
  + **Silent Failures = Sneaky Bugs – App keeps running in a bad state**
  + **Delayed Explosions – Errors surface much later in unrelated places**
  + **Wasted Time – Debugging without logs is like finding a needle in a dark room**
  + **✅ Always log, handle, or rethrow exceptions**
    - **Your future self (and your team) will thank you!**

**In What Scenarios is Code in finally Not Executed?**[**#**](https://interview.in28minutes.com/interview-guides/java/exception-handling-in-java/#in-what-scenarios-is-code-in-finally-not-executed%3F)

* **❌ Example Without finally (Problem Case)**
* **private static void *method2*() {**
* **Connection connection = new Connection();**
* **connection.open();**
* **try {**
* ***// LOGIC***
* **String str = null;**
* **str.*toString*();**
* **} catch (Exception e) {**
* **System.out.*println*("Exception Handled - Method 2");**
* **}**
* ***// Connection is never closed if exception occurs ❌***

**}**

* + **Problem: If an exception occurs, opened connections may remain unclosed.**
* **✅ Example With finally (Proper Resource Cleanup)**
* **private static void *method2*() {**
* **Connection connection = new Connection();**
* **connection.open();**
* **try {**
* **String str = null;**
* **str.*toString*();**
* **} catch (Exception e) {**
* **System.out.*println*("Exception Handled - Method 2");**
* **} finally {**
* ***// Ensures connection is always closed ✅***
* **connection.*close*();**
* **}**

**}**

* **finally is NOT executed in these two cases**
  + **Exception Inside finally – If something goes wrong *inside* finally, it may skip the rest**
  + **JVM Shutdown – If the JVM dies (e.g., System.exit() or crash), finally won’t get a chance**

**What Combinations of try, catch, and finally Are Allowed?**[**#**](https://interview.in28minutes.com/interview-guides/java/exception-handling-in-java/#what-combinations-of-try%2C-catch%2C-and-finally-are-allowed%3F)

* **try + catch – ✅ Allowed**
  + **Handles exceptions, no need for finally**
* **try {**
* ***riskyThing*();**
* **} catch (Exception e) {**
* ***handleIt*(e);**

**}**

* **try + finally – ✅ Allowed**
  + **Cleanup logic runs, even without handling exceptions**
* **try {**
* ***riskyThing*();**
* **} finally {**
* ***cleanup*();**

**}**

* **try + catch + finally – ✅ Allowed**
  + **The full combo: handle errors + cleanup**
* **try {**
* ***riskyThing*();**
* **} catch (Exception e) {**
* ***handleIt*(e);**
* **} finally {**
* ***cleanup*();**

**}**

* **try Alone – ❌ Not allowed**
  + **Compilation error: must have at least a catch or finally**
* **try {**
* ***riskyThing*();**
* **}**

***// ❌ Error: Missing catch or finally***

**What Is the Class Hierarchy Behind Java Exceptions?**[**#**](https://interview.in28minutes.com/interview-guides/java/exception-handling-in-java/#what-is-the-class-hierarchy-behind-java-exceptions%3F)

| **Class Name** | **Category (Parent)** | **Checked?** | **Description** |
| --- | --- | --- | --- |
| **Throwable** | **—** | **—** | **Root of all exceptions and errors** |
| **Error** | **Throwable** | **No** | **Serious system-level issues** |
| **OutOfMemoryError** | **VirtualMachineError (Error)** | **No** | **JVM ran out of memory** |
| **StackOverflowError** | **VirtualMachineError (Error)** | **No** | **Stack memory exhausted (e.g., recursion)** |
| **Exception** | **Throwable** | **Yes** | **Base for all application exceptions** |
| **IOException** | **Exception** | **Yes** | **Input/output failure** |
| **SQLException** | **Exception** | **Yes** | **Database access error** |
| **Class NotFound Exception** | **Exception** | **Yes** | **Class loading failed** |
| **RuntimeException** | **Exception** | **No** | **Base for unchecked exceptions** |
| **NullPointerException** | **RuntimeException** | **No** | **Accessing null reference** |
| **ArrayIndex OutOfBounds Exception** | **RuntimeException** | **No** | **Invalid array index** |
| **Illegal Argument Exception** | **RuntimeException** | **No** | **Invalid method argument** |
| **Arithmetic Exception** | **RuntimeException** | **No** | **Math error (e.g., divide by zero)** |

**Error vs Exception**

| **Error** | **Exception** |
| --- | --- |
| **Used for critical system failures.** | **Used for recoverable issues.** |
| **Cannot be handled by the programmer.** | **Can be handled by the programmer.** |
| **Examples: StackOverflowError, OutOfMemoryError.** | **Examples: IOException, NullPointerException.** |

**How Do Checked and Unchecked Exceptions Differ in Java?**[**#**](https://interview.in28minutes.com/interview-guides/java/exception-handling-in-java/#how-do-checked-and-unchecked-exceptions-differ-in-java%3F)

**1. What Are Checked Exceptions?**

* **What? Subclasses of Exception excluding RuntimeException and its subclasses**
* **When? Compiler checks them at compile time**
* **Must Handle? Yes — use try-catch or declare with throws**
* **Why? These represent recoverable problems (like file not found)**
* **Common Checked Exceptions:**
  + **IOException – File or network access error**
  + **ClassNotFoundException – Class loading issue**
* **Compiler Says: "Handle it, or tell me you're throwing it!"**
* **import java.io.\*;**
* **class FileReaderExample {**
* **public static void *readFile*()**
* **throws IOException {**
* ***// Method must declare or handle IOException***
* **FileReader file = new FileReader("file.txt");**
* **BufferedReader br = new BufferedReader(file);**
* **System.out.*println*(br.*readLine*());**
* **br.*close*();**
* **}**
* **public static void *main*(String[] args) {**
* **try {**
* ***readFile*(); *// Handling the checked exception***
* **} catch (IOException e) {**
* **System.out.*println*("Error: " + e.*getMessage*());**
* **}**
* **}**

**}**

**2. What are Unchecked (Runtime) Exceptions?**

* **What? Subclasses of RuntimeException**
* **When? Compiler doesn’t force you to handle them**
* **Checked? Nope — they're unchecked at compile-time**
* **Why? These represent programming bugs or logic errors**
* **Common Unchecked Exceptions:**
  + **NullPointerException – Operation on a null variable**
  + **ArithmeticException – Divide by zero exception**
  + **ArrayIndexOutOfBoundsException – Bad array index**
  + **IllegalArgumentException – Invalid method argument**
* **class UncheckedExample {**
* **public static void *main*(String[] args) {**
* **String text = null;**
* ***// ❌ NullPointerException***
* **System.out.*println*(text.*length*());**
* **}**

**}**

**How to Choose Between Checked and Unchecked Exceptions?**

* **Use Checked – When the caller can and should handle the issue**
  + **Example: Missing file, invalid input, network timeout**
  + **Forces the developer to acknowledge the risk**
* **Use Unchecked – When the issue is a bug or misuse of API**
  + **Example: NullPointerException, IndexOutOfBoundsException**
  + **Caller usually can’t recover**

**What Are Chained Exceptions?**[**#**](https://interview.in28minutes.com/interview-guides/java/exception-handling-in-java/#what-are-chained-exceptions%3F)

* **What? Chained exceptions allow you to link one exception to another**
* **Why? To show the original cause of an exception**
* **How? Use initCause() method**
* **Helps With? Debugging and tracing multi-layered errors**
* **Common Use Case:**
  + **You catch a low-level exception and wrap it inside a higher-level one**
  + **This way, you don’t lose the original stack trace**
* **Example:**
* **public class ChainedExceptionDemo {**
* **public static void *main*(String[] args) {**
* **try {**
* ***validate*(null);**
* **} catch (Exception e) {**
* **System.out.*println*("Exception: " + e);**
* **System.out.*println*("Caused by: "**
* **+ e.*getCause*());**
* **}**
* **}**
* **static void *validate*(String name) throws Exception {**
* **try {**
* **if (name == null) {**
* **throw new NullPointerException(**
* **"Username is null");**
* **}**
* **} catch (NullPointerException e) {**
* **Exception wrapped**
* **= new Exception(**
* **"User validation failed");**
* ***// ✅ Linking original cause***
* **wrapped.*initCause*(e);**
* **throw wrapped;**
* **}**
* **}**

**}**

**What Are the Best Practices for Handling Exceptions in Java?**[**#**](https://interview.in28minutes.com/interview-guides/java/exception-handling-in-java/#what-are-the-best-practices-for-handling-exceptions-in-java%3F)

* **1. Never Hide Exceptions – Always log them, even if you re-throw or recover**
  + **Why? Silent failures are hard to debug**
  + **Bad Example:**
  + **try {**
  + ***riskyOperation*();**
  + **} catch (Exception e) {**
  + ***// ❌ Nothing logged, issue disappears***

**}**

* + **Good Example:**
  + **try {**
  + ***riskyOperation*();**
  + **} catch (Exception e) {**
  + ***// ✅ Log the exception***
  + **logger.*error*("Something went wrong", e);**
  + **throw e; *// Optional: rethrow if needed***

**}**

* **2. User-Friendly Messages – Show clear, non-technical messages to users**
  + **Why? Users shouldn't see scary stack traces**
  + **Example (Spring Controller):**
  + **@GetMapping("/item")**
  + **public ResponseEntity<?> *getItem*() {**
  + **try {**
  + **return ResponseEntity.*ok*(service.*getItem*());**
  + **} catch (ItemNotFoundException e) {**
  + **return ResponseEntity**
  + **.*status*(404)**
  + **.*body*("Item not found." +**
  + **" Please check the ID.");**
  + **}**

**}**

* **3. Provide Debugging Info – Log stack traces and context for developers**
  + **Why? Helps devs trace root cause**
  + **Example:**
  + **try {**
  + ***processData*();**
  + **} catch (Exception e) {**
  + **logger.*error*(**
  + **"Failed to process data for user: "**
  + **+ userId, e);**

**}**

* **4. Global Exception Handling – Use centralized mechanisms (like @ControllerAdvice in Spring)**
  + **Why? Avoids duplicate try-catch blocks everywhere**
  + **Example:**
  + **@RestControllerAdvice**
  + **public class GlobalExceptionHandler {**
  + **@ExceptionHandler(IllegalArgumentException.class)**
  + **public ResponseEntity<String> *handleBadInput*(**
  + **Exception e) {**
  + **return ResponseEntity**
  + **.*badRequest*()**
  + **.*body*(**
  + **"Invalid input. Please try again.");**
  + **}**

**}**

* **5. Avoid Misusing Exceptions – Don’t use exceptions for expected logic like looping or validation**
  + **Why? Bad for performance and readability**
  + **Bad Example:**
  + **while (true) {**
  + **try {**
  + **return Integer.*parseInt*(input);**
  + **} catch (NumberFormatException e) {**
  + ***// ❌ Using exception to control loop***
  + **input = *getNewInput*();**
  + **}**

**}**

* + **Good Example:**
  + ***// ✅ Validate before parsing***
  + **while (!*isNumeric*(input)) {**
  + **input = *getNewInput*();**
  + **}**

**return Integer.*parseInt*(input);**

**What Are the Newer Java Features That Help with Exception Handling?**[**#**](https://interview.in28minutes.com/interview-guides/java/exception-handling-in-java/#what-are-the-newer-java-features-that-help-with-exception-handling%3F)

**1. Try-With-Resources (Automatic Resource Management)**

* **What? A try block that automatically closes resources**
* **Why? No need for manual finally cleanup**
* **When? Use with classes that implement AutoCloseable**
* **Benefit? Less boilerplate, fewer resource leaks**
* **Common Resources:**
  + **BufferedReader, FileInputStream, Connection, Scanner**
* **try (BufferedReader br =**
* **new BufferedReader(**
* **new FileReader("file.txt"))) {**
* **System.out.*println*(br.*readLine*());**
* **} catch (IOException e) {**
* **e.*printStackTrace*();**
* **}**
* ***//Use with classes that implement AutoCloseable***
* **public interface AutoCloseable {**
* **void *close*() throws Exception;**

**}**

**2. Multi-Catch Block**

* **What? Handle multiple exception types in one catch block**
* **How? Use | to separate exception classes**
* **Why? Avoids repeating the same logic for different exceptions**
* **When? When multiple exceptions need the same handling**
* **try {**
* ***methodThatThrowsException*();**
* ***// Multi-catch block***
* **} catch (IOException | SQLException ex) {**
* **System.err.*println*(**
* **"Exception occurred: "**
* **+ ex.*getMessage*());**

**}**

**3. Optional.orElseThrow()**

* **What? Throws an exception if the value is absent**
* **Why? Avoids null checks and improves readability**
* **When? Use when a value is required and absence is exceptional**
* **Benefit? Cleaner, expressive, and avoids if (optional.isPresent())**
* **Optional<String> name = Optional.*ofNullable*(null);**
* **String result = name.*orElseThrow*(**
* **() -> new IllegalArgumentException(**

**"Value is missing"));**

**Collections in Java**

**How did the Collections Framework Evolve in Java?**[**#**](https://interview.in28minutes.com/interview-guides/java/collections-in-java/#how-did-the-collections-framework-evolve-in-java%3F)

**Early Collections**

* **Early classes for grouping objects included Dictionary, Vector, Stack, and Properties.**
* **These classes had an ad-hoc design.**
* **There was no consistent set of interfaces.**

**Collections Framework**

* **The Collections Framework was introduced in Java 1.2.**
* **It had two main goals:**
  + **Unify collection handling through standard interfaces and implementations**
  + **Improve performance, flexibility, and ease of use**

**Evolution**

* **Over time, the framework evolved with new features:**
  + **Java 5: Generics, enhanced for-each loop, concurrent collections (e.g., ConcurrentHashMap)**
  + **Java 8: Stream API, forEach, lambda expressions**
  + **Java 9: Factory methods for immutable collections (List.of(), Set.of(), Map.of())**
  + **Java 21: Sequenced collections for consistent ordering across types**

**Why are Collections needed in Java?**[**#**](https://interview.in28minutes.com/interview-guides/java/collections-in-java/#why-are-collections-needed-in-java%3F)

**Collections in Java provide a flexible and efficient way to store, retrieve, and manipulate data.**

**Limitations of Arrays:**

* **Fixed Size – Cannot grow or shrink dynamically.**
* **Manual Operations – Searching, adding, or removing elements requires extra logic**
* **Lack of Built-in Methods – No built-in sorting, searching, or advanced data structures.**

**How Collections Solve These Problems:**

* **Dynamic Resizing – Can grow or shrink as needed.**
* **Predefined Methods – Supports adding, removing, sorting, and searching elements easily.**
* **Rich Data Structures – Includes Lists, Sets, Maps, and Queues for different needs.**

**Example – Array vs. Collection**

**String[] names = new String[3];**

**names[0] = "Alice";**

**names[1] = "Bob";**

**names[2] = "Charlie";**

***// Adding a new name?***

***//❌ Requires creating a new array.***

**List<String> names = new ArrayList<>();**

**names.*add*("Alice");**

**names.*add*("Bob");**

**names.*add*("Charlie");**

**names.*add*("David"); *// ✅ No need to resize manually***

**Can you give a 10,000 feet overview of interfaces in collection framework?**[**#**](https://interview.in28minutes.com/interview-guides/java/collections-in-java/#can-you-give-a-10%2C000-feet-overview-of-interfaces-in-collection-framework%3F)

* **Collection is at the root of the hierarchy (add element, remove element, ..)**
  + **SequencedCollection provides well-defined order (add first, add last, ..)**
  + **List stores ordered elements, allows duplicates, supports index (add element e at index 3)**
  + **Set holds unique elements; no duplicates allowed**
    - **SortedSet maintains elements in sorted order**
    - **NavigableSet adds search/navigation methods**
  + **Queue is used for FIFO processing of elements**
  + **Deque is a double-ended queue (SequencedCollection)**
* **Map stores key-value pairs; keys must be unique (Does not inherit from Collection interface)**
  + **SortedMap maintains elements in sorted order**
  + **NavigableMap adds search/navigation methods**

**What are the main interfaces in the Collections Framework?**[**#**](https://interview.in28minutes.com/interview-guides/java/collections-in-java/#what-are-the-main-interfaces-in-the-collections-framework%3F)

**NOTE: This is provided as a quick reference. We will not have a video discussing this question. We will discuss all the interfaces in videos for subsequent questions**

| **Name** | **Description** | **Important Operations** | **Extends** |
| --- | --- | --- | --- |
| **Collection** | **Root interface for collections hierarchy.** | **add(e), remove(e), size(), iterator(), contains(e), ..** | **Iterable** |
| **Sequenced Collection** | **A collection that has a well-defined encounter order, that supports operations at both ends, and that is reversible.** | **Inherited + addFirst(E e), addLast(E e), getFirst(), getLast(), removeFirst(), removeLast(), reversed(), ..** | **Collection** |
| **List** | **Ordered collection allowing duplicates. Access using index (positioned).** | **Inherited + get(int index), set(int index, E element), void add(int index, E element), remove(int index), indexOf(Object element),...** | **Sequenced Collection** |
| **Set** | **Collection of unique elements (NO duplicates)** | **Inherited Methods** | **Collection** |
| **Sequenced Set** | **A collection that is both a Sequenced Collection and a Set** | **`Inherited Methods** | **Sequenced Collection, Set** |
| **SortedSet** | **Set maintaining elements in sorted order** | **first(), last(), subSet(), headSet(), tailSet()** | **Set, Sequenced Set** |
| **NavigableSet** | **Sorted set with navigation methods (find closest match elements)** | **lower(e), higher(e), floor(e), ceiling(e)** | **SortedSet** |
| **Queue** | **Collection that follows FIFO order.** | **offer(e), poll(), peek()** | **Collection** |
| **Deque** | **Double-ended queue allowing insertions/removals at both ends** | **Queue + SequencedCollection** | **Queue, Sequenced Collection** |
| **Blocking Queue** | **Thread-safe queue that blocks when full (on put()) or empty (on take())** | **put(e), take(), offer(e, timeout), poll(timeout)** | **Queue** |
| **Map** | **Stores key-value pairs** | **put(k,v), get(k), remove(k)** | **(DOES NOT extend Collection)** |
| **Sequenced Map** | **A Map that has a well-defined encounter order, that supports operations at both ends, and that is reversible.** | **reversed(), firstEntry(), lastEntry(), pollFirstEntry(), pollLastEntry(), putFirst(k, v), putLast(k, v), sequencedKeySet(), sequencedValues(), sequencedEntrySet()** | **Map** |
| **SortedMap** | **Maintains keys in sorted order** | **firstKey(), lastKey(), subMap(), headMap(), tailMap()** | **Sequenced Map** |
| **NavigableMap** | **SortedMap with navigation methods** | **lowerKey(k), higherKey(k),lowerEntry(k), higherEntry(k)** | **Sorted Map** |

**What are the key methods in the Collection interface?**[**#**](https://interview.in28minutes.com/interview-guides/java/collections-in-java/#what-are-the-key-methods-in-the-collection-interface%3F)

**Collection is root interface of the collection framework!**

**Key Methods**

* **add(E element): Adds an element to the collection.**
* **remove(Object element): Removes an element from the collection.**
* **size(): Returns the number of elements in the collection.**
* **isEmpty(): Checks if the collection is empty.**
* **clear(): Removes all elements from the collection.**
* **contains(Object element): Checks if an element exists in the collection.**
* **containsAll(Collection<?> c): Checks if all elements in another collection exist.**
* **addAll(Collection<? extends E> c): Adds all elements from another collection.**
* **removeAll(Collection<?> c): Removes all matching elements.**
* **retainAll(Collection<?> c): Keeps only elements present in another collection.**
* **iterator(): Returns an iterator to traverse elements.**
* **stream(): Returns a stream using the elements in the collection**
* **parallelStream(): Returns a parallel stream using the elements in the collection**

**interface Collection<E> extends Iterable<E> {**

**boolean *add*(E element);**

**boolean *remove*(Object element);**

**int *size*();**

**boolean *isEmpty*();**

**void *clear*();**

**boolean *contains*(Object element);**

**boolean *containsAll*(Collection<?> c);**

**boolean *addAll*(Collection<? extends E> c);**

**boolean *removeAll*(Collection<?> c);**

**boolean *retainAll*(Collection<?> c);**

**Iterator<E> *iterator*();**

**Stream<E> *stream*();**

**Stream<E> *parallelStream*();**

***//...***

***//...***

**}**

**What are the key methods in the SequencedCollection interface?**[**#**](https://interview.in28minutes.com/interview-guides/java/collections-in-java/#what-are-the-key-methods-in-the-sequencedcollection-interface%3F)

**Introduced in JDK 21, SequencedCollection extends Collection Interface. It represents a collection that has a well-defined encounter order, that supports operations at both ends, and that is reversible.**

**Key Methods**

* **addFirst(E e): Adds an element at the beginning of the collection.**
* **addLast(E e): Adds an element at the end of the collection.**
* **getFirst(): Retrieves the first element of the collection.**
* **getLast(): Retrieves the last element of the collection.**
* **removeFirst(): Removes and returns the first element of the collection.**
* **removeLast(): Removes and returns the last element of the collection.**
* **reversed(): Returns a view of the collection with elements in reverse order.**

**public interface SequencedCollection<E> extends Collection<E> {**

**void *addFirst*(E e);**

**void *addLast*(E e);**

**E *getFirst*();**

**E *getLast*();**

**E *removeFirst*();**

**E *removeLast*();**

**SequencedCollection<E> *reversed*();**

**}**

**What are the key methods in the List interface?**[**#**](https://interview.in28minutes.com/interview-guides/java/collections-in-java/#what-are-the-key-methods-in-the-list-interface%3F)

* **Extension of SequencedCollection**
  + **List extends SequencedCollection, inheriting all its methods.**
  + **Provides additional methods for position-based operations.**
  + **A list may have duplicate elements.**
* **Insertion Order is Maintained**
  + **Maintains the insertion order of elements.**
  + **Elements are stored in the order they are added.**
* **Inserting Elements**
  + **Use add(int position, E element) to insert at a specific position.**
  + **Elements added without position go to the end.**

**Example:**

**List<Integer> numbers = new ArrayList<>();**

***// Insertion Order is Maintained***

**numbers.*add*(10); *// index 0***

**numbers.*add*(20); *// index 1***

**numbers.*add*(30); *// index 2***

**System.out.*println*(numbers); *// [10, 20, 30]***

***// Inserting Elements - position-based operation***

**numbers.*add*(1, 15); *// insert 15 at index 1***

**System.out.*println*(numbers); *// [10, 15, 20, 30]***

***// Duplicates allowed***

**numbers.*add*(20); *// add duplicate***

**System.out.*println*(numbers); *// [10, 15, 20, 30, 20]***

***// Fast lookup using index (O(1))***

**System.out.*println*(numbers.*get*(2)); *// 20***

**interface List<E> extends SequencedCollection<E> {**

**boolean *addAll*(int index, Collection<? extends E> c);**

**E *get*(int index);**

**E *set*(int index, E element);**

**void *add*(int index, E element);**

**E *remove*(int index);**

**int *indexOf*(Object element);**

**int *lastIndexOf*(Object element);**

**ListIterator<E> *listIterator*();**

**ListIterator<E> *listIterator*(int index);**

**List<E> *subList*(int fromIndex, int toIndex);**

**static <E> List<E> *of*(); *//Static methods added in JDK 9***

**}**

**How do you decide which List implementation to use?**[**#**](https://interview.in28minutes.com/interview-guides/java/collections-in-java/#how-do-you-decide-which-list-implementation-to-use%3F)

**📌 1. ArrayList – Best for Fast Random Access**

**Key Characteristics:**

* **Uses a dynamic array internally**
* **Maintains insertion order of elements**
* **Supports indexed access to elements**

**When to Use:**

* **Fast element access via get(index)**
* **Insertions/removals mostly at the end**
* **You don’t need thread safety**

**When to Avoid:**

* **Frequent insertions or deletions in the middle**
* **Large shifts in elements (reallocation is expensive)**

**📌 2. LinkedList – Best for Frequent Insertions/Deletions**

**Key Characteristics:**

* **Uses a doubly linked list internally**
* **Each element has references to the next and previous elements**

**When to Use:**

* **Frequent insertions or deletions in the middle**

**When to Avoid:**

* **Fast random access (get(index)) is needed**
* **Memory overhead is a concern (extra references per element)**

**📌 3. Vector – Best for Thread-Safe Lists**

**Key Characteristics:**

* **Uses a dynamic array like ArrayList**
* **All methods are synchronized (thread-safe)**
* **Slower than ArrayList due to synchronization overhead**

**When to Use:**

* **Multiple threads access the list simultaneously**
* **Working with legacy code that already uses Vector**

**When to Avoid:**

* **Thread safety is not required (ArrayList is faster)**
* **Performance is important (synchronization adds overhead)**

**📌 4. CopyOnWriteArrayList – Best for Read-Mostly, Thread-Safe Scenarios**

**Key Characteristics:**

* **Thread-safe variant of ArrayList**
* **Creates a new copy of the array on each write (add/remove)**
* **Safe to iterate even during concurrent modifications**

**When to Use:**

* **Concurrent, read-heavy scenarios**

**When to Avoid:**

* **Frequent writes or large lists (copying is expensive)**

**How do you choose the right approach to sort a List?**[**#**](https://interview.in28minutes.com/interview-guides/java/collections-in-java/#how-do-you-choose-the-right-approach-to-sort-a-list%3F)

**📌 1. Using Collections.sort() – Best for Simple Sorting**

* **Uses natural ordering (e.g., alphabetical for Strings, ascending for numbers).**
* **Modifies the original list in place.**

**When to Use:**

* **Sorting a list of Strings or primitive types.**
* **Sorting objects that implement Comparable.**

**Example:**

**List<String> numbers = new ArrayList<>();**

**numbers.*add*("one");**

**numbers.*add*("two");**

**numbers.*add*("three");**

**numbers.*add*("four");**

**Collections.*sort*(numbers);**

**System.out.*println*(numbers); *// [four, one, three, two]***

**📌 2. Using Comparable – Best for Natural Ordering**

* **Implement Comparable<T> in the class.**
* **Used when objects have a default sorting order (e.g., sorting players by runs).**

**When to Use:**

* **When an object has a natural ordering (e.g., sorting Cricketers by runs).**
* **If the class controls its sorting logic.**

**Example:**

**class Cricketer implements Comparable<Cricketer> {**

**int runs;**

**String name;**

**public Cricketer(String name, int runs) {**

**this.name = name;**

**this.runs = runs;**

**}**

**@Override**

**public int *compareTo*(Cricketer that) {**

**return Integer.*compare*(this.runs, that.runs);**

**}**

**@Override**

**public String *toString*() {**

**return name + " " + runs;**

**}**

**}**

**List<Cricketer> players = Arrays.*asList*(**

**new Cricketer("Virat", 12200),**

**new Cricketer("Rohit", 11600),**

**new Cricketer("Gill", 2400),**

**new Cricketer("Dhoni", 10700)**

**);**

***// Sort using Comparable (ascending by runs)***

**Collections.*sort*(players);**

**📌 3. Using Comparator – Best for Custom Sorting**

* **Implement Comparator<T> separately from the class.**
* **Allows sorting by different criteria (e.g., descending order).**

**When to Use:**

* **When multiple sorting orders are needed (e.g., sorting by name, then by runs).**
* **When modifying the original class is not possible.**

**Example:**

**import java.util.Comparator;**

**class DescendingSorter implements Comparator<Cricketer> {**

**@Override**

**public int *compare*(Cricketer c1, Cricketer c2) {**

**return Integer.*compare*(c2.runs, c1.runs);**

**}**

**}**

**List<Cricketer> players = Arrays.*asList*(**

**new Cricketer("Virat", 12200),**

**new Cricketer("Rohit", 11600),**

**new Cricketer("Gill", 2400),**

**new Cricketer("Dhoni", 10700)**

**);**

***// Sort using Comparator (descending by runs)***

**Collections.*sort*(players, new DescendingSorter());**

**📌 4. Comparison of Sorting Approaches**

| **Approach** | **When to Use** | **Modifies Original Class?** |
| --- | --- | --- |
| **Collections.sort()** | **Simple sorting of Strings/numbers** | **No** |
| **Comparable** | **When objects have a default sorting order** | **Yes (compareTo())** |
| **Comparator** | **For custom sorting orders** | **No** |

**What are the key interfaces associated with Set operations?**[**#**](https://interview.in28minutes.com/interview-guides/java/collections-in-java/#what-are-the-key-interfaces-associated-with-set-operations%3F)

* **Set – Extends Collection interface. Does not allow duplicates.**
* **SequencedSet - A collection that is both a Sequenced Collection and a Set. Represents a collection of unique elements with a well-defined order.**
* **SortedSet – Extends SequencedSet and maintains sorted order. Provides methods like first(), last(), subSet(), headSet(), tailSet()**
* **NavigableSet – Extends SortedSet and provides navigation methods like lower(), higher(), floor(), ceiling().**

***//Set - NO DUPLICATES***

**Set<String> simpleSet = new HashSet<>();**

**simpleSet.*add*("Apple");**

**simpleSet.*add*("Banana");**

**simpleSet.*add*("Apple"); *// Ignored***

***// Set: [Banana, Apple]***

**System.out.*println*("Set: " + simpleSet);**

***// SequencedSet: SET + ORDER***

**SequencedSet<String> sequencedSet = new LinkedHashSet<>();**

**sequencedSet.*add*("One");**

**sequencedSet.*add*("Two");**

**sequencedSet.*add*("Three");**

***// No output, modifies set***

**sequencedSet.*addFirst*("Zero"); *// Java 21+***

***// No output, modifies set***

**sequencedSet.*addLast*("Four"); *// Java 21+***

***// SeqSet: [Zero, One, Two, Three, Four]***

**System.out.*println*("SeqSet: " + sequencedSet);**

***// First: Zero***

**System.out.*println*("First: " + sequencedSet.*getFirst*());**

***// Last: Four***

**System.out.*println*("Last: " + sequencedSet.*getLast*());**

***// SortedSet: SEQUENCED SET + SORTED ORDER***

**SortedSet<Integer> sortedSet = new TreeSet<>();**

**sortedSet.*add*(30);**

**sortedSet.*add*(10);**

**sortedSet.*add*(20);**

***// Sorted: [10, 20, 30]***

**System.out.*println*("Sorted: " + sortedSet);**

***// First: 10***

**System.out.*println*("First: " + sortedSet.*first*());**

**System.out.*println*("First: " + sortedSet.*getFirst*());**

***// Last: 30***

**System.out.*println*("Last: " + sortedSet.*last*());**

**System.out.*println*("Last: " + sortedSet.*getLast*());**

***// HeadSet: [10]***

**System.out.*println*("HeadSet: " + sortedSet.*headSet*(20));**

***// TailSet: [20, 30]***

**System.out.*println*("TailSet: " + sortedSet.*tailSet*(20));**

***// SubSet: [10, 20]***

**System.out.*println*("SubSet: " + sortedSet.*subSet*(10, 30));**

***//NavigableSet - SORTED SET + NAVIGATION***

**NavigableSet<Integer> navSet = new TreeSet<>();**

**navSet.*add*(10);**

**navSet.*add*(20);**

**navSet.*add*(30);**

***// NavSet: [10, 20, 30]***

**System.out.*println*("NavSet: " + navSet);**

***// Lower: 10***

**System.out.*println*("Lower: " + navSet.*lower*(20));**

***// Floor: 20***

**System.out.*println*("Floor: " + navSet.*floor*(20));**

***// Higher: 30***

**System.out.*println*("Higher: " + navSet.*higher*(20));**

***// Ceiling: 20***

**System.out.*println*("Ceiling: " + navSet.*ceiling*(20));**

**interface Set<E> extends Collection<E> {**

***// No duplicate elements allowed***

***// No ordering guaranteed***

**}**

**interface SequencedSet<E> extends Set<E>, SequencedCollection<E> {**

**SequencedSet<E> *reversed*();**

**void *addFirst*(E e); *//From SequencedCollection***

**void *addLast*(E e); *//From SequencedCollection***

**E *getFirst*(); *//From SequencedCollection***

**E *getLast*(); *//From SequencedCollection***

**E *removeFirst*(); *//From SequencedCollection***

**E *removeLast*(); *//From SequencedCollection***

**}**

**interface SortedSet<E> extends SequencedSet<E> {**

**SortedSet<E> *subSet*(E start, E end); *// start to end-1***

**SortedSet<E> *headSet*(E end); *// < end***

**SortedSet<E> *tailSet*(E start); *// >= start***

**E *first*();**

**E *last*();**

**}**

**interface NavigableSet<E> extends SortedSet<E> {**

**E *lower*(E e); *//largest x such that x < e***

**E *floor*(E e); *//largest x such that x <= e***

**E *ceiling*(E e);*//smallest x such that x >= e***

**E *higher*(E e); *//smallest x such that x > e***

**E *pollFirst*(); *//return and remove first element***

**E *pollLast*(); *//return and remove last element***

**}**

**How do you decide which Set implementation to use?**[**#**](https://interview.in28minutes.com/interview-guides/java/collections-in-java/#how-do-you-decide-which-set-implementation-to-use%3F)

**📌 1. HashSet – Best for Fast Lookups**

**Key Characteristics:**

* **Unordered collection**
* **Does not allow duplicate elements**

**When to Use:**

* **Fast lookups and uniqueness are priorities**
* **You don’t care about the order of elements**

**When to Avoid:**

* **You need to preserve insertion or sorted order**
* **You need thread safety**

**📌 2. LinkedHashSet – Best for Maintaining Insertion Order**

**Key Characteristics:**

* **Extends HashSet**
* **Maintains insertion order**
* **Slightly slower than HashSet**

**When to Use:**

* **You want predictable iteration order**
* **Need fast operations + ordering**

**When to Avoid:**

* **Order doesn’t matter (HashSet is faster)**
* **You need sorting (TreeSet is more appropriate)**
* **You need thread safety**

**📌 3. TreeSet – Best for Sorted Elements**

**Key Characteristics:**

* **Implements NavigableSet**
* **Maintains elements in natural or custom sort order**
* **Backed by a Red-Black Tree**

**When to Use:**

* **You need sorted elements**
* **You need Range-based queries (subSet, headSet, tailSet)**

**When to Avoid:**

* **Sorting is unnecessary (HashSet is faster)**
* **You need thread safety**

**TreeSet<Integer> set = new TreeSet<>();**

**set.*add*(55); set.*add*(25); set.*add*(35);**

**set.*add*(5); set.*add*(45);**

**System.out.*println*(set); *// [5, 25, 35, 45, 55]***

**System.out.*println*(set.*first*()); *// 5***

**System.out.*println*(set.*last*()); *// 55***

**System.out.*println*(set.*subSet*(10, 50)); *// [25, 35, 45]***

**System.out.*println*(set.*headSet*(35)); *// [5, 25]***

**System.out.*println*(set.*tailSet*(35)); *// [35, 45, 55]***

**System.out.*println*(set.*lower*(25)); *// 5***

**System.out.*println*(set.*floor*(25)); *// 25***

**System.out.*println*(set.*higher*(25)); *// 35***

**System.out.*println*(set.*ceiling*(25)); *// 25***

**📌 4. CopyOnWriteArraySet – Best for Read-Mostly, Thread-Safe Scenarios**

**Key Characteristics:**

* **Thread-safe Set implementation**
* **All write operations (add/remove) create a new copy**

**When to Use:**

* **Many reads, few writes**
* **You need thread-safety**

**When to Avoid:**

* **Frequent writes or large data sets (copying is expensive)**

**📌 5. ConcurrentSkipListSet – Best for Concurrent & Sorted Access**

**Key Characteristics:**

* **Thread-safe version of TreeSet**
* **Backed by a concurrent skip list**
* **Maintains sorted order**

**When to Use:**

* **Multi-threaded applications needing sorted data**

**When to Avoid:**

* **Single-threaded scenarios (overhead is unnecessary)**

**What are the main interfaces related to Queue?**[**#**](https://interview.in28minutes.com/interview-guides/java/collections-in-java/#what-are-the-main-interfaces-related-to-queue%3F)

| **Interface** | **Description** | **Key Methods** | **Extends** |
| --- | --- | --- | --- |
| **Queue** | **Collection that holds elements for processing in FIFO order** | **offer(e), poll(), peek()** | **Collection** |
| **Deque** | **Double-ended queue allowing insertion/removal from both ends** | **addFirst(e), removeLast(), peekFirst()** | **Queue,SequencedCollection** |
| **BlockingQueue** | **Thread-safe queue that blocks when full/empty** | **put(e), take(), poll(timeout)** | **Queue** |

**Queue Interface**

* **Extends the Collection interface.**
* **Used for holding elements in order for processing.**
* **Key Methods:**
  + **peek(): Retrieves the head without removal.**
  + **poll(): Retrieves and removes the head.**

***// Queue example using LinkedList***

**Queue<String> queue = new LinkedList<>();**

**queue.*offer*("A");**

**queue.*offer*("B");**

***// A (peek without remove)***

**System.out.*println*(queue.*peek*());**

***// A (remove and return)***

**System.out.*println*(queue.*poll*());**

***// B (now at head)***

**System.out.*println*(queue.*peek*());**

**interface Queue<E> extends Collection<E> {**

***// inserts element, exception if full***

**boolean *add*(E e);**

***// inserts element, returns false if full***

**boolean *offer*(E e);**

***// removes head, exception if empty***

**E *remove*();**

***// removes head, null if empty***

**E *poll*();**

***// gets head, exception if empty***

**E *element*();**

***// gets head, null if empty***

**E *peek*();**

**}**

**Deque Interface**

* **Extends Queue & SequencedCollection interfaces**
* **Supports insertion and removal from both ends.**

***// Deque example using ArrayDeque***

**Deque<String> deque = new ArrayDeque<>();**

**deque.*addFirst*("B");**

**deque.*addLast*("C");**

**deque.*addFirst*("A");**

***// A***

**System.out.*println*(deque.*peekFirst*());**

***// C***

**System.out.*println*(deque.*peekLast*());**

***// A (remove front)***

**System.out.*println*(deque.*removeFirst*());**

***// C (remove end)***

**System.out.*println*(deque.*removeLast*());**

**public interface Deque<E>**

**extends Queue<E>, SequencedCollection<E> {**

***// add to front, exception if full***

**void *addFirst*(E e);**

***// add to end, exception if full***

**void *addLast*(E e);**

***// add to front, false if full***

**boolean *offerFirst*(E e);**

***// add to end, false if full***

**boolean *offerLast*(E e);**

***// remove from front, exception if empty***

**E *removeFirst*();**

***// remove from end, exception if empty***

**E *removeLast*();**

***// remove from front, null if empty***

**E *pollFirst*();**

***// remove from end, null if empty***

**E *pollLast*();**

***// get front, exception if empty***

**E *getFirst*();**

***// get end, exception if empty***

**E *getLast*();**

***// get front, null if empty***

**E *peekFirst*();**

***// get end, null if empty***

**E *peekLast*();**

***// remove first match***

**boolean *removeFirstOccurrence*(Object o);**

***// remove last match***

**boolean *removeLastOccurrence*(Object o);**

**}**

**BlockingQueue Interface**

* **A thread-safe queue designed for use in producer-consumer scenarios**
* **Blocks:**
  + **On put(e) if the queue is full**
  + **On take() if the queue is empty**

***// BlockingQueue example with ArrayBlockingQueue***

***// Queue with capacity 2***

**BlockingQueue<String> bq =**

**new ArrayBlockingQueue<>(2);**

***//Produce elements***

**bq.*put*("A");**

**bq.*put*("B");**

***// The queue is now full***

***// The next line will block if uncommented***

***// bq.put("C");***

***// Consume one element***

**System.out.*println*(bq.*take*()); *// A***

***// Add another element (now space is available)***

**bq.*put*("C");**

***// Consume remaining elements***

**System.out.*println*(bq.*take*()); *// B***

**System.out.*println*(bq.*take*()); *// C***

**interface BlockingQueue<E> extends Queue<E> {**

***// inserts, waits if full***

**void *put*(E e) throws InterruptedException;**

***// inserts with timeout, waits if full***

**boolean *offer*(E e, long timeout, TimeUnit unit)**

**throws InterruptedException;**

***// removes, waits if empty***

**E *take*() throws InterruptedException;**

***// removes with timeout, waits if empty***

**E *poll*(long timeout, TimeUnit unit)**

**throws InterruptedException;**

***// remaining capacity in queue***

**int *remainingCapacity*();**

***// removes all into given collection***

**int *drainTo*(Collection<? super E> c);**

***// removes up to maxElements into given collection***

**int *drainTo*(Collection<? super E> c, int maxElements);**

**}**

**What are the important implementations of Queue?**[**#**](https://interview.in28minutes.com/interview-guides/java/collections-in-java/#what-are-the-important-implementations-of-queue%3F)

| **Name** | **Description** | **Extends** | **Implements** |
| --- | --- | --- | --- |
| **PriorityQueue** | **Queue where elements are sorted by priority** | **AbstractQueue** | **Queue** |
| **ArrayDeque** | **Resizable array-based Deque** | **AbstractCollection** | **Deque** |
| **ArrayBlockingQueue** | **Array-backed blocking queue (Fixed Size)** | **AbstractQueue** | **BlockingQueue** |
| **LinkedBlockingQueue** | **Linked-list blocking queue** | **AbstractQueue** | **BlockingQueue** |
| **ConcurrentLinkedQueue** | **Lock-free thread-safe queue** | **AbstractQueue** | **Queue** |
| **ConcurrentLinkedDeque** | **Lock-free thread-safe deque** | **AbstractCollection** | **Deque** |

**What are the key interfaces related to Map?**[**#**](https://interview.in28minutes.com/interview-guides/java/collections-in-java/#what-are-the-key-interfaces-related-to-map%3F)

**Here are the Map-related interfaces in Java:**

**1. Map Interface**

* **What? A collection that stores key-value pairs.**
* **Why? Allows quick lookup and modification of values using keys.**
* **Key Characteristics:**
  + **Does not extend Collection.**
  + **Keys are unique, but values can be duplicate.**
  + **Uses Entry objects to store key-value pairs.**

**Map<String, Integer> map = new HashMap<>();**

**map.*put*("Alice", 25);**

**map.*put*("Bob", 30);**

**System.out.*println*(map.*get*("Alice")); *// Output: 25***

**Important Methods:**

| **Method** | **Description** |
| --- | --- |
| **put(K key, V value)** | **Adds a key-value pair.** |
| **get(Object key)** | **Retrieves the value for a given key.** |
| **remove(Object key)** | **Removes a key-value pair.** |
| **containsKey(Object key)** | **Checks if a key exists.** |
| **containsValue(Object value)** | **Checks if a value exists.** |
| **keySet()** | **Returns all keys.** |
| **values()** | **Returns all values.** |
| **entrySet()** | **Returns all key-value pairs.** |

**2. SequencedMap Interface**

* **What? A Map that maintains a well-defined sequence of entries**
* **Why? Allows consistent iteration order and reverse traversal**

**Key Characteristics:**

* **Extends Map**
* **Maintains the insertion or access order (depending on implementation)**
* **Adds methods to get/remove first and last entries**
* **Useful when both ordering and key-based access are important**

**SequencedMap<String, Integer> seqMap = new LinkedHashMap<>();**

**seqMap.*put*("A", 1);**

**seqMap.*put*("B", 2);**

**seqMap.*put*("C", 3);**

**System.out.*println*(seqMap.*firstEntry*()); *// A=1***

**System.out.*println*(seqMap.*lastEntry*()); *// C=3***

**Important Methods:**

| **Method** | **Description** |
| --- | --- |
| **firstEntry()** | **Returns the first entry in sequence** |
| **lastEntry()** | **Returns the last entry in sequence** |
| **pollFirstEntry()** | **Removes and returns the first entry** |
| **pollLastEntry()** | **Removes and returns the last entry** |
| **reversed()** | **Returns a reversed view of the map** |
| **putFirst(K key, V value)** | **Inserts entry at the beginning of sequence** |
| **putLast(K key, V value)** | **Inserts entry at the end of sequence** |

**3. SortedMap Interface**

* **What? A Map that maintains keys in sorted order.**
* **Why? Useful when sorting keys automatically.**
* **Key Characteristics:**
  + **Extends SequencedMap.**
  + **Keys are sorted in natural order or by a Comparator.**

**SortedMap<Integer, String> sortedMap = new TreeMap<>();**

**sortedMap.*put*(3, "C");**

**sortedMap.*put*(1, "A");**

**sortedMap.*put*(2, "B");**

**System.out.*println*(sortedMap); *// Output: {1=A, 2=B, 3=C}***

**Important Methods:**

| **Method** | **Description** |
| --- | --- |
| **firstKey()** | **Returns the first (lowest) key.** |
| **lastKey()** | **Returns the last (highest) key.** |
| **subMap(fromKey, toKey)** | **Returns a portion of the map.** |
| **headMap(toKey)** | **Returns view of map with keys < toKey** |
| **tailMap(fromKey)** | **Returns view of map with keys ≥ fromKey** |

**4. NavigableMap Interface**

* **What? An extension of SortedMap with navigation methods.**
* **Why? Provides methods to find the closest matches for keys.**
* **Key Characteristics:**
  + **Allows floor, ceiling, higher, and lower key lookups.**
  + **Supports reverse order traversal.**

**NavigableMap<Integer, String> navMap = new TreeMap<>();**

**navMap.*put*(1, "A");**

**navMap.*put*(3, "C");**

**navMap.*put*(5, "E");**

**System.out.*println*(navMap.*floorKey*(4)); *// Output: 3***

**System.out.*println*(navMap.*ceilingKey*(4)); *// Output: 5***

**Important Methods:**

| **Method** | **Description** |
| --- | --- |
| **floorKey(K key)** | **Returns the largest key ≤ given key** |
| **ceilingKey(K key)** | **Returns the smallest key ≥ given key** |
| **lowerKey(K key)** | **Returns the largest key < given key** |
| **higherKey(K key)** | **Returns the smallest key > given key** |
| **floorEntry(K key)** | **Returns entry with the largest key ≤ given key** |
| **ceilingEntry(K key)** | **Returns entry with the smallest key ≥ given key** |
| **lowerEntry(K key)** | **Returns entry with the largest key < given key** |
| **higherEntry(K key)** | **Returns entry with the smallest key > given key** |
| **descendingMap()** | **Returns a reverse-order view of the map** |

**What are the key implementations of Map Interface?**[**#**](https://interview.in28minutes.com/interview-guides/java/collections-in-java/#what-are-the-key-implementations-of-map-interface%3F)

| **Name** | **Description** | **Extends** | **Implements** |
| --- | --- | --- | --- |
| **HashMap** | **Unordered key-value store (NOT thread safe)** | **AbstractMap** | **Map** |
| **LinkedHashMap** | **Map with insertion order (NOT thread safe)** | **HashMap** | **SequencedMap** |
| **TreeMap** | **Sorted key-value store (Red-Black tree)(NOT thread safe)** | **AbstractMap** | **NavigableMap** |
| **ConcurrentHashMap** | **Thread-safe map with high concurrency** | **AbstractMap** | **ConcurrentMap (Map)** |
| **ConcurrentSkipListMap** | **Thread-safe, sorted map (Keys are sorted)** | **AbstractMap** | **ConcurrentNavigableMap (NavigableMap)** |

**What is the purpose of the Collections class?**[**#**](https://interview.in28minutes.com/interview-guides/java/collections-in-java/#what-is-the-purpose-of-the-collections-class%3F)

**The Collections class provides utility methods for working with collections:**

| **Feature** | **Method** | **Purpose** |
| --- | --- | --- |
| **Sorting** | **sort(List)** | **Sorts a list in natural order.** |
| **Searching** | **binarySearch(List, key)** | **Finds an element in a sorted list.** |
| **Thread Safety** | **synchronizedList(List)** | **Creates a thread-safe list.** |
| **Unmodifiable Collections** | **unmodifiableList(List)** | **Makes a list read-only.** |
| **Finding Min/Max** | **min(Collection), max(Collection)** | **Gets the smallest or largest element.** |
| **Shuffling/Reversing** | **shuffle(List), reverse(List)** | **Randomizes or reverses order.** |

**import java.util.\*;**

**public class CollectionsExample {**

**public static void *main*(String[] args) {**

***// Create a list of numbers***

**List<Integer> numbers**

**= new ArrayList<>(Arrays.*asList*(10, 5, 20, 15));**

***// Sort the list***

**Collections.*sort*(numbers);**

***// Output: [5, 10, 15, 20]***

**System.out.*println*("Sorted List: " + numbers);**

***// Find the minimum and maximum values***

**int min = Collections.*min*(numbers);**

**int max = Collections.*max*(numbers);**

***// Output: Min: 5, Max: 20***

**System.out.*println*("Min: " + min + ", Max: " + max);**

***// Reverse the list***

**Collections.*reverse*(numbers);**

***// Output: [20, 15, 10, 5]***

**System.out.*println*("Reversed List: " + numbers);**

***// Shuffle the list***

**Collections.*shuffle*(numbers);**

***// Output: Random order***

**System.out.*println*("Shuffled List: " + numbers);**

***// Perform binary search (List must be sorted first)***

**Collections.*sort*(numbers);**

**int index = Collections**

**.*binarySearch*(numbers, 15);**

***// Output: Index position of 15***

**System.out.*println*("Index of 15: " + index);**

***// Make the list unmodifiable***

**List<Integer> unmodifiableList**

**= Collections.*unmodifiableList*(numbers);**

**System.out.*println*(**

**"Unmodifiable List: " + unmodifiableList);**

***// Uncommenting below line***

***// will throw UnsupportedOperationException***

***// unmodifiableList.add(25);***

**}**

**}**

**How do Synchronized and Concurrent Collections differ?**[**#**](https://interview.in28minutes.com/interview-guides/java/collections-in-java/#how-do-synchronized-and-concurrent-collections-differ%3F)

| **Feature** | **Synchronized Collections** | **Concurrent Collections** |
| --- | --- | --- |
| **Synchronization Method** | **Uses synchronized methods & blocks** | **Uses modern concurrency mechanisms like locks and non-blocking algorithms** |
| **Performance** | **Slower due to full synchronization** | **Faster as multiple threads can work concurrently** |
| **Thread Safety** | **Only one thread can access the collection at a time** | **Multiple threads can safely modify the collection** |
| **Examples** | **Vector, Hashtable, Collections. synchronizedList()** | **ConcurrentHashMap, CopyOnWriteArrayList, ConcurrentLinkedQueue** |
| **Best Use Case** | **When multiple threads rarely modify the collection** | **When multiple threads frequently modify the collection** |

**What is the CopyOnWrite approach in Concurrent Collections?**[**#**](https://interview.in28minutes.com/interview-guides/java/collections-in-java/#what-is-the-copyonwrite-approach-in-concurrent-collections%3F)

* **What? A technique where modifications create a new copy of the collection instead of modifying the existing one.**
* **Why? Ensures thread safety without synchronization during read operations.**
* **How?**
  + **Stores values in an immutable internal array.**
  + **When modified, a new array is created with the updated data.**
  + **Read operations are fast and do not block, only write operations are synchronized.**
* **Best Use Case: When reads outnumber writes**
* **Example Implementations: CopyOnWriteArrayList, CopyOnWriteArraySet.**

**What is the Compare-And-Swap (CAS) approach?**[**#**](https://interview.in28minutes.com/interview-guides/java/collections-in-java/#what-is-the-compare-and-swap-(cas)-approach%3F)

* **What? A low-level, non-blocking synchronization technique used for thread-safe updates**
* **Why? Avoids locking, improving performance in high-concurrency scenarios**
* **How it works:**
  1. **Read the current value (expected value)**
  2. **Compute the new value**
  3. **Compare the current value with the expected value**
  4. **If they match, update to the new value**
  5. **If not, retries until it succeeds or exits (in very rare cases)**
* **Used In: ConcurrentLinkedQueue, AtomicInteger etc..**

**In what scenarios does a Java Collection throw UnsupportedOperationException?**[**#**](https://interview.in28minutes.com/interview-guides/java/collections-in-java/#in-what-scenarios-does-a-java-collection-throw-unsupportedoperationexception%3F)

* **What? Thrown when an operation is not supported by a collection.**
* **Fixed Size Lists**
  + **Example: Arrays.asList()**
  + **Operations that change the size of a list, like add()/remove(), throw UnsupportedOperationException**
  + **Operations that do not change the size of the list, like set(), are allowed**
* **Unmodifiable Collections**
  + **Example: List.of(), Set.of(), Collections.unmodifiableList()**
  + **All mutation operations (like add(), remove(), set(), clear()) throw UnsupportedOperationException`**

**Example Code:**

**List<String> list = Arrays.*asList*("A", "B");**

**list.*add*("C"); *// Throws UnsupportedOperationException***

**List<String> immutable = List.*of*("X", "Y");**

**immutable.*remove*(0); *// Throws UnsupportedOperationException***

**What is the difference between Fail-Fast and Fail-Safe Iterators?**[**#**](https://interview.in28minutes.com/interview-guides/java/collections-in-java/#what-is-the-difference-between-fail-fast-and-fail-safe-iterators%3F)

**1. What is a Fail-Fast Iterator?**

* **What? Throws ConcurrentModificationException if the collection is modified during iteration.**
* **Where? Used in non-thread-safe collections like ArrayList, HashMap, etc.**
* **When? Happens when a structural modification (add/remove) occurs outside the iterator.**

**import java.util.HashMap;**

**import java.util.Iterator;**

**import java.util.Map;**

**public class FailFast {**

**public static void *main*(String[] args) {**

**Map<String, String> map = new HashMap<>();**

**map.*put*("key1", "value1");**

**map.*put*("key2", "value2");**

**map.*put*("key3", "value3");**

**Iterator<String> iterator = map.*keySet*().*iterator*();**

**while (iterator.*hasNext*()) {**

**System.out.*println*(map.*get*(iterator.*next*()));**

***// Throws ConcurrentModificationException***

**map.*put*("key4", "value4");**

**}**

**}**

**}**

**2. What is a Fail-Safe Iterator?**

* **What? Does not throw exceptions on modification.**
* **Where? Used in concurrent collections (ConcurrentHashMap, CopyOnWriteArrayList).**
* **Why? Works on a separate copy of the data, ensuring safe iteration.**

**import java.util.Iterator;**

**import java.util.concurrent.ConcurrentHashMap;**

**public class FailSafe {**

**public static void *main*(String[] args) {**

**ConcurrentHashMap<String, String> map**

**= new ConcurrentHashMap<>();**

**map.*put*("key1", "value1");**

**map.*put*("key2", "value2");**

**map.*put*("key3", "value3");**

**Iterator<String> iterator = map.*keySet*().*iterator*();**

**while (iterator.*hasNext*()) {**

**System.out.*println*(map.*get*(iterator.*next*()));**

**map.*put*("key4", "value4"); *// No Exception***

**}**

**}**

**}**

**3. Fail-Fast vs Fail-Safe**

| **Feature** | **Fail-Fast** | **Fail-Safe** |
| --- | --- | --- |
| **Modification** | **Throws Concurrent Modification Exception** | **No exception** |
| **Thread Safety** | **Not safe** | **Thread-safe** |
| **Works On** | **Direct collection** | **A copy of the collection** |
| **Examples** | **ArrayList, HashMap** | **ConcurrentHashMap, CopyOnWriteArrayList** |

**What are some best practices for using Collections in Java?**[**#**](https://interview.in28minutes.com/interview-guides/java/collections-in-java/#what-are-some-best-practices-for-using-collections-in-java%3F)

**1. Choose the Right Collection Type**

* **Invest time to decide the right List, Map, Queue, Set or Map to use**

**2. Set Initial Capacity to Reduce Resizing**

* **Prevents frequent resizing and improves performance.**

***// Avoids unnecessary resizing***

**Map<String, Integer> map = new HashMap<>(100);**

**3. Use the Right Data Structure for Thread Safety**

* **Use ConcurrentHashMap instead of HashMap in multi-threaded environments.**
* **Use CopyOnWriteArrayList for read-heavy operations with rare writes.**

**4. Use Immutable Collections When Possible**

* **Prevents accidental modification and improves safety.**
* **Use Collections.unmodifiableList() or List.of().**

***// Immutable List***

**List<String> names =**

**List.*of*("Alice", "Bob", "Charlie");**

**5. Prefer Iterators or Streams Over Manual Loops**

* **Improves readability and avoids IndexOutOfBoundsException.**

***// More readable than a for-loop***

**list.*forEach*(System.out::*println*);**

**Why are Sequenced Collections introduced?**[**#**](https://interview.in28minutes.com/interview-guides/java/collections-in-java/#why-are-sequenced-collections-introduced%3F)

* **Problem: Getting/Modifying first/last elements is tricky**
* **Solution: Java 21 added SequencedCollection interface**

**Sequenced Collection Interfaces**

**interface SequencedCollection<E> extends Collection<E> {**

**void *addFirst*(E element);**

**void *addLast*(E element);**

**E *getFirst*();**

**E *getLast*();**

**E *removeFirst*();**

**E *removeLast*();**

**SequencedCollection<E> *reversed*();**

**}**

**interface SequencedSet<E>**

**extends Set<E>, SequencedCollection<E> {**

**SequencedSet<E> *reversed*();**

**}**

**interface SequencedMap<K, V> extends Map<K, V> {**

**SequencedMap<K, V> *reversed*();**

**SequencedSet<K> *sequencedKeySet*();**

**SequencedCollection<V> *sequencedValues*();**

**SequencedSet<Entry<K, V>> *sequencedEntrySet*();**

**V *putFirst*(K key, V value);**

**V *putLast*(K key, V value);**

**Entry<K, V> *firstEntry*();**

**Entry<K, V> *lastEntry*();**

**Entry<K, V> *pollFirstEntry*();**

**Entry<K, V> *pollLastEntry*();**

**}**

**Example: ArrayList**

**var courses = new ArrayList<>();**

**courses.*add*("Java");**

**courses.*addFirst*("Python");**

**courses.*addLast*("JavaScript");**

**System.out.*println*(**

**courses.*getFirst*()); *// Python***

**System.out.*println*(**

**courses.*getLast*()); *// JavaScript***

**System.out.*println*(**

**courses.*reversed*()); *// [JavaScript, Java, Python]***

**Example: LinkedHashSet**

**var courses = new LinkedHashSet<>(**

**List.*of*("Spring", "AWS", "Azure"));**

**courses.*addFirst*("Java");**

**courses.*addLast*("Kotlin");**

**System.out.*println*(**

**courses.*getFirst*()); *// Java***

**System.out.*println*(**

**courses.*getLast*()); *// Kotlin***

**System.out.*println*(**

**courses.*reversed*()); *// [Kotlin, Azure, AWS, Spring, Java]***

**Example: LinkedHashMap**

**var courses = new LinkedHashMap<>();**

**courses.*put*(1, "Spring");**

**courses.*put*(2, "Spring Boot");**

**courses.*put*(3, "Spring AI");**

**courses.*putFirst*(0, "Java");**

**courses.*putLast*(4, "Kotlin");**

***//{4=Kotlin, 3=Spring AI, 2=Spring Boot, 1=Spring, 0=Java}***

**System.out.*println*(courses.*reversed*());**

***//[0, 1, 2, 3, 4]***

**System.out.*println*(courses.*sequencedKeySet*());**

***//[Java, Spring, Spring Boot, Spring AI, Kotlin]***

**System.out.*println*(courses.*sequencedValues*());**

***//0=Java***

**System.out.*println*(courses.*firstEntry*());**

***//4=Kotlin***

**System.out.*println*(courses.*lastEntry*());**

***//0=Java (Removes and returns the first key-value pair)***

**System.out.*println*(courses.*pollFirstEntry*());**

***//4=Kotlin***

**System.out.*println*(courses.*pollLastEntry*());**

**Generics in Java**

**Why Do We Need Generics in Java?**[**#**](https://interview.in28minutes.com/interview-guides/java/generics-in-java/#why-do-we-need-generics-in-java%3F)

**What Are Generics?**

* **What? A way to create type-safe and reusable code in Java.**
* **Why? Eliminates the need for multiple versions of the same class for different data types.**
* **How? Instead of hardcoding a specific type, we use a type parameter (ex: T).**

**Problems without Generics - 1 - Too Rigid**

* **Works for One Type Only – Code below can only store String values**
* **No Reusability – Need to rewrite class if you want Integer, Double, etc.**
* **Lots of Duplicate Code – Repeating logic for each data type gets messy**
* **Hard to Maintain – More classes to write, test, and manage for every type**
* **class MyList {**
* **private List values = new ArrayList();**
* **void *add*(String value) {**
* **values.*add*(value);**
* **}**
* **void *remove*(String value) {**
* **values.*remove*(value);**
* **}**
* **}**
* **MyList myList = new MyList();**
* **myList.*add*("Value 1");**

**myList.*add*("Value 2");**

**Problems without Generics - 2 - OR Too Much Flexibility**

* **No Type Safety – Any object type can be added without warnings**
* **Mixing Types – Strings, Integers, Objects can all go into one list**
* **Manual Casting – Must cast values when reading from the list**
* **Runtime Errors – Wrong casts cause ClassCastException at runtime**
* **Hard to Read – No clear info on what the list is supposed to store**
* **Code Example:**
* **import java.util.ArrayList;**
* **import java.util.List;**
* **class MyList {**
* **private List values = new ArrayList(); *// Raw list, no generics***
* **void *add*(Object value) {**
* **values.*add*(value);**
* **}**
* **void *remove*(Object value) {**
* **values.*remove*(value);**
* **}**
* **Object *get*(int index) {**
* **return values.*get*(index);**
* **}**
* **}**
* ***//Usage***
* **MyList myList = new MyList();**
* **myList.*add*("Value 1");**
* **myList.*add*("Value 2");**
* **myList.*add*(123); *// ✅ Allowed – no type safety***
* ***// Optional: safe cast if you \*know\* the type***
* **String first = (String) myList.*get*(0); *// ✅ okay***

***// Integer wrong = (Integer) myList.get(0); // ❌ runtime error***

**Solution: Using Generics**

* **Use Type Parameter T – Replace hardcoded type like String.**
* **Flexible and Reusable – One class works for all data types.**
* **Type Safe – Compiler checks type at compile time.**
* **No Casting Needed – You get the right type automatically.**
* **class MyListGeneric<T> {**
* **private List<T> values = new ArrayList<>();**
* **void *add*(T value) {**
* **values.*add*(value);**
* **}**
* **void *remove*(T value) {**
* **values.*remove*(value);**
* **}**
* **T *get*(int index) {**
* **return values.*get*(index);**
* **}**
* **}**
* **MyListGeneric<String> myListString**
* **= new MyListGeneric<>();**
* **myListString.*add*("Value 1"); *// ✅ Type safe***
* **myListString.*add*("Value 2");**
* ***// myListString.add(10); // ❌ Compile-time error***
* ***// ✅ No casting required***
* **String strValue = myListString.*get*(0);**
* **MyListGeneric<Integer> myListInteger**
* **= new MyListGeneric<>();**
* **myListInteger.*add*(1);**
* **myListInteger.*add*(2);**

**Integer num = myListInteger.*get*(0);**

**Generic Class vs Generic Method – What’s the Real Difference?**[**#**](https://interview.in28minutes.com/interview-guides/java/generics-in-java/#generic-class-vs-generic-method-%E2%80%93-what%E2%80%99s-the-real-difference%3F)

**What is a Generic Class?**

* **Uses Type Parameter T in class definition – Works with different data types.**
* **Code Reusability – One class, many data type options.**
* **Type Safe – Compiler checks the type during compilation.**
* **No Need for Casting – You get the right type automatically.**
* **class Box<T> {**
* **private T value;**
* **public void *set*(T value) {**
* **this.value = value;**
* **}**
* **public T *get*() {**
* **return value;**
* **}**
* **}**
* ***// Using Generic Class***
* **Box<String> stringBox = new Box<>();**
* **stringBox.*set*("Hello");**
* **System.out.*println*(stringBox.*get*()); *// Output: Hello***
* **Box<Integer> intBox = new Box<>();**
* **intBox.*set*(10);**
* **System.out.*println*(intBox.*get*()); *// Output: 10***
* ***//Example Generic Class From JDK***
* **public class ArrayList<E> extends AbstractList<E> {**
* **public E *set*(int index, E element) {*/\*Code\*/*}**
* **public boolean *add*(E e) {*/\*Code\*/*}**
* **public void *addFirst*(E e) {*/\*Code\*/*}**
* **public void *addLast*(E e) {*/\*Code\*/*}**
* **public Iterator<E> *iterator*() {*/\*Code\*/*}**

**}**

**What Is a Generic Method?**

* **Method Defines Its Own <T> – Not tied to class-level generic types.**
* **Flexible and Reusable – Method works with any data type.**
* **Common in Utility Classes – Like sorting, printing, comparing.**
* **class Utility {**
* ***// Generic Method (T can be any type)***
* **public static <T> void *printArray*(T[] array) {**
* **for (T element : array) {**
* **System.out.*print*(element + " ");**
* **}**
* **System.out.*println*();**
* **}**
* **}**
* ***// Using Generic Method***
* **String[] words = {"Hello", "World"};**
* **Integer[] numbers = {1, 2, 3};**
* **Utility.*printArray*(words); *// Output: Hello World***
* **Utility.*printArray*(numbers); *// Output: 1 2 3***
* ***//Example Generic Methods From JDK***
* **public class Collections {**
* **public static <T> void *sort*(List<T> list) {*/\*Code\*/*}**
* **public static <T> boolean *replaceAll*(**
* **List<T> list, T oldVal, T newVal) {*/\*Code\*/*}**

**}**

**Differences Between Generic Class and Generic Method**

| **Feature** | **Generic Class** | **Generic Method** |
| --- | --- | --- |
| **Scope** | **Applies to all methods in the class** | **Only applies to the specific method** |
| **Type Parameter Definition** | **Defined at class level (class Box<T>)** | **Defined within method (<T> void method())** |
| **Use Case** | **Used when multiple methods in a class need the same type** | **Used when only one method needs a type parameter** |
| **Example** | **Box<T> for storing generic values** | **printArray<T>() for printing arrays** |

**Do Not Forget About Generic Interfaces**

* **What? Interfaces can define type parameters just like classes**
* **Why? Makes them reusable with any type**
* **Used In? Collections, functional interfaces, comparators, etc.**
* **Code Example:**
* ***//Generic Interface***
* **interface Printer<T> {**
* **void *print*(T value);**
* **}**
* ***// Generic implementation***
* **class GenericPrinter<T> implements Printer<T> {**
* **public void *print*(T value) {**
* **System.out.*println*("Printing: " + value);**
* **}**
* **}**
* ***// Examples from JDK***
* **public interface Comparable<T> {**
* **int *compareTo*(T o);**
* **}**
* **public interface Iterator<E> {**
* **boolean *hasNext*();**
* **E *next*();**
* **}**
* **public interface Supplier<T> {**
* **T *get*();**

**}**

**How Are Generics Actually Used in the JDK?**[**#**](https://interview.in28minutes.com/interview-guides/java/generics-in-java/#how-are-generics-actually-used-in-the-jdk%3F)

**1. Collections Framework (List, Set, Map, ...)**

* **Built Using Generics – Core of Java’s data structures.**
* **Type Safe – Prevents adding wrong data types.**
* **No Typecasting Needed – Compiler handles types automatically.**
* **Flexible and Reusable – Use same class with different types.**
* ***//public class ArrayList<E>***
* **List<String> names = new ArrayList<>();**
* **names.*add*("Java");**
* ***// names.add(123); // Compile error***
* **String first = names.*get*(0); *// No casting***
* ***//public class HashMap<K,V>***
* **Map<String, Integer> map = new HashMap<>();**
* **map.*put*("Alice", 25);**

**map.*put*("Bob", 30);**

**2. Comparable and Comparator Interfaces**

* **Comparable – Define natural order for instances of a Class**
* **Comparator – Define custom sort order for specific cases**
* **Example:**
* ***//public interface Comparable<T> {***
* ***// public int compareTo(T o);***
* ***//}***
* **class Person implements Comparable<Person> {**
* **private String name;**
* **private int age;**
* **public Person(String name, int age) {**
* **this.name = name;**
* **this.age = age;**
* **}**
* **public int *getAge*() { return age; }**
* **public String *getName*() { return name; }**
* **@Override**
* **public int *compareTo*(Person other) {**
* **return Integer.*compare*(this.age, other.age);**
* **}**
* **}**
* ***//USAGE***
* ***// Natural sorting (Comparable)***
* **List<Person> people = new ArrayList<>();**
* **people.*add*(new Person("Alice", 25));**
* **people.*add*(new Person("Bob", 30));**
* **Collections.*sort*(people);**
* ***// Custom sorting (Comparator)***
* ***// public interface Comparator<T> {***
* ***// int compare(T o1, T o2);***
* ***// }***
* **class NameSorter implements Comparator<Person> {**
* **public int *compare*(Person p1, Person p2) {**
* **return p1.*getName*().*compareTo*(p2.*getName*());**
* **}**
* **}**
* ***//USAGE***

**Collections.*sort*(people, new NameSorter());**

**3. Optional in Java 8+**

* **Avoids Null Checks – No more if (obj != null) madness.**
* **Type Safe Wrapper – Wraps values that may or may not be present.**
* **Cleaner Code – Handles defaults with orElse, maps with map.**
* **No NullPointerException – Safer than returning null.**
* ***//public final class Optional<T> {***
* ***// public static <T> Optional<T> of(T value) {}***
* ***//}***
* **Optional<String> optional = Optional.*of*("Hello");**
* **System.out.*println*(**
* **optional.*orElse*("Default")); *// Output: Hello***
* ***//public static<T> Optional<T> empty() {***
* **Optional<String> emptyOptional = Optional.*empty*();**
* **System.out.*println*(**

**emptyOptional.*orElse*("Default")); *// Output: Default***

**4. Stream API (Java 8+)**

* **Type-Safe Processing – Works with the type you stream.**
* **Generic All the Way – filter, map, collect keep type info.**
* **No Casting Required – Results are the right type automatically.**
* **Functional Style – Chain operations in a clean, readable way.**
* **List<String> names = List.*of*("Alice", "Bob", "Charlie");**
* ***//public interface List<E>***
* ***// public Stream<E> stream()***
* ***//public interface Stream<T>***
* ***// Stream<T> filter(Predicate<? super T> predicate);***
* **List<String> filtered = names.*stream*()**
* **.*filter*(name -> name.*startsWith*("A"))**
* **.*collect*(Collectors.*toList*());**

**System.out.*println*(filtered); *// Output: [Alice]***

**What Is Type Erasure and Why Should You Care?**[**#**](https://interview.in28minutes.com/interview-guides/java/generics-in-java/#what-is-type-erasure-and-why-should-you-care%3F)

* **Replaces Type Parameters – T becomes its bound or Object.**
* **Compile Time – Done at compile time, not runtime.**
* **No Runtime Type Info – Reflection can’t access actual type arguments.**
* **Raw Types at Runtime – JVM sees List, not List<String>.**
* **Example: Type Erasure in Action**
  + **Before Type Erasure (What Compiler Sees):**
  + **class Box<T> { *// Generic class***
  + **private T value;**
  + **public void *set*(T value) {**
  + **this.value = value;**
  + **}**
  + **public T *get*() {**
  + **return value;**
  + **}**

**}**

* + **After Type Erasure (What JVM Sees):**
  + **class Box { *// No generics at runtime***
  + **private Object value;**
  + **public void *set*(Object value) {**
  + **this.value = value;**
  + **}**
  + **public Object *get*() { return value; }**

**}**

* **Effects of Erasure**
  + **No Type Info at Runtime – JVM forgets all generic details.**
  + **instanceof Doesn’t Work – Can’t check for Box<String>.**
  + **Type Checks Are Limited – All generics look the same to JVM.**
  + **Box<String> strBox = new Box<>();**
  + ***// ❌ Compilation error***

***// if (strBox instanceof Box<String>) { }***

* **Why Does Java Use Erasure?**
  + **Backward Compatibility – Supports old Java code without generics.**
  + **Simpler Bytecode – No extra bytecode for every generic type.**
  + **No Runtime Overhead – Type info removed, so faster execution.**
  + **Easier JVM Implementation – JVM doesn’t need to handle generics.**

**How Can You Restrict Types Using Generics in Java?**[**#**](https://interview.in28minutes.com/interview-guides/java/generics-in-java/#how-can-you-restrict-types-using-generics-in-java%3F)

**1. Bounded Type (T extends SomeClass)**

* **What? Restricts T to subclasses of a specific class**
* **Why? Allows safe access to superclass methods**
* **Example? Accepts Integer, Double, Float – all extend Number**
* **Prevents? Using unrelated types like String or Boolean**
* **class MathUtils<T extends Number> {**
* **public double *square*(T num) {**
* ***//NOTE: Calling a Number specific method!***
* **return num.*doubleValue*() \* num.*doubleValue*();**
* **}**
* **}**
* **MathUtils<Integer> intMath = new MathUtils<>();**
* **System.out.*println*(intMath.*square*(5)); *// Output: 25.0***
* ***// ❌ Compilation error: String is not a subclass of Number***
* ***// MathUtils<String> strMath = new MathUtils<>();***

**2. Multiple Bounds (T extends Class & Interface)**

* **What? Restricts T to extend a class and implement one or more interfaces**
* **Why? Ensures T has both class behavior and interface methods**
* **Example? T must extend Document and implement Printable**
* **Prevents? Using types that only meet one condition (just class or just interface)**
* **interface Printable {**
* **void *print*();**
* **}**
* **class Document {}**
* **class Report<T extends Document & Printable> {**
* **private T content;**
* **public Report(T content) {**
* **this.content = content;**
* **}**
* **public void *printReport*() {**
* **content.*print*();**
* **}**
* **}**
* **class MyReport extends Document implements Printable {**
* **public void *print*() {**
* **System.out.*println*("Printing report...");**
* **}**
* **}**
* **Report<MyReport> report = new Report<>(new MyReport());**

**report.*printReport*(); *// Output: Printing report...***

**3. Upper Bound Wildcard in Methods (<? extends T>)**

* **What? Accepts any type that extends T (or is T itself)**
* **Example? You can pass Integer, Double, Float to a method expecting ? extends Number**
* **Safe For? Reading only, because all elements are guaranteed to be at least Number**
* **Prevents? Adding elements (compiler says: "I don’t know what exact type to accept!")**
* **How It Works**
  + **The method says: “Give me a list of something that’s a Number”**
  + **But I don’t know what that “something” is (could be Integer, Double, etc.)**
  + **So I’ll just read it — I can safely treat all items as Number**
  + **But I won’t write to it — I might mess up the actual subtype stored inside**
* **public class WildcardExample {**
* **public static void *printNumbers*(**
* **List<? extends Number> numbers) {**
* **for (Number n : numbers) {**
* **System.out.*println*(n);**
* **}**
* ***// ❌ Not safe to add***
* ***// type could be Integer, Double, etc.***
* ***// numbers.add(42);***
* **}**
* **public static void *main*(String[] args) {**
* **List<Integer> intList = List.*of*(1, 2, 3);**
* **List<Double> dblList = List.*of*(1.1, 2.2);**
* ***printNumbers*(intList); *// ✅ Allowed***
* ***printNumbers*(dblList); *// ✅ Allowed***
* **}**

**}**

**4. Lower Bound Wildcard in Methods (? super T)**

* **What? Accepts T or any of its supertypes**
* **Example? Accepts Integer, Number, or Object for ? super Integer**
* **Why? Allows safely adding elements of type T**
* **Prevents? Reading elements as Integer – type is too generic to trust**
* **How It Works**
  + **The method says: “Give me a list that can hold an Integer”**
  + **That could be a List<Integer>, List<Number>, or List<Object>**
  + **You can safely write an Integer to any of those**
  + **But reading? Nope. The compiler doesn’t know what exact type is inside**
* **public class SuperExample {**
* **public static void *addNumbers*(**
* **List<? super Integer> numbers) {**
* **numbers.*add*(10);**
* **numbers.*add*(20); *// ✅ safe***
* ***// ❌ Not safe to read as Integer***
* ***// Compilation Error: Type Mismatch***
* ***// numbers.get(0) is assumed of type Object***
* ***// Integer num = numbers.get(0);***
* ***//Needs Casting***
* ***//Integer num = (Integer) numbers.get(0);***
* **}**
* **public static void *main*(String[] args) {**
* **List<Number> nums = new ArrayList<>();**
* ***// ✅ Number is a supertype of Integer***
* ***addNumbers*(nums);**
* **List<Object> objs = new ArrayList<>();**
* ***// ✅ Object is also a supertype***
* ***addNumbers*(objs);**
* **System.out.*println*(nums); *// Output: [10, 20]***
* **}**
* **}**
* ***//Collections class***
* ***//Replaces all of the elements of the***
* ***//specified list with the specified element.***
* ***//public static <T> void fill(***

***// List<? super T> list, T obj) {***

**5. Generic Method with Both Wildcards**

* **What? A method that uses both Upper Bound Wildcard (? extends T) and Lower Bound Wildcard (? super T)**
* **Why? Offers maximum flexibility for utility methods**
* **Example? Copy elements from one list to another**
* **Use Case? Utility methods in libraries like Collections.copy()**
* **How It Works**
  + **? extends T → read from source (safe for reading)**
  + **? super T → write to destination (safe for writing)**
* **“Producer Extends, Consumer Super” (PECS rule) (collection's point of view)**
  + **If you are only reading items from a generic collection, it is a producer => use extends**
  + **If you are only writing items in, it is a consumer => use super.**
  + **If you do both with the same collection, you shouldn't use either extends or super.**
* **Prevents? Copying incompatible types (like String into Integer list)**
* **public class GenericUtils {**
* **public static <T> void *copyElements*(**
* **List<? super T> destination,**
* **List<? extends T> source) {**
* **for (T item : source) {**
* **destination.*add*(item);**
* **}**
* **}**
* **public static void *main*(String[] args) {**
* **List<Integer> source = List.*of*(1, 2, 3);**
* **List<Number> destination = new ArrayList<>();**
* ***// ✅ Integer → Number***
* ***copyElements*(destination, source);**
* ***// Output: [1, 2, 3]***
* **System.out.*println*(destination);**
* **}**
* **}**
* ***//Collections***
* ***//public static <T> void copy(***

***// List<? super T> dest, List<? extends T> src) {***

**TRICKY JAVA QUESTIONS:-**

**Why are Wrapper Classes Needed?**[**#**](https://interview.in28minutes.com/interview-guides/java/core-java-in-depth/#why-are-wrapper-classes-needed%3F)

**📌 What?**

* **Represents wrappers around primitive data types (int, double, etc.)**
* **Implemented in java.lang package (e.g., Integer, Double)**

**📌 Why?**

* **Allows primitives to be used as objects (e.g., in collections like ArrayList).**
  + **Works with collections and other generic classes (List<Integer>, Optional<Double>).**
* **Provides utility methods (e.g., Integer.parseInt(), Double.valueOf()).**
* **Supports autoboxing and unboxing, making conversions easier.**

**import java.util.ArrayList;**

**import java.util.List;**

**public class WrapperExample {**

**public static void *main*(String[] args) {**

***// Cannot use List<int>***

**List<Integer> numbers = new ArrayList<>();**

**numbers.*add*(10); *// Autoboxing (int → Integer)***

**numbers.*add*(20);**

***// Unboxing (Integer → int)***

**int sum = numbers.*get*(0) + numbers.*get*(1);**

**System.out.*println*("Sum: " + sum);**

**}**

**}**

**📌 Wrapper Classes in Java**

| **Primitive Type** | **Wrapper Class** |
| --- | --- |
| **byte** | **Byte** |
| **short** | **Short** |
| **int** | **Integer** |
| **long** | **Long** |
| **float** | **Float** |
| **double** | **Double** |
| **char** | **Character** |
| **boolean** | **Boolean** |

**📌 Example Methods in Wrapper Classes**

| **Wrapper Class** | **Method** | **Description** | **Example** |
| --- | --- | --- | --- |
| **Integer** | **valueOf (String s)** | **Converts a string to an Integer object** | **Integer num = Integer. valueOf("100");** |
| **Integer** | **parseInt (String s)** | **Converts a string to an int primitive** | **int num = Integer. parseInt("100");** |
| **Integer** | **compare (int x, int y)** | **Compares two int values** | **Integer. compare(10, 20); // returns -1** |
| **Integer** | **toBinaryString (int n)** | **Converts an int to a binary string** | **Integer. toBinaryString(5); // "101"** |
| **Double** | **valueOf (String s)** | **Converts a string to a Double object** | **Double num = Double. valueOf("3.14");** |
| **Double** | **parseDouble (String s)** | **Converts a string to a double primitive** | **double num = Double. parseDouble("3.14");** |
| **Double** | **isNaN (double d)** | **Checks if the value is NaN (Not a Number)** | **Double. isNaN(0.0 / 0.0); // true** |
| **Boolean** | **valueOf (String s)** | **Converts a string to a Boolean object** | **Boolean flag = Boolean. valueOf("true");** |
| **Boolean** | **parseBoolean (String s)** | **Converts a string to a boolean primitive** | **boolean flag = Boolean. parseBoolean("true");** |
| **Character** | **isDigit (char ch)** | **Checks if a character is a digit** | **Character. isDigit('5'); // true** |
| **Character** | **toUpperCase (char ch)** | **Converts a character to uppercase** | **Character. toUpperCase('a'); // 'A'** |
| **Character** | **toLowerCase (char ch)** | **Converts a character to lowercase** | **Character. toLowerCase('A'); // 'a'** |

**📌 Things to Remember**

* **Autoboxing: Converts a primitive into a wrapper object automatically.**

**Integer num = 10; *// int → Integer***

* **Unboxing: Converts a wrapper object back into a primitive automatically.**

**int value = num; *// Integer → int***

* **Immutable: Wrapper class objects cannot be modified after creation.**

**Best Practices in Using Wrapper Classes**[**#**](https://interview.in28minutes.com/interview-guides/java/core-java-in-depth/#best-practices-in-using-wrapper-classes)

**📌 1. Creating Wrapper Objects**

**Use valueOf() instead of new (Efficient Memory Usage)**

**✅ Preferred:**

***// Uses cached objects***

**Integer num = Integer.*valueOf*(10);**

**❌ Avoid:**

***// Unnecessary object creation***

**Integer num = new Integer(10);**

**📌 2. Avoid Autoboxing in Loops (Performance Issue)**

**Autoboxing creates unnecessary objects, leading to performance issues.**

**✅ Preferred:**

**int sum = 0;**

**for (int i = 0; i < 1000; i++) {**

**sum += i; *// Uses int (fast)***

**}**

***//Even Faster***

**int sumFunctional = IntStream.*range*(0, 1000).*sum*();**

**❌ Avoid:**

**Integer sum = 0;**

**for (int i = 0; i < 1000; i++) {**

**sum += i; *// Autoboxing occurs (slow)***

**}**

**📌 3. Use parseXxx() for String to Primitive Conversion**

**Avoids unnecessary object creation**

**✅ Preferred:**

**int num = Integer.*parseInt*("123");**

**❌ Avoid:**

***// Deprecated and inefficient***

**Integer num = new Integer("123");**

**How does Java optimize memory usage with Integer.valueOf()?**[**#**](https://interview.in28minutes.com/interview-guides/java/core-java-in-depth/#how-does-java-optimize-memory-usage-with-integer.valueof()%3F)

**📌 What?**

* **Integer.valueOf(int) caches commonly used integer values.**
* **Instead of creating a new object, it reuses an existing one from the cache when possible.**

**📌 Why?**

* **Reduces memory usage by avoiding unnecessary object creation.**
* **Improves performance since cached objects are returned instead of allocating new memory.**

**📌 How?**

* **Java maintains a cache of Integer objects for values from -128 to 127.**
* **When Integer.valueOf(n) is called within this range, it returns a cached object instead of creating a new one.**
* **If the number is outside the range, a new object is created.**

**📌 Example: Cached vs. Non-Cached Integers**

**public class IntegerCacheExample {**

**public static void *main*(String[] args) {**

**Integer a = Integer.*valueOf*(100); *// Cached***

**Integer b = Integer.*valueOf*(100); *// Cached***

**System.out.*println*(a == b); *// true (Same cached object)***

**Integer x = Integer.*valueOf*(200); *// Not Cached***

**Integer y = Integer.*valueOf*(200); *// Not Cached***

**System.out.*println*(x == y); *// false (Different objects)***

**}**

**}**

**📌 Where is this cache defined?**

* **Java uses an internal cache inside the Integer class.**
* **The cache is implemented in IntegerCache (an inner class of Integer).**

**Source Code (from Integer class)**

**private static class IntegerCache {**

**static final Integer cache[];**

**static {**

***// Cache range: -128 to 127***

**cache = new Integer[-(-128) + 127 + 1];**

**for (int i = 0; i < cache.length; i++)**

**cache[i] = new Integer(i - 128);**

**}**

**}**

**This ensures efficient memory usage for frequently used integers.**

**Cached Values for Wrapper Classes**

| **Wrapper Class** | **Cached Values** |
| --- | --- |
| **Byte** | **All values (-128 to 127)** |
| **Short** | **-128 to 127** |
| **Integer** | **-128 to 127** |
| **Long** | **-128 to 127** |
| **Character** | **0 to 127 (ASCII characters)** |
| **Boolean** | **true and false** |
| **Float** | **❌ No caching** |
| **Double** | **❌ No caching** |

**Why Are Wrapper Classes in Java Immutable?**[**#**](https://interview.in28minutes.com/interview-guides/java/core-java-in-depth/#why-are-wrapper-classes-in-java-immutable%3F)

**📌 What is Immutability?**

* **What? Immutability means an object cannot be modified after it is created.**
* **Why? Helps with thread safety, caching, and predictable behavior.**
* **How? Any modification creates a new object instead of changing the existing one.**

**Example of an Immutable Class in Java:**

* **Final class prevents subclassing.**
* **Private final fields prevent changes after initialization.**
* **No setter methods ensure immutability.**

**final class ImmutableExample {**

**private final int value;**

**ImmutableExample(int value) {**

**this.value = value;**

**}**

**public int *getValue*() {**

**return value;**

**}**

**}**

**📌 Wrapper Classes are Immutable**

* **What? Wrapper classes (Integer, Double, Boolean, etc.) are immutable in Java.**
* **Why?**
  1. **Caching Mechanism – Frequently used values (Integer.valueOf(10)) are reused, reducing memory usage.**
  2. **Thread Safety – Threads cannot modify shared values.**

**Example of Wrapper Class Immutability:**

**public class WrapperImmutableExample {**

**public static void *main*(String[] args) {**

**Integer x = 10;**

***// Creates a new Integer object,***

***//old one is discarded***

**x = x + 1;**

**System.out.*println*(x); *// Output: 11***

**}**

**}**

**String vs StringBuffer vs StringBuilder**[**#**](https://interview.in28minutes.com/interview-guides/java/core-java-in-depth/#string-vs-stringbuffer-vs-stringbuilder)

**📌 Using String for Immutable Data**

* **What? Used for storing fixed values like constants, configuration keys, or error messages.**
* **Example:**
* **public class StringExample {**
* **public static void *main*(String[] args) {**
* **String greeting = "Hello";**
* **greeting += " World"; *// Creates a new String object***
* **System.out.*println*(greeting); *// Output: Hello World***
* **}**

**}**

* **When to Use String class?**
  + **✅ Fixed values that do not change**
  + **❌ Avoid for frequent modifications (creates multiple objects)**

**📌 Using StringBuffer for Thread-Safe Modifications**

* **What? A mutable string alternative that is thread-safe (synchronized).**
* **Why? Ensures safe string modification in multi-threaded environments.**
* **Example:**
* **public class StringBufferExample {**
* **public static void *main*(String[] args) {**
* **StringBuffer buffer**
* **= new StringBuffer("Thread-safe");**
* **buffer.*append*(" operations");**
* **buffer.*insert*(0, "Ensuring ");**
* **buffer.*reverse*();**
* **System.out.*println*(buffer);**
* **}**

**}**

* **When to Use?**
  + **✅ Multi-threaded modifications**
  + **❌ Avoid in single-threaded cases (slower than StringBuilder)**

**📌 Using StringBuilder for Fast String Modifications**

* **What? A mutable string alternative optimized for performance.**
* **Why? Faster than StringBuffer because it is not synchronized.**
* **Example:**
* **public class StringBuilderExample {**
* **public static void *main*(String[] args) {**
* **StringBuilder builder**
* **= new StringBuilder("Efficient");**
* **builder.*append*(" string");**
* **builder.*append*(" operations");**
* **builder.*delete*(0, 4); *// Removes "Effe"***
* **System.out.*println*(builder);**
* **}**

**}**

* **When to Use?**
  + **✅ Frequent modifications in single-threaded programs**
  + **❌ Avoid in multi-threaded environments (not thread-safe)**

**📌 Avoiding Performance Issues with String Concatenation in Loops**

**❌ Inefficient Approach Using String: Creates 1000+ unnecessary String objects, impacting performance.**

**public class StringLoopExample {**

**public static void *main*(String[] args) {**

**String result = "";**

**for (int i = 0; i < 1000; i++) {**

***// Creates a new String object in every iteration***

**result += i;**

**}**

**System.out.*println*(result.*length*());**

**}**

**}**

**✅ Optimized Approach Using StringBuilder: Uses a single StringBuilder object, significantly improving performance.**

**public class StringBuilderLoopExample {**

**public static void *main*(String[] args) {**

**StringBuilder result = new StringBuilder();**

**for (int i = 0; i < 1000; i++) {**

**result.*append*(i);**

**}**

**System.out.*println*(result.*length*());**

**}**

**}**

**📌 Comparison**

| **Factor** | **String** | **StringBuffer** | **StringBuilder** |
| --- | --- | --- | --- |
| **Immutable Data** | **✅ YES** | **❌ NO** | **❌ NOT** |
| **Thread Safety** | **✅ Thread-Safe** | **✅ Thread-Safe** | **❌ Not Safe** |
| **Performance (Loops)** | **❌ Slowest** | **❌ Slower (due to synchronization)** | **✅ Fast** |
| **Frequent Changes** | **❌ Creates new objects** | **✅ Thread-Safe** | **✅ Best for performance** |

**Why Are String Classes in Java Immutable?**[**#**](https://interview.in28minutes.com/interview-guides/java/core-java-in-depth/#why-are-string-classes-in-java-immutable%3F)

* **What? The String class in Java is immutable, meaning its value cannot be changed after creation.**
* **Why?**
  1. **Security – Strings are used in class loading, networking, and security keys. If mutable, an attacker could change "password123" to "password456".**
  2. **Thread Safety – Since a String object cannot change, it can be shared safely across threads.**
  3. **String Pooling Optimization – Multiple references to the same string reuse the same object, saving memory.**

**Example of String Immutability:**

**public class StringImmutableExample {**

**public static void *main*(String[] args) {**

**String s = "Hello";**

**s.*concat*(" World"); *// Creates a new object***

***//does NOT modify `s`***

**System.out.*println*(s); *// Output: Hello***

**}**

**}**

* **Why does "Hello" remain unchanged? → s.concat(" World") creates a new String, but s still points to "Hello".**

**How do Text Blocks Help?**[**#**](https://interview.in28minutes.com/interview-guides/java/core-java-in-depth/#how-do-text-blocks-help%3F)

**Example Without Text Blocks:**

**String json = "{\n" +**

**" \"name\": \"John\",\n" +**

**" \"age\": 30\n" +**

**"}";**

**Same Example With Text Blocks:**

**String json = """**

**{**

**"name": "John",**

**"age": 30**

**}**

**""";**

***//json ==> "{\n \"name\": \"John\",\n \"age\": 30\n}\n"***

**Summary:**

* **Java New Feature: Introduced in JDK 15**
* **Less Escape Sequences: No more \n or \" for multi-line strings.**
* **More Readable: Code looks like actual text formatting.**
* **Auto Alignment: Handles indentation automatically.**
* **Better for JSON, HTML, SQL: Keeps structure intact.**

**What is a String Pool?**[**#**](https://interview.in28minutes.com/interview-guides/java/core-java-in-depth/#what-is-a-string-pool%3F)

* **What? A special memory area inside the heap where string literals are stored only once.**
* **Why? Saves memory by reusing the same String object instead of creating duplicates.**
* **How? When a new string is created using double quotes (""), it is automatically added to the pool.**
  + **If the same string already exists, Java reuses the existing reference instead of creating a new object.**

**Example of String Pool Optimization:**

**Both s1 and s2 refer to the same "Java" object in the pool, improving memory efficiency.**

**public class StringPoolExample {**

**public static void *main*(String[] args) {**

**String s1 = "Java"; *// Stored in String Pool***

**String s2 = "Java"; *// Reuses the same object***

**System.out.*println*(s1 == s2);**

***// Output: true (same reference)***

**}**

**}**

**Heap Memory**

* **When using new String("Hello"), a new object is always created in the heap memory, even if the same value exists in the pool.**
* **This results in two objects: one in the heap and one in the pool (if not already present).**

**Example:**

**public class StringMemoryExample {**

**public static void *main*(String[] args) {**

***// Stored in String Constant Pool***

**String s1 = "Hello";**

***// Reuses the same object***

**String s2 = "Hello";**

***// Creates a new object in Heap***

**String s3 = new String("Hello");**

***// true (same reference)***

**System.out.*println*(s1 == s2);**

***// false (different objects)***

**System.out.*println*(s1 == s3);**

**}**

**}**

**🎯 Best Practice: Always prefer string literals (String str = "Hello";) to save memory and improve performance.**

**How Does intern() Work?**[**#**](https://interview.in28minutes.com/interview-guides/java/core-java-in-depth/#how-does-intern()-work%3F)

* **What? Forces a string to be stored in the String Pool, even if it was created using new.**
* **Why? Helps reduce duplicate objects and optimize memory.**

**Example of intern() Method:**

* **s1 is created outside the pool, but s1.intern() moves it into the pool.**
* **s2 and s3 now share the same reference, saving memory.**

**public class StringInternExample {**

**public static void *main*(String[] args) {**

**String s1 = new String("Hello");**

**String s2 = s1.*intern*(); *// Moves to String Pool***

**String s3 = "Hello";**

**System.out.*println*(s2 == s3); *// Output: true***

**}**

**}**

**What are the things to be careful about when comparing Strings?**[**#**](https://interview.in28minutes.com/interview-guides/java/core-java-in-depth/#what-are-the-things-to-be-careful-about-when-comparing-strings%3F)

1. **Using == instead of .equals()**
   * **== compares references, not values.**
   * **Always use .equals() to check content equality.**
2. **String s1 = "Hello";**
3. **String s2 = new String("Hello");**
4. **System.out.*println*(s1 == s2); *// false (different objects)***

**System.out.*println*(s1.*equals*(s2)); *// true (same content)***

1. **Case Sensitivity**
   * **equals() is case-sensitive. Use equalsIgnoreCase() for case-insensitive checks.**
2. **System.out.*println*(**

**"hello".*equalsIgnoreCase*("HELLO")); *// true***

1. **String Pool Behavior**
   * **String literals are stored in the String Constant Pool and can be shared.**
   * **Using new String() creates a new object, even if the same value exists.**
2. **String a = "Java";**
3. **String b = "Java";**
4. ***// true (same object in pool)***
5. **System.out.*println*(a == b);**
6. **String c = new String("Java");**
7. ***// false (different objects)***

**System.out.*println*(a == c);**

**What Are the Best Practices with Conditionals?**[**#**](https://interview.in28minutes.com/interview-guides/java/core-java-in-depth/#what-are-the-best-practices-with-conditionals%3F)

**✅ Use else if Instead of Multiple if Statements**

**if (age < 18) {**

**System.out.*println*("Minor");**

**} else if (age < 60) {**

**System.out.*println*("Adult");**

**} else {**

**System.out.*println*("Senior");**

**}**

**✅ Avoid Complex Conditions**

**Hard to Read (Complex Condition in if)**

**if ((age > 18 && hasLicense)**

**|| (age > 16 && hasPermit && !hasViolations)) {**

**System.out.*println*("Allowed to drive");**

**}**

**Improved Readability (Using Meaningful Variables)**

**boolean isAdultWithLicense = age > 18 && hasLicense;**

**boolean isTeenWithPermit = age > 16**

**&& hasPermit && !hasViolations;**

**if (isAdultWithLicense || isTeenWithPermit) {**

**System.out.*println*("Allowed to drive");**

**}**

**✅ Prefer Ternary Operator for Simple Cases**

***// ❌ Verbose***

**String result;**

**if (x > 10) result = "High"; else result = "Low";**

***// ✅ Use ternary***

**String result = (x > 10) ? "High" : "Low";**

**✅ Use switch Instead of Multiple if-else When Applicable**

**String result = switch (day) {**

***//Add other days :)***

**case "Monday", "Tuesday" -> "Workday";**

**default -> "Weekend";**

**};**

**System.out.*println*(result);**

**What Are the Best Practices with Loops?**[**#**](https://interview.in28minutes.com/interview-guides/java/core-java-in-depth/#what-are-the-best-practices-with-loops%3F)

**✅ Use Enhanced for Loop or Functional Programming for Collections (Faster & Readable)**

***// ❌ Slower: Uses index-based access repeatedly***

**for (int i = 0; i < list.*size*(); i++) {**

**System.out.*println*(list.*get*(i));**

**}**

***// ✅ Faster: Uses iterator internally***

**for (String item : list) {**

**System.out.*println*(item);**

**}**

***// ✅ ALTERNATIVE: Functional Programming***

**list.*forEach*(System.out::*println*);**

**✅ Avoid Expensive Operations Inside Loops**

***// ❌ Inefficient: Calls `list.size()` in each iteration***

**for (int i = 0; i < list.*size*(); i++) {**

***process*(list.*get*(i));**

**}**

***// ✅ Efficient: Store size in a variable***

**int size = list.*size*();**

**for (int i = 0; i < size; i++) {**

***process*(list.*get*(i));**

**}**

***// ✅ ALTERNATIVE: Functional Programming***

**list.*forEach*(this::*process*);**

**✅ Use StringBuilder Instead of String for Concatenation**

***// ❌ Inefficient: Creates new String objects in each iteration***

**String result = "";**

**for (String word : words) {**

**result += word; *// New object each time***

**}**

***// ✅ Efficient: Uses a single `StringBuilder` object***

**StringBuilder result = new StringBuilder();**

**for (String word : words) {**

**result.*append*(word);**

**}**

**✅ Use break and continue Wisely**

***// ❌ Inefficient: Loops through all elements***

***// even after finding the match***

**for (int num : numbers) {**

**if (num == target) {**

**System.out.*println*("Found!");**

**}**

**}**

***// ✅ Efficient: Stops loop once found***

**for (int num : numbers) {**

**if (num == target) {**

**System.out.*println*("Found!");**

**break; *// Exits loop early***

**}**

**}**

**✅ Use Parallel Streams for Large Data Sets (Java 8+)**

***// ✅ Parallel processing for large lists***

**list.*parallelStream*().*forEach*(System.out::*println*);**

**✅ Choose the Right Loop Type**

| **Scenario** | **Best Loop Type** |
| --- | --- |
| **Fixed number of iterations** | **for loop** |
| **Loop until condition is met** | **while loop** |
| **Iterating over a collection** | **Enhanced for loop** |
| **Processing large datasets** | **parallelStream()** |

**What Are the Best Practices with Using Arrays?**[**#**](https://interview.in28minutes.com/interview-guides/java/core-java-in-depth/#what-are-the-best-practices-with-using-arrays%3F)

**✅ Use Arrays.toString() to Print Arrays**

**int[] myArray = {1, 2, 3, 4};**

***// ❌ Prints something like [I@1b6d3586***

**System.out.*println*(myArray);**

***// ✅ Prints: [1, 2, 3, 4]***

**System.out.*println*(Arrays.*toString*(myArray));**

**✅ Use Enhanced for Loop or Functional Programming instead of Traditional for Loop**

**for (int num : myArray) {**

**System.out.*println*(num);**

**}**

**Arrays.*stream*(myArray).*forEach*(System.out::*println*);**

**✅ Use Arrays.equals() to Compare Arrays**

**if (Arrays.*equals*(array1, array2)) {**

**System.out.*println*("Arrays are equal");**

**}**

***//If you compare arrays using ==,***

***//it checks only the memory references, NOT the actual contents.***

**✅ Use Arrays.deepEquals() for Multidimensional Arrays**

**if (Arrays.*deepEquals*(array1, array2)) {**

**System.out.*println*("2D Arrays are equal");**

**}**

**✅ Use Arrays.copyOf() for Cloning**

**int[] original = {1, 2, 3, 4, 5};**

**int[] copy = Arrays.*copyOf*(original, original.length);**

***//[1, 2, 3]***

**int[] partialCopy = Arrays.*copyOf*(original, 3);**

***//[1, 2, 3, 4, 5, 0, 0]***

**int[] expanded = Arrays.*copyOf*(original, 7);**

***//Creates a new array → No modification of the original.***

***//Simple way to resize arrays***

**✅ Use System.arraycopy() for Fast Copying of values *(Faster than looping for large arrays)***

***//dest is an existing array***

**System.*arraycopy*(src, 0, dest, 0, src.length);**

***//Can copy from any index***

***//Faster (native implementation)***

**✅ Use List Instead of Arrays When Possible**

***//Allows Adding/Removing Elements***

**List<String> list = new ArrayList<>(Arrays.*asList*(array));**

**✅ Use Streams for Advanced Array Operations**

**int sum = Arrays.*stream*(myArray).*sum*();**

**How does Local Variable Type Inference Help? (Java 10)**[**#**](https://interview.in28minutes.com/interview-guides/java/core-java-in-depth/#how-does-local-variable-type-inference-help%3F-(java-10))

**Example:**

***// ArrayList<String> numbers = new ArrayList<>();***

***// No need to repeat ArrayList<String>***

**var numbers = new ArrayList<String>();**

***// Inferred as HashMap<Integer, String>***

**var map = new HashMap<Integer, String>();**

**Advantages**

* **Less Code: Reduces redundancy in variable declarations.**
* **Improves Readability: Focus on logic, not type details.**
* **Flexible: Works with generics, loops, and complex types.**
* **Compiler Safety: Still enforces strong typing at compile-time.**

**How do Records improve the conciseness of Java code?**[**#**](https://interview.in28minutes.com/interview-guides/java/core-java-in-depth/#how-do-records-improve-the-conciseness-of-java-code%3F)

* **Records eliminate boilerplate code in Java Beans by automatically generating:**
  + **Constructor**
  + **Getters (accessors)**
  + **toString(), equals(), and hashCode()**
* **Makes code more readable and maintainable.**

**Before Records (Verbose Java Class)**

**public class Person {**

**private final String name;**

**private final String email;**

**private final String phoneNumber;**

**public Person(**

**String name, String email,**

**String phoneNumber) {**

**this.name = name;**

**this.email = email;**

**this.phoneNumber = phoneNumber;**

**}**

**public String *getName*() {**

**return name;**

**}**

**public String *getEmail*() {**

**return email;**

**}**

**public String *getPhoneNumber*() {**

**return phoneNumber;**

**}**

**@Override**

**public boolean *equals*(Object obj) {**

***/\* Implementation \*/***

**}**

**@Override**

**public int *hashCode*() {**

***/\* Implementation \*/***

**}**

**@Override**

**public String *toString*() {**

***/\* Implementation \*/***

**}**

**}**

**With Records (Concise and Readable)**

**public record Person(**

**String name, String email,**

**String phoneNumber) {}**

***//You Can Write Code like this!***

**Person person = new Person(**

**"John Doe",**

**"john@example.com",**

**"1234567890");**

***// Auto-generated toString()***

**System.out.*println*(person);**

***// No need for getName()***

**System.out.*println*(person.*name*());**

***// No need for getEmail()***

**System.out.*println*(person.*email*());**

**What are the best practices with Records?**[**#**](https://interview.in28minutes.com/interview-guides/java/core-java-in-depth/#what-are-the-best-practices-with-records%3F)

**📌 Use Records for Immutable Data**

* **Best suited for DTOs (Data Transfer Objects) and configuration objects .**

**public record UserDTO(String username, String role) {}**

**📌 Add Custom Validation in Compact Constructors**

* **Use a compact constructor to enforce field constraints. (runs automatically during object creation, validating name. If name is null or blank, it throws an exception.)**
* **public record Person(String name, String email) {**
* **public Person {**
* **if (name == null || name.*isBlank*()) {**
* **throw new IllegalArgumentException(**
* **"Name cannot be empty");**
* **}**
* **}**

**}**

**📌 Use Methods for Additional Functionality**

* **Records can contain instance methods for derived values.**
* **public record Rectangle(int width, int height) {**
* **public int *area*() {**
* **return width \* height;**
* **}**

**}**

**Rectangle rect = new Rectangle(5, 10);**

**System.out.*println*(rect.*area*()); *// Output: 50***

**📌 Use Static Fields and Methods When Needed**

* **Records do NOT allow instance variables, but you can use static fields.**
* **public record Car(String model, int year) {**
* **static String manufacturer = "Tesla";**
* **public static String *getManufacturer*() {**
* **return manufacturer;**
* **}**

**}**

**What are the things that you should be careful about when using Records?**[**#**](https://interview.in28minutes.com/interview-guides/java/core-java-in-depth/#what-are-the-things-that-you-should-be-careful-about-when-using-records%3F)

**📌 Records Are Immutable**

* **Fields cannot be changed after object creation.**
* **Person p = new Person("Alice", "alice@example.com");**

***// p.name = "Bob"; // Compilation Error***

* **If a class needs mutable state or complex behavior, use a regular class instead.**
* **public class BankAccount {**
* **private double balance;**
* **public void *deposit*(double amount) {**
* **balance += amount;**
* **}**
* **public void *withdraw*(double amount) {**
* **balance -= amount;**
* **}**

**}**

**📌 Cannot Add Instance Variables**

* **Fields must be declared in the record header.**
* **public record Product(String name, double price) {**
* ***// private int discount;***
* ***// ❌ ERROR - Cannot add instance variables***

**}**

**📌 Inheritance is Not Supported**

* **Records extend java.lang.Record.**
* **Records cannot extend other classes (but can implement interfaces).**
* **public record Employee(**
* **String name, int id) */\* extends Person \*/* {}**

***// ❌ ERROR***

**📌 When to Use Records?**

* **Data Models (e.g., UserDTO)**
* **Configurations (e.g., AppSettings)**
* **Lightweight Domain Objects (e.g., Rectangle, Person)**

**📌 When NOT to Use Records?**

* **Mutable Objects (e.g., BankAccount)**
* **Complex Business Logic**
* **Inheritance-Based Hierarchies**

**📌 Good to Know: Simple Record Pattern Matching**

**record Course(int id, String name) {}**

**public static void *processCourse*(Object obj) {**

**if (obj instanceof Course(int id, String name)) {**

**System.out.*println*(**

**"Course ID: " + id + ", Name: " + name);**

**}**

**}**

**📌 Good to Know: Nested Record Pattern Matching**

**record Person(String name, int age) {}**

**record Address(String street, String city) {}**

**record Contact(Person person, Address address) {}**

**public static void *processContact*(Object obj) {**

**if (obj instanceof Contact(**

**Person(var name, var age),**

**Address(var street, var city))) {**

**System.out.*println*(name + " lives in " + city);**

**}**

**}**

**Summary:**

* **Records introduced in JDK 16**
* **Advanced Pattern Matching in JDK 21**

**JAVA8 QUESTIONS:-**

**How does Functional Programming differ from Imperative Style?**[**#**](https://interview.in28minutes.com/interview-guides/java/functional-programming-in-java/#how-does-functional-programming-differ-from-imperative-style%3F)

**What is Imperative Style?**

* **Uses step-by-step logic to solve a problem.**
* **Focuses on how to achieve the result.**

**What is Functional Programming?**

* **Uses declarative code instead of loops.**
* **Focuses on what needs to be done, not how.**

**Example 1: Imperative Code (Using Loops)**

**Explicitly loops through the list and keeps a running sum.**

**import java.util.List;**

**public class FP02Imperative {**

**private static int *addListImperative*(List<Integer> numbers) {**

**int sum = 0;**

**for (int number : numbers) {**

**sum += number;**

**}**

**return sum;**

**}**

**}**

**Example 2: Functional Programming (Using Streams)**

**Uses stream() and reduce() to compute the sum in a single line.**

**import java.util.List;**

**public class FP02Functional {**

**private static int *addListFunctional*(List<Integer> numbers) {**

**return numbers.*stream*().*reduce*(0, Integer::*sum*);**

**}**

**}**

**Key Differences**

| **Feature** | **Imperative Style** | **Functional Programming** |
| --- | --- | --- |
| **Style** | **Step-by-step logic** | **Declarative (focus on what, not how)** |
| **Looping** | **Uses for/while loops** | **Uses Streams** |
| **State Management** | **Uses mutable variables (sum)** | **Avoids mutation** |
| **Code Readability** | **More verbose** | **More concise & readable** |

**What are the core concepts of Functional Programming? Provide an example.**[**#**](https://interview.in28minutes.com/interview-guides/java/functional-programming-in-java/#what-are-the-core-concepts-of-functional-programming%3F-provide-an-example.)

**courses.*stream*()**

**.*map*(course -> course.*toUpperCase*())**

**.*forEach*(System.out::*println*);**

**1. Stream**

* **What? A sequence of elements that supports functional-style operations.**
* **Why? Allows processing of data in a declarative and efficient way.**
* **Example:**
  + **courses.stream() creates a stream from the list of courses.**
  + **This stream enables pipeline processing without modifying the original list.**

**2. Lambda Function**

* **What? An anonymous function that represents a short piece of code.**
* **Why? Makes code concise and readable.**
* **Example:**
  + **course -> course.toUpperCase() is a lambda function that converts each course name to uppercase.**
  + **It eliminates the need for writing a separate method.**

**3. Method Reference**

* **What? A shorthand for calling a method using ClassName::methodName or object::methodName.**
* **Why? Further simplifies lambda expressions when an existing method can be used directly.**
* **Example:**
  + **System.out::println is a method reference that replaces x -> System.out.println(x).**

**What are different ways to create Streams in Java?**[**#**](https://interview.in28minutes.com/interview-guides/java/functional-programming-in-java/#what-are-different-ways-to-create-streams-in-java%3F)

**1. From Collections (Lists, Sets, Maps)**

* **What? Convert collections into streams.**
* **Why? Allows functional operations on lists, sets, and maps.**
* **How? Use .stream()**

**List<Integer> numbers = List.*of*(1, 2, 3, 4, 5);**

**Stream<Integer> numberStream = numbers.*stream*();**

**numberStream.*forEach*(System.out::*println*);**

***//{Java=100, Python=80}***

**Map<String, Integer> courses**

**= Map.*of*("Java", 100, "Python", 80);**

***//["Java", "Python"]***

**Stream<String> keyStream = courses.*keySet*().*stream*();**

***//[100, 80]***

**Stream<Integer> valueStream = courses.*values*().*stream*();**

***//{Java=100, Python=80}***

**Stream<Map.Entry<String, Integer>> entryStream**

**= courses.*entrySet*().*stream*();**

**2. Using Stream.of()**

* **What? Create a stream from individual elements.**
* **Why? Useful for small, fixed data sets.**
* **How? Use Stream.of().**

**Stream<String> stream = Stream.*of*("Apple", "Banana", "Cherry");**

**stream.*forEach*(System.out::*println*);**

**3. Using Arrays.stream()**

* **What? Convert primitive arrays into streams.**
* **How? Use Arrays.stream().**

**int[] numbers = {10, 20, 30, 40, 50};**

**IntStream intStream = Arrays.*stream*(numbers);**

**intStream.*forEach*(System.out::*println*);**

**4. Using Stream.builder()**

* **What? Manually build a stream.**
* **Why? Allows adding elements dynamically.**
* **How? Use Stream.builder().**

**Stream.Builder<String> builder = Stream.*builder*();**

**builder.*add*("Java").*add*("Python").*add*("JavaScript");**

**Stream<String> stream = builder.*build*();**

**stream.*forEach*(System.out::*println*);**

**5. Using Stream.generate()**

* **What? Create an stream of generated values.**
* **Why? Useful for lazy evaluation or dynamic data.**
* **How? Use Stream.generate().**

**Stream<Double> randomNumbers**

**= Stream.*generate*(Math::*random*).*limit*(5);**

**randomNumbers.*forEach*(System.out::*println*);**

**6. Using Stream.iterate()**

* **What? Create a stream by applying a function.**
* **Why? Useful for sequences (e.g., even numbers).**
* **How? Use Stream.iterate().**

**Stream<Integer> evenNumbers**

**= Stream.*iterate*(0, n -> n + 2).*limit*(5);**

**evenNumbers.*forEach*(System.out::*println*);**

**7. Using Files (Files.lines())**

* **What? Read a file as a stream of lines.**
* **Why? Helps process large files efficiently.**
* **How? Use Files.lines().**

**Stream<String> fileStream**

**= Files.*lines*(Paths.*get*("file.txt"));**

**fileStream.*forEach*(System.out::*println*);**

**8. Using IntStream.range()**

* **What? Create a stream of numbers within a range.**
* **Why? Useful for generating sequences.**
* **How? Use IntStream.range()**

***// Output: 1234***

**IntStream.*range*(1, 5).*forEach*(System.out::*print*);**

***// Output: 12345***

**IntStream.*rangeClosed*(1, 5).*forEach*(System.out::*print*);**

**Comparison Table**

| **Method** | **Use Case** |
| --- | --- |
| **collection.stream()** | **Convert List, Set, Map to a stream** |
| **Stream.of()** | **Create a stream from values** |
| **Arrays.stream()** | **Efficient for primitive arrays** |
| **Stream.builder()** | **Manually construct a stream** |
| **Stream.generate()** | **Infinite stream using random values** |
| **Stream.iterate()** | **Infinite stream using a pattern** |
| **Files.lines()** | **Stream file lines** |
| **IntStream.range()** | **Number range stream** |

**What operations can you apply to a Java Stream?**[**#**](https://interview.in28minutes.com/interview-guides/java/functional-programming-in-java/#what-operations-can-you-apply-to-a-java-stream%3F)

**1. Intermediate Operations**

* **What? Transform or filter elements and return a new stream.**
* **Lazy Execution: These operations do not execute until a terminal operation is called.**

| **Operation** | **Purpose** | **Example** |
| --- | --- | --- |
| **filter()** | **Retains elements that satisfy a condition.** | **numbers.stream().filter(n -> n % 2 == 0);** |
| **map()** | **Transforms each element in the stream.** | **courses.stream().map(course -> course.length());** |
| **distinct()** | **Removes duplicate elements.** | **numbers.stream().distinct();** |
| **sorted()** | **Sorts elements in natural or custom order.** | **courses.stream().sorted();** |
| **limit(n)** | **Takes only the first n elements.** | **numbers.stream().limit(3);** |
| **skip(n)** | **Skips the first n elements.** | **numbers.stream().skip(2);** |
| **takeWhile(predicate)** | **Takes elements while the condition is true, stops at first failure.** | **numbers.stream().takeWhile(n -> n < 10);** |
| **dropWhile(predicate)** | **Drops elements while the condition is true, then takes the rest.** | **numbers.stream().dropWhile(n -> n < 10);** |

**2. Terminal Operations**

* **What? Consumes the stream and produces a result.**
* **Trigger Execution: The stream is processed when a terminal operation is called.**

| **Operation** | **Purpose** | **Example** |
| --- | --- | --- |
| **forEach()** | **Iterates over each element and performs an action.** | **numbers.stream(). forEach(System.out::println);** |
| **collect()** | **Gathers elements into a collection (List, Set, Map).** | **List<Integer> squared = numbers.stream(). map(n -> n \* n). collect(Collectors.toList());** |
| **reduce()** | **Combines elements to produce a single result.** | **int sum = numbers.stream().reduce(0, Integer::sum);** |
| **count()** | **Returns the number of elements in the stream.** | **long count = numbers.stream().count();** |
| **findFirst()** | **Retrieves the first element of the stream.** | **Optional<Integer> first = numbers.stream().findFirst();** |
| **findAny()** | **Retrieves any element (non-deterministic).** | **Optional<Integer> any = numbers.stream().findAny();** |
| **anyMatch()** | **Returns true if any element matches the condition.** | **numbers.stream().anyMatch(n -> n > 10);** |
| **allMatch()** | **Returns true if all elements match the condition.** | **numbers.stream().allMatch(n -> n > 0);** |
| **noneMatch()** | **Returns true if no element matches the condition.** | **numbers.stream().noneMatch(n -> n < 0);** |

**What should you keep in mind when working with Streams?**[**#**](https://interview.in28minutes.com/interview-guides/java/functional-programming-in-java/#what-should-you-keep-in-mind-when-working-with-streams%3F)

**1. Streams Do Not Store Data**

* **What? Streams process data on the fly without storing it.**
* **Why? This makes them memory efficient, especially for large datasets.**
* **Example:**
* **Stream<Integer> numberStream**

**= List.*of*(1, 2, 3, 4, 5).*stream*();**

**2. Streams Are Lazy**

* **What? Intermediate operations (filter, map, etc.) are not executed immediately.**
* **Why? Streams only process data when a terminal operation (e.g., forEach, collect, reduce) is called.**
* **Example:**
* **List.*of*(1, 2, 3, 4, 5).*stream*().*map*(n -> {**
* **System.out.*println*("Mapping: " + n);**
* **return n \* 2;**

**}); *// No output because there is no terminal operation***

* **Intermediate operations (like filter() and map()) are lazy.**
* **NOT executed immediately when encountered**
  + **Instead, Java "chains" these operations and waits for a terminal operation (like findFirst()) to trigger their execution.**

**3. Streams Are Immutable**

* **What? Stream operations do not modify the original collection.**
* **Why? This ensures functional purity and avoids side effects.**
* **Example:**
* **List<Integer> numbers = List.*of*(1, 2, 3);**
* ***// Original list remains unchanged***

**numbers.*stream*().*map*(n -> n \* 2);**

**4. Streams Cannot Be Reused**

* **What? A stream is consumed once a terminal operation is called.**
* **Why? You need to create a new stream if you need to process the data again.**
* **Example:**
* **Stream<Integer> stream = numbers.*stream*();**
* ***// ✅ Works***
* **stream.*forEach*(System.out::*println*);**
* ***// ❌ ERROR: Stream already consumed!***

**stream.*forEach*(System.out::*println*);**

**Can you provide some examples of Lambda Functions?**[**#**](https://interview.in28minutes.com/interview-guides/java/functional-programming-in-java/#can-you-provide-some-examples-of-lambda-functions%3F)

**Lambda expressions are anonymous functions that simplify writing code by reducing boilerplate.**

**Syntax of Lambda Functions**

* **Parameters: The input to the function.**
* **Arrow (->): Separates parameters from the function body.**
* **Body: Contains the logic of the function.**

**(parameters) -> { body }**

**Filtering Even Numbers Using Lambda**

**numbers.*stream*()**

**.*filter*(number -> number % 2 == 0)**

**.*forEach*(System.out::*println*);**

**Mapping Course Names to Uppercase**

**courses.*stream*()**

**.*map*(course -> course.*toUpperCase*())**

**.*forEach*(System.out::*println*);**

**Sorting Courses by Length of Name**

**courses.*stream*()**

**.*sorted*((c1, c2) -> c1.*length*() - c2.*length*())**

**.*forEach*(System.out::*println*);**

**Using UnaryOperator to Triple a Number**

**UnaryOperator<Integer> triple = x -> x \* 3;**

**System.out.*println*(triple.*apply*(10)); *// Output: 30***

**When should you use Method References instead of Lambda Functions?**[**#**](https://interview.in28minutes.com/interview-guides/java/functional-programming-in-java/#when-should-you-use-method-references-instead-of-lambda-functions%3F)

**Method references are shortcuts for writing lambda expressions that call an existing method. They use the syntax ClassName::methodName or object::methodName.**

**📌 1. Calling a Static Method**

* **When? When an existing static method can be directly referenced without modifications.**
* **Why? It improves readability and eliminates redundant lambda expressions.**
* **Example 1: Java Utility Method**
  + **Using Method Reference**
  + **courses.*stream*()**

**.*forEach*(System.out::*println*);**

* + **Equivalent Lambda Expression**
  + **courses.*stream*()**

**.*forEach*(course -> System.out.*println*(course));**

* **Example 2: Custom Utility Method**
  + **Using Method Reference**
  + **numbers.*stream*()**
  + **.*filter*(FP01Functional::*isEven*)**

**.*forEach*(System.out::*println*);**

* + **Equivalent Lambda Expression**
  + **numbers.*stream*()**
  + **.*filter*(number -> FP01Functional.*isEven*(number))**

**.*forEach*(number -> System.out.*println*(number));**

**📌 2. Calling an Instance Method**

* **When? When calling an instance method on each element.**
* **Using Method Reference**
* **courses.*stream*()**
* **.*map*(String::*toUpperCase*)**

**.*forEach*(System.out::*println*);**

* **Equivalent Lambda Expression**
* **courses.*stream*()**
* **.*map*(course -> course.*toUpperCase*())**

**.*forEach*(course -> System.out.*println*(course));**

**📌 3. When creating new objects - Constructor Reference**

* **When? When creating new objects inside a stream.**
* **Using Constructor Reference**

**Supplier<String> stringSupplier = String::new;**

* **Equivalent Lambda Expression**

**Supplier<String> stringSupplier = () -> new String();**

**What are Functional Interfaces? Provide some examples.**[**#**](https://interview.in28minutes.com/interview-guides/java/functional-programming-in-java/#what-are-functional-interfaces%3F-provide-some-examples.)

* **Core component of Java's functional programming approach**
* **Contains exactly one abstract method**
* **Allow passing behavior as an argument using lambda expressions**
* **Examples:**
  + **Consumer processes an input.**
  + **@FunctionalInterface**
  + **public interface Consumer<T> {**
  + **void *accept*(T t);**
  + **}**
  + **Consumer<Integer> printOdd = number -> {**
  + **if (number % 2 != 0) {**
  + **System.out.*println*(number);**
  + **}**
  + **};**
  + ***//What happens in the background?***
  + **public class PrintOdd implements Consumer<Integer> {**
  + **@Override**
  + **public void *accept*(Integer number) {**
  + **if (number % 2 != 0) {**
  + **System.out.*println*(number);**
  + **}**
  + **}**
  + **}**
  + **Function transforms data and returns the result.**
  + **@FunctionalInterface**
  + **public interface Function<T, R> {**
  + **R *apply*(T t);**
  + **}**
  + **Function<Integer, Integer> squareFunction**
  + **= number -> number \* number;**
  + ***//What happens in the background?***
  + **public class SquareFunction**
  + **implements Function<Integer, Integer> {**
  + **@Override**
  + **public Integer *apply*(Integer number) {**
  + **return number \* number;**
  + **}**

**}**

* + **Predicate tests a condition.**
  + **@FunctionalInterface**
  + **public interface Predicate<T> {**
  + **boolean *test*(T t);**
  + **}**
  + **Predicate<Integer> isEven**
  + **= number -> number % 2 == 0;**
  + ***//What happens in the background?***
  + **public class IsEvenPredicate implements Predicate<Integer> {**
  + **@Override**
  + **public boolean *test*(Integer number) {**
  + **return number % 2 == 0;**
  + **}**
  + **}**
  + **BiFunction combines two inputs into a result.**
  + **@FunctionalInterface**
  + **public interface BiFunction<T, U, R> {**
  + **R *apply*(T t, U u);**
  + **}**
  + **BiFunction<Integer, Integer, Integer> maxFunction**
  + **= (x, y) -> x > y ? x : y;**
  + ***//What happens in the background?***
  + **public class MaxFunction implements BiFunction<Integer, Integer, Integer> {**
  + **@Override**
  + **public Integer *apply*(Integer x, Integer y) {**
  + **return x > y ? x : y;**
  + **}**

**}**

**What are some key Functional Interfaces, and how do they work?**[**#**](https://interview.in28minutes.com/interview-guides/java/functional-programming-in-java/#what-are-some-key-functional-interfaces%2C-and-how-do-they-work%3F)

**1. Predicate (Conditional Testing)**

* **Purpose: Tests a condition on input and returns a boolean.**
* **Example: Checking if a number is even.**
* **Using Lambda (Preferred)**

**Predicate<Integer> isEvenPredicate = x -> x % 2 == 0;**

* **Equivalent Anonymous Class**
* **Predicate<Integer> isEvenPredicate2 = new Predicate<>() {**
* **@Override**
* **public boolean *test*(Integer x) {**
* **return x % 2 == 0;**
* **}**

**};**

* **Usage in Streams**
* **numbers.*stream*()**
* **.*filter*(isEvenPredicate)**

**.*forEach*(System.out::*println*);**

**2. Function (Mapping Input to Output)**

* **Purpose: Accepts one argument and returns a transformed value.**
* **Example: Squaring a number.**
* **Using Lambda (Preferred)**

**Function<Integer, Integer> square = x -> x \* x;**

* **Equivalent Anonymous Class**
* **Function<Integer, Integer> square = new Function<>() {**
* **@Override**
* **public Integer *apply*(Integer x) {**
* **return x \* x;**
* **}**

**};**

* **Usage in Streams**
* **numbers.*stream*()**
* **.*map*(squareFunction)**

**.*forEach*(System.out::*println*);**

**3. Consumer (Processing Without Return)**

* **Purpose: Accepts an input but does not return anything.**
* **Example: Printing values.**
* **Using Lambda (Preferred)**

**Consumer<Integer> sysoutConsumer = System.out::*println*;**

* **Equivalent Anonymous Class**
* **Consumer<Integer> sysoutConsumer2 = new Consumer<>() {**
* **@Override**
* **public void *accept*(Integer x) {**
* **System.out.*println*(x);**
* **}**

**};**

* **Usage in Streams**

**numbers.*stream*().*forEach*(sysoutConsumer);**

**4. BinaryOperator (Two Inputs, Same Type)**

* **Purpose: Takes two arguments and returns a result of the same type.**
* **Example: Summing two numbers.**
* **Using Lambda**

**BinaryOperator<Integer> sum = (a, b) -> a + b;**

* **Using Method Reference**

**BinaryOperator<Integer> sum = Integer::*sum*;**

* **Equivalent Anonymous Class**
* **BinaryOperator<Integer> sumBinaryOperator2**
* **= new BinaryOperator<>() {**
* **@Override**
* **public Integer *apply*(Integer a, Integer b) {**
* **return Integer.*sum*(a, b);**
* **}**

**};**

* **Usage in Reduce Operation**
* **int sum = numbers.*stream*().*reduce*(0, sumBinaryOperator);**

**System.out.*println*(sum);**

**5. Supplier (Generating Values Without Input)**

* **Purpose: Provides a value without any input.**
* **Example: Returning a random number.**
* **Using Lambda (Preferred)**
* **Supplier<Integer> randomIntegerSupplier**

**= () -> new Random().*nextInt*(100);**

* **Equivalent Anonymous Class**
* **Supplier<Integer> randomIntegerSupplier2 = new Supplier<>() {**
* **@Override**
* **public Integer *get*() {**
* **return new Random().*nextInt*(100);**
* **}**

**};**

* **Usage**

**System.out.*println*(randomIntegerSupplier.*get*());**

**6. UnaryOperator (Single Input, Same Type)**

* **Purpose: Takes one argument and returns a result of the same type.**
* **Example: Tripling a number.**
* **Using Lambda (Preferred)**

**UnaryOperator<Integer> tripleOperator = x -> 3 \* x;**

* **Equivalent Anonymous Class**
* **UnaryOperator<Integer> tripleOperator2 = new UnaryOperator<>() {**
* **@Override**
* **public Integer *apply*(Integer x) {**
* **return 3 \* x;**
* **}**

**};**

* **Usage in Streams**
* **numbers.*stream*()**
* **.*map*(tripleOperator)**

**.*forEach*(System.out::*println*);**

**7. BiPredicate (Conditional Testing with Two Inputs)**

* **Purpose: Tests two inputs and returns a boolean.**
* **Example: Checking if two numbers are equal.**
* **Using Lambda (Preferred)**

**BiPredicate<Integer, Integer> isEqual = (a, b) -> a.*equals*(b);**

* **Equivalent Anonymous Class**
* **BiPredicate<Integer, Integer> isEqual2 = new BiPredicate<>() {**
* **@Override**
* **public boolean *test*(Integer a, Integer b) {**
* **return a.*equals*(b);**
* **}**

**};**

* **Usage**

**System.out.*println*(isEqual.*test*(5, 5)); *// Output: true***

**8. BiFunction (Two Inputs, One Output)**

* **Purpose: Accepts two inputs and returns one output.**
* **Example: Concatenating strings.**
* **Using Lambda (Preferred)**
* **BiFunction<String, String, String> concat**

**= (a, b) -> a + " " + b;**

* **Equivalent Anonymous Class**
* **BiFunction<String, String, String> concat2 = new BiFunction<>() {**
* **@Override**
* **public String *apply*(String a, String b) {**
* **return a + " " + b;**
* **}**

**};**

* **Usage**
* ***// Output: Hello World***

**System.out.*println*(concat.*apply*("Hello", "World"));**

**9. BiConsumer (Processing Two Inputs Without Return)**

* **Purpose: Accepts two inputs but does not return anything.**
* **Example: Printing a key-value pair.**
* **Using Lambda (Preferred)**
* **BiConsumer<String, Integer> printKeyValue =**

**(key, value) -> System.out.*println*(key + ": " + value);**

* **Equivalent Anonymous Class**
* **BiConsumer<String, Integer> printKeyValue2 = new BiConsumer<>() {**
* **@Override**
* **public void *accept*(String key, Integer value) {**
* **System.out.*println*(key + ": " + value);**
* **}**

**};**

* **Usage**

**printKeyValue.*accept*("Java", 8); *// Output: Java: 8***

**Summary**

| **Functional Interface** | **Purpose** | **Example Lambda** |
| --- | --- | --- |
| **Predicate** | **Returns true or false** | **x -> x % 2 == 0** |
| **Function<T, R>** | **Maps input T to output R** | **x -> x \* x** |
| **Consumer** | **Processes input T without return** | **System.out::println** |
| **BinaryOperator** | **Takes two T inputs, returns T** | **(a, b) -> a + b** |
| **Supplier** | **Generates a T value** | **() -> new Random().nextInt(100)** |
| **UnaryOperator** | **Takes T, returns T** | **x -> x \* 3** |
| **BiPredicate<T, U>** | **Tests two inputs, returns boolean** | **(a, b) -> a.equals(b)** |
| **BiFunction<T, U, R>** | **Takes two inputs, returns R** | **(a, b) -> a + " " + b** |
| **BiConsumer<T, U>** | **Processes two inputs without return** | **(k, v) -> System.out.println(k + v)** |

**Why were Primitive Functional Interfaces introduced in Java?**[**#**](https://interview.in28minutes.com/interview-guides/java/functional-programming-in-java/#why-were-primitive-functional-interfaces-introduced-in-java%3F)

**Overview**

* **What? Special interfaces for primitive data types like int, double, long.**
* **Why? Avoids boxing/unboxing overhead, making code faster and memory-efficient.**
* **Example: Use IntBinaryOperator instead of BinaryOperator<Integer>.**
* **Before (Less Efficient)**

**BinaryOperator<Integer> sum = (x, y) -> x + y;**

* **After (Better Performance)**

**IntBinaryOperator sum = (x, y) -> x + y;**

* **Key Benefits**
  + **Faster Execution: No need to wrap/unwrap primitive values.**
  + **Less Memory Usage: No Integer, just int!**
  + **Cleaner Code: Works directly with primitives, reducing complexity.**

**Other Primitive Interfaces**

| **Interface** | **Purpose** | **Example Type** |
| --- | --- | --- |
| **IntPredicate** | **Checks conditions** | **int -> boolean** |
| **IntFunction<R>** | **Maps int to result** | **int -> R** |
| **IntConsumer** | **Accepts int, no return** | **int -> void** |
| **IntBinaryOperator** | **Operates on two int** | **(int, int) -> int** |

**Other Examples: LongPredicate, LongFunction, LongConsumer, DoublePredicate, DoubleFunction, DoubleConsumer and a lot of others!**

**When Do You Use Collectors.groupingBy?**[**#**](https://interview.in28minutes.com/interview-guides/java/functional-programming-in-java/#when-do-you-use-collectors.groupingby%3F)

* **What? Groups elements of a stream into a Map based on a key.**
* **Why? Simplifies grouping, counting, and aggregating data in Java Streams.**

**public record Course(**

**String name, String category,**

**int reviewScore, int students) {}**

**List<Course> courses = List.*of*(**

**new Course("AWS", "Cloud", 95, 200000),**

**new Course("Azure", "Cloud", 90, 150000),**

**new Course("Docker", "Cloud", 85, 180000),**

**new Course("Kubernetes", "Cloud", 88, 170000),**

**new Course("Spring", "Framework", 97, 220000),**

**new Course("Spring Boot", "Framework", 93, 210000),**

**new Course("Microservices", "Microservices", 98, 250000),**

**new Course("FullStack", "FullStack", 96, 230000)**

**);**

**Group Courses by Category**

**Map<String, List<Course>> coursesByCategory = courses.*stream*()**

**.*collect*(Collectors.*groupingBy*(Course::*category*));**

**System.out.*println*(coursesByCategory);**

**Output:**

***//Simplified only to show course name***

**{Cloud=[AWS, Azure, Docker, Kubernetes],**

**Framework=[Spring, Spring Boot],**

**Microservices=[Microservices],**

**FullStack=[FullStack]}**

**Count Courses in Each Category**

**Map<String, Long> courseCountByCategory = courses.*stream*()**

**.*collect*(Collectors.*groupingBy*(**

**Course::*category*, Collectors.*counting*()));**

**System.out.*println*(courseCountByCategory);**

**Output:**

**{Cloud=4, Framework=2, Microservices=1, FullStack=1}**

**Find Highest Rated Course in Each Category**

**Map<String, Optional<Course>> highestRatedCourseByCategory = courses.*stream*()**

**.*collect*(Collectors.*groupingBy*(**

**Course::*category*,**

**Collectors.*maxBy*(Comparator.*comparing*(Course::*reviewScore*))**

**));**

**System.out.*println*(highestRatedCourseByCategory);**

**Output:**

**{Cloud=Optional[AWS],**

**Framework=Optional[Spring],**

**Microservices=Optional[Microservices],**

**FullStack=Optional[FullStack]}**

**Get List of Course Names per Category**

**Map<String, List<String>> courseNamesByCategory = courses.*stream*()**

**.*collect*(Collectors.*groupingBy*(**

**Course::*category*,**

**Collectors.*mapping*(Course::*name*, Collectors.*toList*())**

**));**

**System.out.*println*(courseNamesByCategory);**

**Output:**

**{Cloud=[AWS, Azure, Docker, Kubernetes],**

**Framework=[Spring, Spring Boot],**

**Microservices=[Microservices],**

**FullStack=[FullStack]}**

**How does the Optional class help handle missing values?**[**#**](https://interview.in28minutes.com/interview-guides/java/functional-programming-in-java/#how-does-the-optional-class-help-handle-missing-values%3F)

* **What? A container that may or may not hold a value.**
* **Why? Avoids NullPointerException by explicitly handling missing values.**
* **Example:**

***// Safe handling of null***

**Optional<String> optional**

**= Optional.*ofNullable*(*getValueFromDatabase*());**

**optional.*ifPresent*(System.out::*println*);**

| **Method** | **Purpose** | **Example Usage** |
| --- | --- | --- |
| **Optional.of(value)** | **Creates an Optional with a non-null value** | **Optional.of("Hello")** |
| **Optional.ofNullable(value)** | **Creates an Optional (null safe)** | **Optional.ofNullable("Hello")** |
| **Optional.empty()** | **Creates an empty Optional** | **Optional.empty()** |
| **isPresent()** | **Checks if a value is present** | **opt.isPresent()** |
| **ifPresent(Consumer)** | **Executes code if value exists** | **opt.ifPresent(System.out::println)** |
| **orElse(defaultValue)** | **Returns value or default if empty** | **opt.orElse("Default")** |
| **orElseGet(Supplier)** | **Returns value or calls supplier function** | **opt.orElseGet(() -> "Generated")** |
| **orElseThrow()** | **Throws an exception if value is missing** | **opt.orElseThrow(() -> new RuntimeException("No Value"))** |

**How do Parallel Streams enhance performance in Java?**[**#**](https://interview.in28minutes.com/interview-guides/java/functional-programming-in-java/#how-do-parallel-streams-enhance-performance-in-java%3F)

* **Modern multi-core processors: Today's computers have multiple cores.**
  + **They can run different tasks at the same time.**
* **Traditional Programming: Old-style coding uses loops and shared variables.**
  + **Why is this a problem? Many threads need to share the same data (e.g., a sum variable).**
  + **This makes parallel execution hard.**
  + ***//Traditional***
  + **int sum = 0;**
  + **for (int number : numbers) {**
  + **sum += number;**
  + **}**

**return sum;**

* **Functional Programming: Uses stateless streams.**
  + **Why is this good? No shared data → Easy to run in parallel.**
  + ***//Functional***

**return numbers.*stream*().*reduce*(0, Integer::*sum*);**

* **How does Java parallelize streams?**
  + **parallel() tells Java to use multiple cores.**
  + **Java splits the stream into smaller parts.**
  + **Each part runs on a different core.**
  + **Finally, Java combines all results.**
  + ***// Sequential***
  + **long sum = LongStream.*range*(0, 1000000000L)**
  + **.*sum*();**
  + ***// Parallel on Existing Streams***
  + **long parallelSum = LongStream.*range*(0, 1000000000L)**
  + **.*parallel*()**

**.*sum*();**

**How Does It Work?**

* **Creates a Stream of Longs → LongStream.range(0, 1000000000L) generates numbers from 0 to 999,999,999.**
* **Enables Parallel Processing → .parallel() splits the range into chunks and processes them across multiple cores.**
* **Calculates the Sum Efficiently → .sum() combines results from all threads.**

**parallel() vs parallelStream()**

| **Feature** | **parallel()** | **parallelStream()** |
| --- | --- | --- |
| **Applied On** | **Any existing stream** | **Collections (List, Set, etc.)** |
| **Purpose** | **Converts a sequential stream to parallel** | **Directly creates a parallel stream** |
| **When to Use?** | **When you already have a stream** | **When starting from a collection** |
| **Example** | **list.stream().parallel()** | **list.parallelStream()** |

**Example - Using parallel()**

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

**numbers.*stream*()**

**.*parallel*()**

**.*forEach*(System.out::*println*);**

**Example - Using parallelStream()**

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

**numbers.*parallelStream*()**

**.*forEach*(System.out::*println*);**

**In what ways does Functional Programming simplify Java code?**[**#**](https://interview.in28minutes.com/interview-guides/java/functional-programming-in-java/#in-what-ways-does-functional-programming-simplify-java-code%3F)

**1. Creating a Thread**

* **Before FP:**
* ***// ❌ Traditional way (Verbose)***
* **Thread thread1 = new Thread(new Runnable() {**
* **@Override**
* **public void *run*() {**
* **System.out.*println*("Thread running...");**
* **}**
* **});**

**thread1.*start*();**

* **With FP (Lambda Expression): Shorter and more readable.**
* ***// ✅ Functional way (Lambda)***
* **Thread thread2 = new Thread(**
* **() -> System.out.*println*("Thread running..."));**

**thread2.*start*();**

**2. Creating a Comparator**

* **Before FP:**
* **import java.util.\*;**
* **List<String> names = Arrays.*asList*("John", "Alice", "Bob");**
* ***// ❌ Traditional way***
* **Collections.*sort*(names, new Comparator<String>() {**
* **@Override**
* **public int *compare*(String s1, String s2) {**
* **return s1.*compareTo*(s2);**
* **}**

**});**

* **With FP: Uses a lambda function, reducing code complexity.**
* ***// ✅ Functional way (Lambda)***

**names.*sort*((s1, s2) -> s1.*compareTo*(s2));**

**3. Listing Files in a Directory**

* **Before FP: Uses a for-loop.**
* **import java.nio.file.\*;**
* **Path path = Paths.*get*(".");**
* ***// ❌ Traditional way***
* **File[] files = new File(".").*listFiles*();**
* **for (File file : files) {**
* **System.out.*println*(file.*getName*());**

**}**

* **With FP: Uses Files.list() with method reference.**
* ***// ✅ Functional way (Streams)***

**Files.*list*(path).*forEach*(System.out::*println*);**

**4. Filtering & Transforming a List**

* **Before FP: Uses loops and conditionals.**
* **List<String> words = Arrays.*asList*("apple", "banana", "cherry");**
* ***// ❌ Traditional way***
* **List<String> ucWords = new ArrayList<>();**
* **for (String word : words) {**
* **if (word.*length*() > 5) {**
* **uppercaseWords.*add*(word.*toUpperCase*());**
* **}**

**}**

* **With FP: Uses stream().filter().map().**
* ***// ✅ Functional way (Streams)***
* **List<String> ucWordsFp = words.*stream*()**
* **.*filter*(word -> word.*length*() > 5)**
* **.*map*(String::*toUpperCase*)**

**.*toList*();**

**How do Higher-Order Functions, Behavior Parameterization, and First-Class Functions differ?**[**#**](https://interview.in28minutes.com/interview-guides/java/functional-programming-in-java/#how-do-higher-order-functions%2C-behavior-parameterization%2C-and-first-class-functions-differ%3F)

**To be frank, they are all very similar concepts!**

**Quick Comparison**

| **Concept** | **What It Is?** |
| --- | --- |
| **Higher-Order Functions** | **Function that returns or takes a function as a parameter** |
| **Behavior Parameterization** | **Passing behavior (a function) dynamically (as a parameter)** |
| **Functions as First-Class Citizens** | **Treats functions like values (Store a function in a variable, Pass a function to a method, Return a function)** |

**Example: Returning a Function**

* **This function returns a predicate based on a cutoff value.**
* **This allows dynamic filtering without hardcoding conditions.**
* **The returned predicate is then applied to a stream.**
* **public static Predicate<Course>**
* ***createPredicate*(int cutoff) {**
* **return course ->**
* **course.*getReviewScore*() > cutoff;**
* **}**
* ***// Usage***
* **Predicate<Course> highScore = *createPredicate*(95);**
* **List<Course> filtered = courses.*stream*()**
* **.*filter*(highScore)**

**.*collect*(Collectors.*toList*());**

**Example: Passing a Function**

* **This method accepts a predicate as an argument, allowing flexible filtering logic.**
* **Different conditions can be applied without modifying the method, making the code more reusable.**
* **public static void *filterAndPrint*(**
* **List<Integer> numbers,**
* **Predicate<Integer> predicate) {**
* **numbers.*stream*()**
* **.*filter*(predicate)**
* **.*forEach*(System.out::*println*);**
* **}**
* ***// Usage***
* ***filterAndPrint*(numbers, x -> x % 2 == 0);**
* ***filterAndPrint*(numbers, x -> x % 2 != 0);**

***filterAndPrint*(numbers, x -> x % 3 == 0);**

**What are the benefits of using Functional Programming?**[**#**](https://interview.in28minutes.com/interview-guides/java/functional-programming-in-java/#what-are-the-benefits-of-using-functional-programming%3F)

**What?**

* **Computation is treated as the evaluation of mathematical functions**
* **List<String> transformedNumbers = numbers.*stream*()**
* **.*map*(x -> x \* x) *// Step 1: Square each number***
* **.*map*(x -> x + 10) *// Step 2: Add 10***
* **.*map*(x -> "Value: " + x) *// Step 3: Convert***

**.*toList*(); *// Collect the result***

* **Core idea: focus on "what to do" rather than "how to do it."**

**Key Principles:**

* **First-Class Functions: Functions are treated as values, meaning they can be passed as arguments, returned from other functions, or assigned to variables.**
* **Immutability: Avoids modifying variables or data.**
* **No Side Effects: Functions do NOT affect other parts of the program (e.g., modifying global variables).**

**Why Use Functional Programming?**

* **Declarative Style → Focuses on what needs to be done, not how, making code cleaner and easier to read.**
* **Immutability → Data is not modified, reducing unexpected side effects and making debugging easier.**
* **Concise Code → Less boilerplate compared to traditional loops and conditionals.**
* **Thread Safety → No shared state means safer parallel execution without race conditions.**
* **Parallel Processing → Works well with streams and multi-core processors, improving performance.**

**LOGICAL:-**

**# Exercise 1 Print Only Odd Numbers from the List**

**# Exercise 2 Print All Courses individually**

**`List<String> courses = List.of("Spring", "Spring Boot", "API" , "Microservices", "AWS", "PCF","Azure", "Docker", "Kubernetes")`**

**# Exercise 3 Print Courses Containing the word "Spring"**

**# Exercise 4 Print Courses Whose Name has atleast 4 letters**

**# Exercise 5 Print the cubes of odd numbers**

**# Exercise 6 Print the number of characters in each course name**

**# Exercise 7 Square every number in a list and find the sum of squares**

**# Exercise 8 Cube every number in a list and find the sum of cubes**

**# Exercise 9 Find Sum of Odd Numbers in a list**

**# Exercise 10 Create a List with Even Numbers Filtered from the Numbers List**

**# Exercise 11 Create a List with lengths of all course titles.**

**# Exercise 12 Find Functional Interface behind the second argument of reduce method. Create an implementation for the Functional Interface.**

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

**# Exercise 13 Do Behavior Parameterization for the mapping logic.**

**List<Integer> squaredNumbers = numbers.stream() .map(x -> x\*x).collect(Collectors.toList());**

System.out.println("Ranga")

List<Integer> numbers = List.of(12, 9, 13, 4, 6, 2, 4, 12, 15);

numbers.stream().reduce(0, (x,y)->x+y)

numbers.stream().reduce(0, (x,y)->x)

numbers.stream().reduce(0, (x,y)->y)

numbers.stream().reduce(0, (x,y)-> x>y ? x:y)

numbers.stream().reduce(Integer.MIN\_VALUE, (x,y)-> x>y ? x:y)

numbers.stream().reduce(Integer.MIN\_VALUE, (x,y)-> x>y ? y:x)

numbers.stream().reduce(Integer.MAX\_VALUE, (x,y)-> x>y ? y:x)

numbers

numbers

numbers.stream().reduce(0, (x,y) -> x\*x + y\*y)

numbers.stream().map(x -> x\*x).reduce(0, Integer::sum)

numbers.stream().map(x -> x\*x\*x).reduce(0, Integer::sum)

numbers.stream().filter(x -> x%2==1).reduce(0, Integer::sum)

numbers.stream().filter(x -> x%2==0).reduce(0, Integer::sum)

numbers.stream().distinct().forEach(System.out::println)

numbers

numbers.stream().sorted().forEach(System.out::println)

numbers.stream().distinct().sorted().forEach(System.out::println)

List<String> courses = List.of("Spring", "Spring Boot", "API" , "Microservices","AWS", "PCF","Azure", "Docker", "Kubernetes");

courses

courses.stream().sorted().forEach(System.out::println)

courses

courses.stream().sorted().forEach(System.out::println)

courses.stream().sorted(Comparator.naturalOrder()).forEach(System.out::println)

courses.stream().sorted(Comparator.reverseOrder()).forEach(System.out::println)

courses.stream().sorted(Comparator.comparing(str -> str.length())).forEach(System.out::println)

numbers

courses

courses.stream().map(x -> x.length()).collect(Collectors.toList())

numbers.stream().map(x -> x\*x).collect(Collectors.toList())

Supplier<String> supplier = () -> {return "Ranga";};

Consumer<String> consumer = (str) -> { System.out.println(str);System.out.println(str);};

numbers.stream()

List<Integer> numbers = List.of(12, 9, 13, 4, 6, 2, 4, 12, 15);

numbers.stream()

Stream.of(12, 9, 13, 4, 6, 2, 4, 12, 15).count()

Stream.of(12, 9, 13, 4, 6, 2, 4, 12, 15).reduce(0, Integer::sum)

Stream.of(12, 9, 13, 4, 6, 2, 4, 12, 15)

int[] numberArray = {12, 9, 13, 4, 6, 2, 4, 12, 15};

Arrays.stream(numberArray)

Arrays.stream(numberArray).sum()

Arrays.stream(numberArray).average()

Arrays.stream(numberArray).min()

Arrays.stream(numberArray).max()

IntStream.range(1,10)

IntStream.range(1,10).sum()

IntStream.rangeClosed(1,10).sum()

IntStream.iterate(1, e -> e + 2).limit(10).sum()

IntStream.iterate(1, e -> e + 2).limit(10).peek(System.out::println).sum()

IntStream.iterate(2, e -> e + 2).limit(10).peek(System.out::println).sum()

IntStream.iterate(2, e -> e \* 2).limit(10).peek(System.out::println).sum()

IntStream.iterate(2, e -> e \* 2).limit(10).boxed().collect(Collectors.toList())

Integer.MAX\_VALUE

Long.MAX\_VALUE

IntStream.rangeClosed(1,50).reduce(1, (x,y)->x\*y)

LongStream.rangeClosed(1,50).reduce(1, (x,y)->x\*y)

LongStream.rangeClosed(1,50).reduce(1L, (x,y)->x\*y)

LongStream.rangeClosed(1,10).reduce(1, (x,y)->x\*y)

LongStream.rangeClosed(1,20).reduce(1, (x,y)->x\*y)

LongStream.rangeClosed(1,40).reduce(1, (x,y)->x\*y)

LongStream.rangeClosed(1,50).mapToObj(BigInteger::valueOf).reduce(BigInteger.ONE, BigInteger::multiply)

courses.stream().collect(Collectors.joining(" "))

courses.stream().collect(Collectors.joining(","))

"Spring".split("")

courses.stream().map(course -> course.split("")).collect(Collectors.toList())

courses.stream().map(course -> course.split(""))

courses.stream().map(course -> course.split("")).flatMap(Arrays::stream).collect(Collectors.toList())

courses.stream().map(course -> course.split("")).flatMap(Arrays::stream).distinct().collect(Collectors.toList())

List<String> courses = List.of("Spring", "Spring Boot", "API" , "Microservices","AWS", "PCF","Azure", "Docker", "Kubernetes");

List<String> courses2 = List.of("Spring", "Spring Boot", "API" , "Microservices","AWS", "PCF","Azure", "Docker", "Kubernetes");

courses.stream().flatMap(course -> courses2.stream().map(course2 -> List.of(course,course2))).collect(Collectors.toList())

courses.stream().flatMap(course -> courses2.stream().map(course2 -> List.of(course,course2))).filter(list -> list.get(0).equals(list.get(1))).collect(Collectors.toList())

courses.stream().flatMap(course -> courses2.stream().map(course2 -> List.of(course,course2))).filter(list -> !list.get(0).equals(list.get(1))).collect(Collectors.toList())

courses.stream().flatMap(course -> courses2.stream().filter(course2 -> course2.length()==course.length()).map(course2 -> List.of(course,course2))).filter(list -> !list.get(0).equals(list.get(1))).collect(Collectors.toList())

courses.stream().filter(courses -> courses.length()>11).map(String::toUpperCase).findFirst()

courses.stream().peek(System.out::println).filter(courses -> courses.length()>11).map(String::toUpperCase).peek(System.out::println).findFirst()

courses.stream().peek(System.out::println).filter(courses -> courses.length()>11).map(String::toUpperCase).peek(System.out::println)

$4.findFirst()

List<String> courses = List.of("Spring", "Spring Boot", "API" , "Microservices","AWS", "PCF","Azure", "Docker", "Kubernetes");

courses.replaceAll( str -> str.toUpperCase())

List<String> modifyableCourses = new ArrayList(courses);

modifyableCourses.replaceAll(str -> str.toUpperCase())

modifyableCourses

modifyableCourses.removeIf(course -> course.length()<6)

modifyableCourses

**PROGRAMS:-**

package programming;

import java.util.List;

public class FP01Exercises {

public static void main(String[] args) {

List<Integer> numbers = List.of(12, 9, 13, 4, 6, 2, 4, 12, 15);

printCubesOfOddNumbersInListFunctional(numbers);

// printOddNumbersInListFunctional(numbers);

List<String> courses = List.of("Spring", "Spring Boot", "API" , "Microservices","AWS", "PCF","Azure", "Docker", "Kubernetes");

// courses.stream()

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

// courses.stream()

// .filter(course -> course.contains("Spring"))

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

// courses.stream()

// .filter(course -> course.length() >= 4)

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

courses.stream()

.map(course -> course + " " + course.length())

.forEach(System.out::println);

}

private static void printOddNumbersInListFunctional(List<Integer> numbers) {

numbers.stream() // Convert to Stream

.filter(number -> number % 2 != 0) // Lamdba Expression

.forEach(System.out::println);// Method Reference

}

private static void printCubesOfOddNumbersInListFunctional(List<Integer> numbers) {

numbers.stream() // Convert to Stream

.filter(number -> number % 2 != 0) // Lamdba Expression

.map (number -> number \* number \* number)

.forEach(System.out::println);// Method Reference

}

}

2) package programming;

import java.util.List;

public class FP01Functional {

public static void main(String[] args) {

List<Integer> numbers = List.of(12, 9, 13, 4, 6, 2, 4, 12, 15);

// printAllNumbersInListFunctional(numbers);

//printEvenNumbersInListFunctional(numbers);

printSquaresOfEvenNumbersInListFunctional(numbers);

}

// private static void print(int number) {

// System.out.println(number);

// }

// private static boolean isEven(int number) {

// return number%2 == 0;

// }

private static void printAllNumbersInListFunctional(List<Integer> numbers) {

// What to do?

numbers.stream().forEach(System.out::println);// Method Reference

}

// number -> number%2 == 0

private static void printEvenNumbersInListFunctional(List<Integer> numbers) {

// What to do?

numbers.stream() // Convert to Stream

.filter(number -> number % 2 == 0) // Lamdba Expression

.forEach(System.out::println);// Method Reference

// .filter(FP01Functional::isEven)//Filter - Only Allow Even Numbers

}

private static void printSquaresOfEvenNumbersInListFunctional(List<Integer> numbers) {

numbers.stream() // Convert to Stream

.filter(number -> number % 2 == 0) // Lamdba Expression

//mapping - x -> x \* x

.map(number -> number \* number)

.forEach(System.out::println);// Method Reference

// .filter(FP01Functional::isEven)//Filter - Only Allow Even Numbers

}

}

3) package programming;

import java.util.List;

public class FP01Structured {

public static void main(String[] args) {

List<Integer> numbers = List.of(12, 9, 13, 4, 6, 2, 4, 12, 15);

//printAllNumbersInListStructured(numbers);

printEvenNumbersInListStructured(numbers);

}

private static void printAllNumbersInListStructured(List<Integer> numbers) {

// How to loop the numbers?

for (int number : numbers) {

System.out.println(number);

}

}

private static void printEvenNumbersInListStructured(List<Integer> numbers) {

// How to loop the numbers?

for (int number : numbers) {

if (number % 2 == 0) {

System.out.println(number);

}

}

}

}

4) package programming;

import java.util.List;

import java.util.stream.Collectors;

public class FP02Functional {

public static void main(String[] args) {

List<Integer> numbers = List.of(12, 9, 13, 4, 6, 2, 4, 12, 15);

List<Integer> squaredNumbers = squareList(numbers);

List<Integer> evenNumbersOnly = numbers.stream()

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

.collect(Collectors.toList());

//System.out.println(squaredNumbers);

System.out.println(evenNumbersOnly);

// 0 12

// 12 9

// 21 13

// 34 4

// 38 6

// 44 2

// 46 4

// 50 12

// 62 15

// 77

// int sum = addListFunctional(numbers);

//

// System.out.println(sum);

}

private static List<Integer> squareList(List<Integer> numbers) {

//1 , 5, 6

//1 -> 1

//5 -> 25

//6 -> 36

return numbers.stream()

.map(number -> number \* number)

.collect(Collectors.toList());

}

private static int sum(int aggregate, int nextNumber) {

System.out.println(aggregate + " " + nextNumber);

return aggregate + nextNumber;

}

private static int addListFunctional(List<Integer> numbers) {

//Stream of number -> One result value

//Combine them into one result => One Value

// 0 and FP02Functional::sum

return numbers.stream()

.parallel()

//.reduce(0, FP02Functional::sum);

// .reduce(0, (x,y) -> x + y);

.reduce(0, Integer::sum);

}

}

5) package programming;

import java.util.Comparator;

import java.util.List;

import java.util.stream.Collectors;

public class FP02StreamOperations {

@SuppressWarnings("unused")

public static void main(String[] args) {

List<Integer> numbers = List.of(12, 9, 13, 4, 6, 2, 4, 12, 15);

numbers.stream()

.distinct() //Stream<T> Intermediate

.sorted() //Stream<T>

.forEach(System.out::println); //void

List<Integer> squaredNumbers = numbers.stream()

.map(number -> number \* number) //Stream<R>

.collect(Collectors.toList()); //R

List<Integer> evenNumbersOnly = numbers.stream()

.filter(x -> x % 2 == 0) //Stream<T>

.collect(Collectors.toList());

int sum = numbers.stream()

.reduce(0, (x,y) -> x\*x + y\*y); //T

int greatest = numbers.stream()

.reduce(Integer.MIN\_VALUE, (x,y)-> x>y ? x:y);

List<String> courses = List.of("Spring", "Spring Boot", "API" , "Microservices","AWS", "PCF","Azure", "Docker", "Kubernetes");

List<String> coursesSortedByLengthOfCourseTitle = courses.stream()

.sorted(Comparator.comparing(str -> str.length()))

.collect(Collectors.toList());

}

}

6) package programming;

import java.util.List;

public class FP02Structured {

public static void main(String[] args) {

List<Integer> numbers = List.of(12, 9, 13, 4, 6, 2, 4, 12, 15);

int sum = addListStructured(numbers);

System.out.println(sum);

}

private static int addListStructured(List<Integer> numbers) {

//how to loop?

//how to store the sum?

int sum = 0;

for(int number:numbers) {

sum += number;

}

return sum;

}

}

7) package programming;

import java.util.List;

import java.util.function.Function;

import java.util.function.Predicate;

import java.util.stream.Collectors;

public class FP03BehaviorParameterization {

@SuppressWarnings("unused")

public static void main(String[] args) {

List<Integer> numbers = List.of(12, 9, 13, 4, 6, 2, 4, 12, 15);

//filterAndPrint(numbers, x -> x%2==0);

//filterAndPrint(numbers, x -> x%2!=0);

filterAndPrint(numbers, x -> x%3==0);

Function<Integer, Integer> mappingFunction = x -> x\*x;

List<Integer> squaredNumbers = mapAndCreateNewList(numbers, mappingFunction);

List<Integer> cubedNumbers = mapAndCreateNewList(numbers, x -> x\*x\*x);

List<Integer> doubledNumbers = mapAndCreateNewList(numbers, x -> x + x);

System.out.println(doubledNumbers);

}

private static List<Integer> mapAndCreateNewList(List<Integer> numbers,

Function<Integer, Integer> mappingFunction) {

return numbers.stream()

.map(mappingFunction)

.collect(Collectors.toList());

}

private static void filterAndPrint(List<Integer> numbers, Predicate<? super Integer> predicate) {

numbers.stream()

.filter(predicate)

.forEach(System.out::println);

}

}

8) package programming;

import java.util.List;

import java.util.function.BinaryOperator;

import java.util.function.Consumer;

import java.util.function.Function;

import java.util.function.Predicate;

import java.util.function.Supplier;

public class FP03FunctionalInterfaces {

/\*

boolean isEven(int x) {

return x%2==0;

}

int squared(int x) {

return x \* x;

}

\*/

@SuppressWarnings("unused")

public static void main(String[] args) {

List<Integer> numbers = List.of(12, 9, 13, 4, 6, 2, 4, 12, 15);

Predicate<Integer> isEvenPredicate = x -> x%2==0;

Predicate<Integer> isEvenPredicate2 = new Predicate<Integer>() {

@Override

public boolean test(Integer x) {

return x%2==0;

}

};

Function<Integer, Integer> squareFunction = x -> x \* x;

Function<Integer, Integer> squareFunction2 = new Function<Integer, Integer>() {

@Override

public Integer apply(Integer x) {

return x\*x;

}

};

Consumer<Integer> sysoutConsumer = System.out::println;

Consumer<Integer> sysoutConsumer2 = new Consumer<Integer>() {

public void accept(Integer x) {

System.out.println(x);

}

};

numbers.stream()

.filter(isEvenPredicate2)

.map(squareFunction2)

.forEach(sysoutConsumer2);

BinaryOperator<Integer> sumBinaryOperator = Integer::sum;

//BinaryOperator<Integer> sumBinaryOperator = (x,y) => x + y;

BinaryOperator<Integer> sumBinaryOperator2 = new BinaryOperator<Integer>() {

@Override

public Integer apply(Integer a, Integer b) {

// TODO Auto-generated method stub

return a + b;

}

};

int sum = numbers.stream()

.reduce(0, sumBinaryOperator);

}

}

9) package programming;

import java.util.List;

import java.util.Random;

import java.util.function.BiConsumer;

import java.util.function.BiFunction;

import java.util.function.BiPredicate;

import java.util.function.BinaryOperator;

import java.util.function.Consumer;

import java.util.function.Function;

import java.util.function.IntBinaryOperator;

import java.util.function.Predicate;

import java.util.function.Supplier;

import java.util.function.UnaryOperator;

public class FP03FunctionalInterfaces2 {

@SuppressWarnings("unused")

public static void main(String[] args) {

List<Integer> numbers = List.of(12, 9, 13, 4, 6, 2, 4, 12, 15);

Predicate<Integer> isEvenPredicate = (Integer x) -> x % 2 == 0;

Function<Integer, Integer> squareFunction = x -> x \* x;

Function<Integer, String> stringOutpuFunction = x -> x + " ";

Consumer<Integer> sysoutConsumer = x -> System.out.println(x);

BinaryOperator<Integer> sumBinaryOperator = (x, y) -> x + y;

//No input > Return Something

Supplier<Integer> randomIntegerSupplier = () -> {

Random random = new Random();

return random.nextInt(1000);

};

//System.out.println(randomIntegerSupplier.get());

UnaryOperator<Integer> unaryOperator = x -> 3 \* x;

System.out.println(unaryOperator.apply(10));

BiPredicate<Integer, String> biPredicate = (number,str) -> {

return number<10 && str.length()>5;

};

System.out.println(biPredicate.test(10, "in28minutes"));

BiFunction<Integer, String, String> biFunction = (number,str) -> {

return number + " " + str;

};

System.out.println(biFunction.apply(15, "in28minutes"));

BiConsumer<Integer, String> biConsumer = (s1,s2) -> {

System.out.println(s1);

System.out.println(s2);

};

biConsumer.accept(25, "in28Minutes");

BinaryOperator<Integer> sumBinaryOperator2 = (x, y) -> x + y;

IntBinaryOperator intBinaryOperator = (x,y) -> x + y;

//IntBinaryOperator

//IntConsumer

//IntFunction

//IntPredicate

//IntSupplier

//IntToDoubleFunction

//IntToLongFunction

//IntUnaryOperator

//Long, Double, Int

//numbers.stream().filter(isEvenPredicate).map(squareFunction).forEach(sysoutConsumer);

//int sum = numbers.stream().reduce(0, sumBinaryOperator);

}

}

10) package programming;

import java.util.List;

import java.util.Random;

import java.util.function.Supplier;

public class FP03LambdaExpressions {

@SuppressWarnings("unused")

public static void main(String[] args) {

List<String> courses = List.of("Spring", "Spring Boot", "API" , "Microservices","AWS", "PCF","Azure", "Docker", "Kubernetes");

courses.stream().sorted().forEach( (String x) -> System.out.println(x));

List<Integer> numbers = List.of(12, 9, 13, 4, 6, 2, 4, 12, 15);

Integer sum = numbers.stream().reduce(0, (Integer x,Integer y) -> x + y);

Supplier<Integer> randomSupplier = () -> {

// create instance of Random class

Random rand = new Random();

return rand.nextInt(1000);

};

System.out.println(randomSupplier.get());

//Playing with Return Statements

//Playing with Braces

}

}

11) package programming;

import java.util.List;

import java.util.function.Supplier;

public class FP03MethodReferences {

private static void print(String str) {

System.out.println(str);

}

@SuppressWarnings("unused")

public static void main(String[] args) {

List<String> courses = List.of("Spring", "Spring Boot", "API", "Microservices",

"AWS", "PCF", "Azure", "Docker",

"Kubernetes");

courses.stream()

//.map(str -> str.toUpperCase())

.map(String::toUpperCase)

.forEach(FP03MethodReferences::print);

Supplier<String> supplier = String::new;

}

}

12) package programming;

import java.util.Comparator;

import java.util.List;

import java.util.function.Predicate;

import java.util.stream.Collectors;

class Course {

private String name;

private String category;

private int reviewScore;

private int noOfStudents;

public Course(String name, String category, int reviewScore, int noOfStudents) {

super();

this.name = name;

this.category = category;

this.reviewScore = reviewScore;

this.noOfStudents = noOfStudents;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getCategory() {

return category;

}

public void setCategory(String category) {

this.category = category;

}

public int getReviewScore() {

return reviewScore;

}

public void setReviewScore(int reviewScore) {

this.reviewScore = reviewScore;

}

public int getNoOfStudents() {

return noOfStudents;

}

public void setNoOfStudents(int noOfStudents) {

this.noOfStudents = noOfStudents;

}

public String toString() {

return name + ":" + noOfStudents + ":" + reviewScore;

}

}

public class FP04CustomClass {

public static void main(String[] args) {

List<Course> courses = List.of(new Course("Spring", "Framework", 98, 20000),

new Course("Spring Boot", "Framework", 95, 18000), new Course("API", "Microservices", 97, 22000),

new Course("Microservices", "Microservices", 96, 25000),

new Course("FullStack", "FullStack", 91, 14000), new Course("AWS", "Cloud", 92, 21000),

new Course("Azure", "Cloud", 99, 21000), new Course("Docker", "Cloud", 92, 20000),

new Course("Kubernetes", "Cloud", 91, 20000));

// allMatch, noneMatch, anyMatch

Predicate<Course> reviewScoreGreaterThan95Predicate

= course -> course.getReviewScore() > 95;

Predicate<Course> reviewScoreGreaterThan90Predicate

= course -> course.getReviewScore() > 90;

Predicate<Course> reviewScoreLessThan90Predicate

= course -> course.getReviewScore() < 90;

System.out.println(courses.stream().allMatch(reviewScoreGreaterThan95Predicate));

System.out.println(courses.stream().noneMatch(reviewScoreLessThan90Predicate));

System.out.println(courses.stream().anyMatch(reviewScoreLessThan90Predicate));

System.out.println(courses.stream().anyMatch(reviewScoreGreaterThan95Predicate));

Comparator<Course> comparingByNoOfStudentsIncreasing

= Comparator.comparingInt(Course::getNoOfStudents);

System.out.println(

courses.stream()

.sorted(comparingByNoOfStudentsIncreasing)

.collect(Collectors.toList()));

//[FullStack:14000:91, Spring Boot:18000:95, Spring:20000:98, Docker:20000:92, Kubernetes:20000:91, AWS:21000:92, Azure:21000:99, API:22000:97, Microservices:25000:96]

Comparator<Course> comparingByNoOfStudentsDecreasing

= Comparator.comparingInt(Course::getNoOfStudents).reversed();

System.out.println(

courses.stream()

.sorted(comparingByNoOfStudentsDecreasing)

.collect(Collectors.toList()));

//[Microservices:25000:96, API:22000:97, AWS:21000:92, Azure:21000:99, Spring:20000:98, Docker:20000:92, Kubernetes:20000:91, Spring Boot:18000:95, FullStack:14000:91]

Comparator<Course> comparingByNoOfStudentsAndNoOfReviews

= Comparator.comparingInt(Course::getNoOfStudents)

.thenComparingInt(Course::getReviewScore)

.reversed();

System.out.println(

courses.stream()

.sorted(comparingByNoOfStudentsAndNoOfReviews)

.collect(Collectors.toList()));

//[Microservices:25000:96, API:22000:97, Azure:21000:99, AWS:21000:92, Spring:20000:98, Docker:20000:92, Kubernetes:20000:91, Spring Boot:18000:95, FullStack:14000:91]

System.out.println(

courses.stream()

.sorted(comparingByNoOfStudentsAndNoOfReviews)

.limit(5)

.collect(Collectors.toList()));

//[Microservices:25000:96, API:22000:97, Azure:21000:99, AWS:21000:92, Spring:20000:98]

System.out.println(

courses.stream()

.sorted(comparingByNoOfStudentsAndNoOfReviews)

.skip(3)

.collect(Collectors.toList()));

//[AWS:21000:92, Spring:20000:98, Docker:20000:92, Kubernetes:20000:91, Spring Boot:18000:95, FullStack:14000:91]

System.out.println(

courses.stream()

.sorted(comparingByNoOfStudentsAndNoOfReviews)

.skip(3)

.limit(5)

.collect(Collectors.toList()));

//[AWS:21000:92, Spring:20000:98, Docker:20000:92, Kubernetes:20000:91, Spring Boot:18000:95]

System.out.println(courses);

//[Spring:20000:98, Spring Boot:18000:95, API:22000:97, Microservices:25000:96, FullStack:14000:91, AWS:21000:92, Azure:21000:99, Docker:20000:92, Kubernetes:20000:91]

System.out.println(

courses.stream()

.takeWhile(course -> course.getReviewScore()>=95)

.collect(Collectors.toList()));

//[Spring:20000:98, Spring Boot:18000:95, API:22000:97, Microservices:25000:96]

System.out.println(

courses.stream()

.dropWhile(course -> course.getReviewScore()>=95)

.collect(Collectors.toList()));

//[FullStack:14000:91, AWS:21000:92, Azure:21000:99, Docker:20000:92, Kubernetes:20000:91]

System.out.println(

courses.stream()

.max(comparingByNoOfStudentsAndNoOfReviews));

//Optional[FullStack:14000:91]

System.out.println(

courses.stream()

.min(comparingByNoOfStudentsAndNoOfReviews)

.orElse(new Course("Kubernetes", "Cloud", 91, 20000))

);

//Optional[Microservices:25000:96]

//Microservices:25000:96

System.out.println(

courses.stream()

.filter(reviewScoreLessThan90Predicate)

.min(comparingByNoOfStudentsAndNoOfReviews)

.orElse(new Course("Kubernetes", "Cloud", 91, 20000))

);

//Optional.empty

//Kubernetes:20000:91

System.out.println(

courses.stream()

.filter(reviewScoreLessThan90Predicate)

.findFirst()

);//Optional.empty

System.out.println(

courses.stream()

.filter(reviewScoreGreaterThan95Predicate)

.findFirst()

);//Optional[Spring:20000:98]

System.out.println(

courses.stream()

.filter(reviewScoreGreaterThan95Predicate)

.findAny()

);//Optional[Spring:20000:98]

System.out.println(

courses.stream()

.filter(reviewScoreGreaterThan95Predicate)

.mapToInt(Course::getNoOfStudents)

.sum());//88000

System.out.println(

courses.stream()

.filter(reviewScoreGreaterThan95Predicate)

.mapToInt(Course::getNoOfStudents)

.average());//OptionalDouble[22000.0]

System.out.println(

courses.stream()

.filter(reviewScoreGreaterThan95Predicate)

.mapToInt(Course::getNoOfStudents)

.count());//4

System.out.println(

courses.stream()

.filter(reviewScoreGreaterThan95Predicate)

.mapToInt(Course::getNoOfStudents)

.max());//OptionalInt[25000]

System.out.println(

courses.stream()

.collect(Collectors.groupingBy(Course::getCategory)));

//{Cloud=[AWS:21000:92, Azure:21000:99, Docker:20000:92, Kubernetes:20000:91],

// FullStack=[FullStack:14000:91],

// Microservices=[API:22000:97, Microservices:25000:96],

// Framework=[Spring:20000:98, Spring Boot:18000:95]}

System.out.println(

courses.stream()

.collect(Collectors.groupingBy(Course::getCategory, Collectors.counting())));

//{Cloud=4, FullStack=1, Microservices=2, Framework=2}

System.out.println(

courses.stream()

.collect(Collectors.groupingBy(Course::getCategory,

Collectors.maxBy(Comparator.comparing(Course::getReviewScore)))));

//{Cloud=Optional[Azure:21000:99], FullStack=Optional[FullStack:14000:91], Microservices=Optional[API:22000:97], Framework=Optional[Spring:20000:98]}

System.out.println(

courses.stream()

.collect(Collectors.groupingBy(Course::getCategory,

Collectors.mapping(Course::getName, Collectors.toList()))));

//{Cloud=[AWS, Azure, Docker, Kubernetes], FullStack=[FullStack], Microservices=[API, Microservices], Framework=[Spring, Spring Boot]}

Predicate<Course> reviewScoreGreaterThan95Predicate2

= createPredicateWithCutoffReviewScore(95);

Predicate<Course> reviewScoreGreaterThan90Predicate2

= createPredicateWithCutoffReviewScore(90);

}

private static Predicate<Course> createPredicateWithCutoffReviewScore(int cutoffReviewScore) {

return course -> course.getReviewScore() > cutoffReviewScore;

}

}

13) package programming;

import java.io.IOException;

import java.nio.file.Files;

import java.nio.file.Paths;

import java.util.Arrays;

public class FP05Files {

public static void main(String[] args) throws IOException {

// Files.lines(Paths.get("file.txt"))

// .map(str -> str.split(" "))

// .flatMap(Arrays::stream)

// .distinct()

// .sorted()

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

Files.list(Paths.get("."))

.filter(Files::isDirectory)

.forEach(System.out::println);

}

}

14) package programming;

import java.util.stream.LongStream;

public class FP05Parallelizing {

public static void main(String[] args) {

long time = System.currentTimeMillis();

//0, 1000000000 869

//System.out.println(LongStream.range(0,1000000000).sum());//499999999500000000

//601

System.out.println(LongStream.range(0,1000000000).parallel().sum());

System.out.println(System.currentTimeMillis() - time);

}

}

15) package programming;

public class FP05Threads {

public static void main(String[] args) {

Runnable runnable = new Runnable() {

@Override

public void run() {

for (int i = 0; i < 10000; i++) {

System.out.println(Thread.currentThread().getId() + ":" + i);

}

}

};

Runnable runnable2 = () -> {

for (int i = 0; i < 10000; i++) {

System.out.println(Thread.currentThread().getId() + ":" + i);

}

};

Thread thread = new Thread(runnable2);

thread.start();

Thread thread1 = new Thread(runnable2);

thread1.start();

Thread thread2 = new Thread(runnable2);

thread2.start();

}

}

NOTES:-

## Table of Contents

### Introduction to the Course

- Step 01: Functional Programming with Java - Course Overview

- Step 02: Functional Programming with Java - Exporing Github Repo and Installations

### [Chapter 1 - Introduction to Functional Programming with Java](#introduction-to-functional-programming-with-java)

- Step 01: Getting Started with Functional Programming with Java

- Step 02: Writing Your First Java Functional Program

- Step 03: Improving Java Functional Program with filter

- Step 04: Using Lambda Expression to enhance your Functional Program

- Step 05: Do Functional Programming Exercises with Streams, Filters and Lambdas

- Step 06: Using map in Functional Programs - with Exercises

- Step 07: Quick Review of Functional Programming Basics

### [Chapter 2 - Playing with Streams](#playing-with-streams)

- Step 01: Learning Stream Operations - Calculate Sum using reduce

- Step 02: Playing with reduce

- Step 03: Exploring Streams with Puzzles in JShell

- Step 04: Do Functional Programming Exercises with Streams and reduce

- Step 05: Learn Stream Operations - distinct and sorted

- Step 06: Using Comparators to Sort Streams with sorted

- Step 07: Collecting Stream Elements to List using collect

- Step 08: Reviewing Streams - Intermediate and Stream Operations

### [Chapter 3 - Exploring Java Functional Interfaces and Lambdas](#exploring-java-functional-interfaces-and-lambdas)

- Step 01: Getting Started with Functional Interfaces - Predicate, Consumer and Function

- Step 02: Do Exercises with Functional Interfaces - BinaryOperator

- Step 03: Doing Behavior Parameterization with Functional Programming

- Step 04: Do Exercise with Behavior Parameterization

- Step 05: Exploring Supplier and UnaryOperator Functional Interfaces

- Step 06: Exploring BiPredicate, BiFunction, BiConsumer and Primitive Functional Interfaces

- Step 07: Playing Puzzles with Functional Interfaces and Lambdas

- Step 08: Exploring Method References with Java

### [Chapter 4 - Java Functional Programming with Custom Classes](#java-functional-programming-with-custom-classes)

- Step 01: Creating Custom Class Course with some Test Data

- Step 02: Playing with allMatch, noneMatch and anyMatch

- Step 03: Sorting courses with sorted and creating Comparators

- Step 04: Playing with skip, limit, takeWhile and dropWhile

- Step 05: Finding top, max and min courses with max, min, findFirst and findAny

- Step 06: Playing with sum, average and count

- Step 07: Grouping courses into Map using groupingBy

### [Chapter 5 - Playing with Java Functional Programming](#playing-with-java-functional-programming)

- Step 01: Creating Streams using Stream of method and for Arrays

- Step 02: Creating Streams for First 100 Numbers, Squares of Numbers and More

- Step 03: Doing Big Number calculations with BigInteger

### [Chapter 6 - Playing further with Java Functional Programming](#playing-further-with-java-functional-programming)

- Step 01: Joining Strings with joining and Playing with flapMap

- Step 02: Creating Higher Order Functions

- Step 03: FP and Performance - Intermediate Stream Operations are Lazy

- Step 04: Improving Performance with Parallelization of Streams

### [Chapter 7 - Functional Programming makes Java Easy](#functional-programming-makes-java-easy)

- Step 01: Modifying lists with replaceAll and removeIf

- Step 02: Playing with Files using Functional Programming

- Step 03: Playing with Threads using Functional Programming

- Step 04: Using Functional Programming in Java Applications

## Introduction to Functional Programming with Java

### Step 01: Getting Started with Functional Programming with Java

- Functional programming is introduced as a different approach from traditional programming.

### Key Objectives

- To understand how functional programming differs from traditional programming.

- Introduction to basic functional programming concepts:

- Streams

- Lambda Expressions

- Method References

### Hands-On Approach

- The course adopts a hands-on approach, starting with the creation of a new Java project in Eclipse suitable for functional programming.

### Setting Up the Java Project

1. \*\*Open Eclipse\*\* and select or create a new workspace.

2. \*\*Create a New Java Project:\*\*

- Navigate to `File > New > Java Project`.

- Name the project `functional-programming-with-java`.

- Ensure the Java version is set to Java 9 or above (Java 9, 10, 11, 12 are compatible).

### Writing the First Code:

### Problem Statement

- \*\*Objective:\*\* Print a set of numbers, each on a separate line.

### Steps to Solve the Problem

1. \*\*Create a New Java Class:\*\*

- Go to `File > New > Class`.

- Name the class `FP01Structured`.

- Package name: `programming`.

- Include a `main` method by checking the appropriate box.

```java

public class FP01Structured {

public static void main(String[] args) {

printAllNumbersInListStructured(List.of(12, 9, 13, 4, 6, 2, 4, 12, 15));

}

}

2. \*\*Method to Print Numbers:\*\*

- Define a method printAllNumbersInListStructured(List<Integer> numbers).

```java

private static void printAllNumbersInListStructured(List<Integer> numbers) {

for (int number : numbers) {

System.out.println(number);

}

}

3. \*\*Run the Program:\*\*

- Right-click on the file in Eclipse.

- Select Run As > Java Application.

- Check the console for output.

- The traditional approach focuses on "how" to achieve tasks, like looping through numbers.

---

### Step 02: Writing Your First Java Functional Program

### Key Objectives:

- Transition from a traditional programming approach to a functional approach in Java.

- Learn how to print a list of numbers using \*\*streams\*\* and \*\*method references\*\* in functional programming.

- Understand the functional programming focus on \*\*"what to do"\*\* rather than \*\*"how to do it"\*\*.

### Problem Statement:

\*\*Objective\*\*: Print a set of numbers from a list using a \*\*functional programming approach\*\*.

We previously solved this problem using a structured loop. Now, we will refactor the solution using functional programming.

### 1. Refactoring the Code to Use Streams

- In functional programming, we focus on \*\*"what to do"\*\* rather than \*\*"how to do it"\*\*.

- Instead of explicitly looping through each element in the list, we can leverage \*\*streams\*\*.

### Refactoring the Method:

```java

private static void printAllNumbersInList(List<Integer> numbers) {

numbers.stream()

.forEach(FP01Functional::print); // Method reference

}

```

- `numbers.stream()`: Converts the list of numbers into a stream, which is a sequence of elements.

- `.forEach()`: Applies the specified action (in this case, printing) to each element in the stream.

- `FP01Functional::print`: This is a method reference to the print method, which will be used to print each number.

### 2. Creating the print Method

- The print method is a simple helper method that prints each number passed to it.

```java

private static void print(int number) {

System.out.println(number);

}

```

- This method takes an integer and prints it to the console using `System.out.println()`.

- The method is referenced inside the stream's `forEach()` method to print each number from the list.

### Understanding Key Functional Programming Concepts

\*\*Streams:\*\* A stream is a sequence of elements that supports various operations to process data in a declarative way. In this case, we convert the list of numbers into a stream to process each element without manually looping.

\*\*Method References (::):\*\* Instead of passing a full lambda expression or explicitly calling a method inside `forEach()`, we use a method reference (`FP01Functional::print`). This is a shorthand for telling Java to execute the print method for each element in the stream.

#### When the program is run, it will print each number in the list as output.

#### Full Code Example:

```java

public class FP01Functional {

public static void main(String[] args) {

List<Integer> numbers = List.of(12, 9, 13, 4, 6, 2, 4, 12, 15);

printAllNumbersInList(numbers);

}

private static void printAllNumbersInList(List<Integer> numbers) {

numbers.stream() // Convert the list to a stream

.forEach(FP01Functional::print); // Print each number using method reference

}

private static void print(int number) {

System.out.println(number); // Print the number

}

}

```

#### Output:

```

12

9

13

4

6

2

4

12

15

```

---

### Step 03: Improving Java Functional Program with filter

### Key Objectives

- Understand how to simplify the existing functional code.

- Learn how to filter data in a functional programming approach using \*\*streams\*\*.

- Compare the structured approach and functional approach for filtering even numbers from a list.

### Problem Statement:

\*\*Objective\*\*: Print only the even numbers from a list of numbers using both \*\*structured\*\* and \*\*functional programming approaches\*\*.

We will improve upon the functional program by adding a \*\*filter\*\* to select only even numbers.

### 1. Simplifying the Existing Functional Code

- In the previous lecture, we used a custom method (`print`) to print each number in the list.

- We can simplify this approach further by directly using `System.out::println`.

### Updated Functional Code:

```java

numbers.stream()

.forEach(System.out::println); // Simplified method reference

```

- Instead of defining a separate `print` method, we can use Java’s built-in method reference `System.out::println`.

- This reduces complexity and makes the code more concise.

### 2. Problem - Printing Even Numbers in Structured Approach

- Let's go back to the structured approach in the `FP01Structured` class and modify it to print only even numbers.

#### Structured Code:

```java

private static void printEvenNumbersInListStructured(List<Integer> numbers) {

for (int number : numbers) {

if (number % 2 == 0) {

System.out.println(number);

}

}

}

```

- The condition `number % 2 == 0` checks whether a number is even by dividing the number by 2 and checking if the remainder is 0.

- If the number is even, it gets printed.

- When this structured approach runs, the output will only display even numbers from the list.

### 3. Printing Even Numbers in the Functional Approach

- Now, let's implement the same functionality using a functional approach with streams.

#### Refactoring to Use a Filter:

```java

private static void printEvenNumbersInListFunctional(List<Integer> numbers) {

numbers.stream() // Convert the list to a stream

.filter(FP01Functional::isEven) // Filter to select even numbers

.forEach(System.out::println); // Print each even number

}

```

- `numbers.stream()`: This converts the list of numbers into a stream.

- `filter(FP01Functional::isEven)`: This applies a filter to the stream, only allowing numbers that pass the isEven condition.

- `forEach(System.out::println)`: After filtering, the remaining even numbers are printed.

### 4. Creating the `isEven` Method

- We need a method to check if a number is even, which will be used in the `filter()` function.

#### Method to Check Even Numbers:

```java

private static boolean isEven(int number) {

return number % 2 == 0;

}

```

- The method checks if a number is divisible by 2 without a remainder.

- If the condition is true, it returns `true`, meaning the number is even.

### Method Reference:

In the functional approach, we pass the method reference `FP01Functional::isEven` into the `filter()` method, allowing the stream to filter numbers based on this condition.

### 5. Full Functional Code

```java

public class FP01Functional {

public static void main(String[] args) {

List<Integer> numbers = List.of(12, 9, 13, 4, 6, 2, 4, 12, 15);

printEvenNumbersInListFunctional(numbers);

}

private static void printEvenNumbersInListFunctional(List<Integer> numbers) {

numbers.stream() // Convert the list to a stream

.filter(FP01Functional::isEven) // Filter only even numbers

.forEach(System.out::println); // Print each number

}

private static boolean isEven(int number) {

return number % 2 == 0; // Check if the number is even

}

}

```

#### Output:

```

12

4

6

2

4

12

```

- The list is filtered to contain only even numbers, which are then printed.

### Structured vs Functional Approach:

#### Structured Approach:

- We used a traditional `for` loop and an `if` statement to manually check each number and print the even numbers.

#### Functional Approach:

- We used a stream, applied a filter to select only even numbers, and then printed them using `forEach()` with a method reference to `System.out::println`.

---

### Step 04: Using Lambda Expression to enhance your Functional Program

### Key Objectives

- Learn about \*\*Lambda Expressions\*\* and how they simplify functional programming in Java.

- Replace the method reference (`isEven`) with a \*\*Lambda Expression\*\*.

- Understand how Lambda Expressions allow you to define logic directly within functional constructs.

### Introduction to Lambda Expressions

- In the previous step, we used a separate method (`isEven`) to filter even numbers from the list.

- While this works, it introduces unnecessary complexity since the logic is very simple.

- In this step, we’ll learn how to simplify this process using a \*\*Lambda Expression\*\*, a feature in Java that allows you to write the logic of a method in a more concise way, without needing to define a separate method.

### 1. Refactoring the `filter` to Use a Lambda Expression

- Instead of using a method reference like `FP01Functional::isEven`, we will define the logic for filtering even numbers \*\*directly inside\*\* the `filter()` method using a Lambda Expression.

### Lambda Expression Syntax:

```java

numbers.stream()

.filter(number -> number % 2 == 0) // Lambda Expression

.forEach(System.out::println); // Print the even numbers

```

- `number ->`: This is the Lambda Expression syntax. It means we are passing number as an argument to the function.

- `number % 2 == 0`: The logic checks whether the number is even (i.e., divisible by 2).

- `filter()`: Filters out only the even numbers.

- `forEach(System.out::println)`: Prints the filtered numbers.

- Instead of defining a separate method (isEven), we now define the entire logic inside the `filter()` method.

- The expression `number -> number % 2 == 0` takes each number, checks if it is even, and returns true or false.

- If the number is even (true), it passes through the filter and gets printed. If not, it is ignored.

### 2. Replacing the `isEven` Method

- Previously, we had a method called `isEven` that checked whether a number was divisible by 2.

- Now, this method is no longer needed, as the logic is directly handled by the Lambda Expression.

### Old Code with isEven Method:

```java

private static boolean isEven(int number) {

return number % 2 == 0;

}

```

- This method can now be commented out or removed, as the Lambda Expression handles the same logic with fewer lines of code.

### 3. Full Functional Code with Lambda Expression

- Here is the full code that uses a Lambda Expression instead of a method reference:

```java

public class FP01Functional {

public static void main(String[] args) {

List<Integer> numbers = List.of(12, 9, 13, 4, 6, 2, 4, 12, 15);

printEvenNumbersInListFunctional(numbers);

}

private static void printEvenNumbersInListFunctional(List<Integer> numbers) {

numbers.stream() // Convert the list to a stream

.filter(number -> number % 2 == 0) // Lambda Expression to filter even numbers

.forEach(System.out::println); // Print each even number

}

}

```

#### Output:

```

12

4

6

2

4

12

```

- The Lambda Expression `number -> number % 2 == 0` is used to filter even numbers.

- This makes the code more concise and eliminates the need for an external method like `isEven`.

### Method Reference vs Lambda Expression

#### Method Reference (`FP01Functional::isEven`):

- Refers to an existing method defined elsewhere. This is useful when you want to reuse logic defined in a separate method.

#### Lambda Expression (`number -> number % 2 == 0`):

- Defines the logic inline, making the code shorter and eliminating the need for separate methods.

---

### Step 05: Do Functional Programming Exercises with Streams, Filters and Lambdas

### Key Objectives

- Consolidate functional programming concepts learned so far through a series of exercises.

- Practice using \*\*streams\*\*, \*\*filters\*\*, and \*\*lambda expressions\*\* to solve problems in a functional style.

### Exercises Overview

In this lesson, we will perform the following exercises using functional programming techniques:

1. Print only the \*\*odd numbers\*\* from a list of integers.

2. Print all \*\*courses\*\* individually from a list of course names.

3. Print only the courses that \*\*contain the word Spring\*\*.

4. Print courses whose names have \*\*at least four letters\*\*.

### Exercise 1 - Printing Odd Numbers from the List

- The first exercise is to modify the method that prints even numbers to print \*\*odd numbers\*\* instead.

### Refactor the Code:

- We can use a similar logic as for even numbers, but this time we will check if a number is \*\*odd\*\* by modifying the condition.

#### Code:

```java

private static void printOddNumbersInListFunctional(List<Integer> numbers) {

numbers.stream()

.filter(number -> number % 2 != 0) // Filter odd numbers

.forEach(System.out::println); // Print each odd number

}

```

- The `filter(number -> number % 2 != 0)` lambda expression checks if the number is odd.

- If the number is odd, it gets printed using `forEach(System.out::println)`.

#### Output:

```

9

13

15

```

- The output contains only the odd numbers from the list.

### Exercise 2 - Printing All Courses Individually

- In this exercise, we will work with a list of course names instead of numbers. The task is to print each course name on its own line.

#### Code:

```java

List<String> courses = List.of("Spring", "Spring Boot", "API", "Microservices",

"AWS", "PCF", "Azure", "Docker", "Kubernetes");

courses.stream()

.forEach(System.out::println); // Print each course name

```

- We convert the list of courses to a stream using `courses.stream()`.

- We print each course using `forEach(System.out::println)`.

#### Output:

```

Spring

Spring Boot

API

Microservices

AWS

PCF

Azure

Docker

Kubernetes

```

### Exercise 3 - Printing Courses That Contain the Word "Spring"

- For this exercise, the goal is to filter the list of courses and print only those that contain the word Spring.

#### Code:

```java

courses.stream()

.filter(course -> course.contains("Spring")) // Filter courses containing "Spring"

.forEach(System.out::println); // Print each filtered course

```

- The `filter(course -> course.contains("Spring"))` lambda expression checks if a course name contains the word "Spring".

- Only the courses that pass this condition are printed.

#### Output:

```

Spring

Spring Boot

```

### Exercise 4 - Printing Courses with at Least Four Letters

- In this final exercise, we need to filter the courses so that only those with four or more letters are printed.

#### Code:

```java

courses.stream()

.filter(course -> course.length() >= 4) // Filter courses with at least 4 characters

.forEach(System.out::println); // Print each filtered course

```

- The `filter(course -> course.length() >= 4)` lambda expression checks if the length of the course name is at least four.

- Only the courses that have names of four or more characters are printed.

#### Output:

```

Spring

Spring Boot

API

Microservices

Azure

Docker

Kubernetes

```

#### Full Code:

- Here’s the full code for all four exercises:

```java

public class FP01Exercises {

public static void main(String[] args) {

List<Integer> numbers = List.of(12, 9, 13, 4, 6, 2, 4, 12, 15);

printOddNumbersInListFunctional(numbers);

List<String> courses = List.of("Spring", "Spring Boot", "API", "Microservices",

"AWS", "PCF", "Azure", "Docker", "Kubernetes");

// Exercise 2: Print all courses

courses.stream()

.forEach(System.out::println);

// Exercise 3: Print courses that contain the word "Spring"

courses.stream()

.filter(course -> course.contains("Spring"))

.forEach(System.out::println);

// Exercise 4: Print courses with at least four letters

courses.stream()

.filter(course -> course.length() >= 4)

.forEach(System.out::println);

}

private static void printOddNumbersInListFunctional(List<Integer> numbers) {

numbers.stream()

.filter(number -> number % 2 != 0) // Filter odd numbers

.forEach(System.out::println); // Print each odd number

}

}

```

---

### Step 06: Using map in Functional Programs - with Exercises

### Key Objectives

- Learn how to use the \*\*map()\*\* function in Java streams.

- Understand how to transform data using \*\*lambda expressions\*\* and `map()`.

- Perform exercises to print squares, cubes, and the lengths of strings.

### Introduction to Mapping in Functional Programming

- In this lesson, we will learn how to transform elements in a stream using the \*\*map()\*\* function.

- The `map()` function allows us to apply a function to each element in a stream, transforming the data in some way.

- We’ll start by printing the \*\*squares\*\* of even numbers and then move on to other exercises.

### Printing Squares of Even Numbers

- The task is to print the squares of all \*\*even numbers\*\* in the list.

- Instead of simply printing the numbers, we will use the \*\*map()\*\* function to transform each even number into its square.

#### Code:

```java

private static void printSquaresOfEvenNumbersInListFunctional(List<Integer> numbers) {

numbers.stream()

.filter(number -> number % 2 == 0) // Filter only even numbers

.map(number -> number \* number) // Map each number to its square

.forEach(System.out::println); // Print each squared number

}

```

- `filter(number -> number % 2 == 0)`: Filters only the even numbers.

- `map(number -> number \* number)`: Maps each even number to its square.

- `forEach(System.out::println)`: Prints each squared number.

#### Output:

```

144

16

36

4

144

```

### Exercise 5 - Printing Cubes of Odd Numbers

- Now, let’s extend the logic to print the cubes of the odd numbers.

- A cube is calculated by multiplying a number by itself three times `(number \* number \* number)`.

#### Code:

```java

private static void printCubesOfOddNumbersInListFunctional(List<Integer> numbers) {

numbers.stream()

.filter(number -> number % 2 != 0) // Filter only odd numbers

.map(number -> number \* number \* number) // Map each odd number to its cube

.forEach(System.out::println); // Print each cubed number

}

```

- `filter(number -> number % 2 != 0)`: Filters only the odd numbers.

- `map(number -> number \* number \* number)`: Maps each odd number to its cube.

- `forEach(System.out::println)`: Prints each cubed number.

#### Output:

```

729

2197

3375

```

### Exercise 6 - Printing the Number of Characters in Course Names

- In this exercise, we will print the length of each course name in a list of courses.

- We will use the `map()` function to transform each course name into its length (number of characters).

#### Code:

```java

List<String> courses = List.of("Spring", "Spring Boot", "API", "Microservices",

"AWS", "PCF", "Azure", "Docker", "Kubernetes");

courses.stream()

.map(course -> course.length()) // Map each course to its length

.forEach(System.out::println); // Print the length of each course

```

- `map(course -> course.length())`: Maps each course name to the number of characters in the name.

- `forEach(System.out::println)`: Prints the length of each course name.

#### Output:

```

6

11

3

13

3

3

5

6

10

```

### Printing Course Names with Their Lengths

- To make it more informative, let’s modify the code to print both the course name and its length, side by side.

#### Code:

```java

courses.stream()

.map(course -> course + " " + course.length()) // Map to "course name + length"

.forEach(System.out::println); // Print the course and its length

```

- The `map(course -> course + " " + course.length())` expression maps each course to a string that combines the course name and its length, separated by a space.

- This will print both the name of the course and the number of characters in the name.

#### Output:

```

Spring 6

Spring Boot 11

API 3

Microservices 13

AWS 3

PCF 3

Azure 5

Docker 6

Kubernetes 10

```

### Full Code

- Here’s the full code with all the exercises included:

```java

public class FP01Exercises {

public static void main(String[] args) {

List<Integer> numbers = List.of(12, 9, 13, 4, 6, 2, 4, 12, 15);

// Printing squares of even numbers

printSquaresOfEvenNumbersInListFunctional(numbers);

// Printing cubes of odd numbers

printCubesOfOddNumbersInListFunctional(numbers);

// Printing number of characters in each course name

List<String> courses = List.of("Spring", "Spring Boot", "API", "Microservices",

"AWS", "PCF", "Azure", "Docker", "Kubernetes");

// Printing the length of each course

courses.stream()

.map(course -> course.length())

.forEach(System.out::println);

// Printing both course name and length

courses.stream()

.map(course -> course + " " + course.length())

.forEach(System.out::println);

}

private static void printSquaresOfEvenNumbersInListFunctional(List<Integer> numbers) {

numbers.stream()

.filter(number -> number % 2 == 0) // Filter only even numbers

.map(number -> number \* number) // Map to square of each number

.forEach(System.out::println); // Print each squared number

}

private static void printCubesOfOddNumbersInListFunctional(List<Integer> numbers) {

numbers.stream()

.filter(number -> number % 2 != 0) // Filter only odd numbers

.map(number -> number \* number \* number) // Map to cube of each number

.forEach(System.out::println); // Print each cubed number

}

}

```

---

### Step 07: Quick Review of Functional Programming Basics

### Key Objectives

- Review the foundational concepts of functional programming learned in the previous lessons.

- Understand the key functional programming constructs like \*\*streams\*\*, \*\*filter\*\*, \*\*forEach\*\*, \*\*map\*\*, \*\*lambda expressions\*\*, and \*\*method references\*\*.

### 1. Streams in Functional Programming

- In functional programming, we focus on specifying \*\*what to do\*\* rather than \*\*how to do it\*\*. The first step is to convert a \*\*List\*\* into a \*\*stream\*\*, which is a sequence of elements that can be processed functionally.

#### Example:

```java

List<Integer> numbers = List.of(12, 9, 13, 4, 6, 2);

numbers.stream() // Convert List to Stream

.filter(number -> number % 2 != 0) // Allow only odd numbers

.forEach(System.out::println); // Print each odd number

```

- The stream represents the sequence of elements from the list.

- The functional operations such as filter and forEach specify what should be done with each element in the stream.

### 2. The `filter()` Method

- The `filter()` method is used to allow only elements that meet a certain condition to pass through. All other elements are filtered out.

#### Example:

```java

numbers.stream()

.filter(number -> number % 2 == 0) // Allow only even numbers

.forEach(System.out::println);

```

- In this example, we use `filter()` to allow only even numbers through the stream by checking if `number % 2 == 0`.

- If the condition is true, the number is passed to the next stage of the stream pipeline.

### 3. The `forEach()` Method

- The `forEach()` method is used to consume each element in the stream. This is typically where side effects occur, such as printing elements to the console.

#### Example:

```java

numbers.stream()

.forEach(System.out::println); // Print each number

```

- `forEach()` applies an operation (like printing) to each element in the stream.

- It is a terminal operation, meaning that it processes all elements in the stream and doesn’t return a new stream.

### 4. The `map()` Method

- The `map()` method is used to transform each element in the stream. It allows you to convert one value into another.

#### Example:

```java

numbers.stream()

.map(number -> number \* number) // Square each number

.forEach(System.out::println);

```

- `map()` takes each element in the stream and applies a transformation (in this case, squaring the number).

- The transformed values are then passed to the next stage of the stream pipeline.

### 5. Lambda Expressions

- A Lambda Expression is a simplified way of defining a method in a functional programming style. It provides a concise way to express a method using a shorter syntax.

#### Example:

```java

numbers.stream()

.filter(number -> number % 2 == 0) // Lambda expression to check even numbers

.map(number -> number \* number) // Lambda expression to square the numbers

.forEach(System.out::println);

```

- `number -> number % 2 == 0` is a lambda expression that checks if a number is even.

- `number -> number \* number` is a lambda expression that squares each number.

- Lambda expressions allow us to define functional logic inline without writing separate methods.

#### Simplifying Methods with Lambdas:

- Instead of writing a full method, like:

```java

private static boolean isEven(int number) {

return number % 2 == 0;

}

```

- We can replace it with a simpler lambda expression:

```java

number -> number % 2 == 0

```

### 6. Method References

- A Method Reference is a shorthand notation for referring to methods in Java, particularly when using functional programming.

- Method references allow you to call a method directly without explicitly using a lambda expression.

#### Example:

```java

numbers.stream()

.forEach(System.out::println); // Method reference to print each number

```

- `System.out::println` is a method reference that points to the println method, allowing us to print each element of the stream.

- We also used custom method references like `ClassName::methodName` to call static methods, such as `FP01Functional::isEven`.

#### Custom Method Reference:

```java

numbers.stream()

.filter(FP01Functional::isEven) // Method reference to custom static method

.forEach(System.out::println);

```

---

## Playing with Streams

### Step 01: Learning Stream Operations - Calculate Sum using reduce

- In this section, we explore different ways to perform operations on streams in Java, specifically focusing on calculating the sum of a list of numbers.

- We will look at both a traditional structured approach and a functional programming approach using the Stream API and the `reduce` method.

### 1. Summing Numbers with the Traditional (Structured) Approach

- The structured approach involves manually looping through the list of numbers and accumulating the result in a temporary variable.

### Key Objectives:

- Understand how to sum numbers in a list using a loop.

- Learn how to initialize a sum variable and use an enhanced `for` loop to iterate through the list.

- Return the final sum after processing all numbers.

#### Code: Traditional Structured Approach

```java

import java.util.List;

public class FP02Structured {

public static void main(String[] args) {

// Step 1: Create a list of numbers

List<Integer> numbers = List.of(12, 9, 13, 4, 6, 2, 4, 12, 15);

// Step 2: Calculate the sum using the structured approach

System.out.println(addListStructured(numbers)); // Output: 77

}

// Step 3: Define a method to calculate the sum

private static int addListStructured(List<Integer> numbers) {

int sum = 0; // Initialize sum to 0

// Loop over each number in the list and add it to sum

for (int number : numbers) {

sum += number;

}

// Return the final sum

return sum;

}

}

```

- A list of numbers is created using `List.of(...)`.

- The `addListStructured` method loops through the list, adding each number to the sum variable initialized at 0.

- After the loop finishes, the final sum is returned and printed. In this example, the sum is `77`.

### 2. Functional Programming Approach: Using Streams and reduce

- Java’s Stream API offers a more functional and declarative way to process data.

- In this approach, we use the reduce method to calculate the sum of the numbers in the list.

### Key Objectives:

- Learn how to create a stream from a list of numbers.

- Use the reduce method to combine elements into a single result.

- Understand how the reduce method works with an identity value and an accumulator function.

#### Code: Functional Approach Using reduce

```java

import java.util.List;

public class FP02Functional {

public static void main(String[] args) {

// Step 1: Create a list of numbers

List<Integer> numbers = List.of(12, 9, 13, 4, 6, 2, 4, 12, 15);

// Step 2: Calculate the sum using the functional approach

System.out.println(addListFunctional(numbers)); // Output: 77

}

// Step 3: Define a method to calculate the sum using streams

private static int addListFunctional(List<Integer> numbers) {

return numbers.stream() // Convert the list to a stream

.reduce(0, Integer::sum); // Use reduce to sum the elements

}

}

```

- A list of numbers is converted into a stream using `stream()`.

- The reduce method is used to combine the numbers into a sum. The first argument (0) is the identity value, and `Integer::sum` is the method reference used to add the numbers.

- The final result, which is the sum of the numbers, is returned and printed. The output is `77`.

### 3. Understanding the reduce Method

- The reduce method is a terminal operation in Java Streams that allows you to combine elements into a single result.

### Key Objectives:

- Understand the role of the identity value in the reduce method.

- Learn how the accumulator function works to combine elements in the stream.

- Apply reduce to sum numbers, but also recognize that it can be used for other operations such as finding the maximum, minimum, etc.

### Key Concepts:

- \*\*Identity:\*\* The initial value of the reduction operation. In this case, 0 is the identity for summing numbers.

- \*\*Accumulator:\*\* A function that defines how to combine two values. In this case, Integer::sum adds two numbers.

- \*\*Result:\*\* The final result after processing all elements in the stream.

Example:

```java

return numbers.stream()

.reduce(0, Integer::sum); // Reduce the numbers to their sum

```

#### In this code:

- The stream is reduced to a single value (sum) starting from 0 as the identity value.

- `Integer::sum` acts as the accumulator function to combine each pair of numbers.

### 4. Parallel Streams (Theoretical Concept)

- Java Streams can also be processed in parallel, enabling multi-threaded execution for performance improvement on large datasets.

### Key Objectives:

- Understand how parallel streams distribute data across multiple threads.

- Learn how changing `stream()` to `parallelStream()` can improve performance for large datasets.

#### Example of Using Parallel Stream:

```java

private static int addListParallel(List<Integer> numbers) {

return numbers.parallelStream()

.reduce(0, Integer::sum); // Sum numbers using parallel streams

}

```

- The `parallelStream()` method divides the list into smaller parts, processing them in parallel across multiple threads.

- The reduce operation remains the same but is now executed concurrently, which can improve performance for larger datasets.

---

### Step 02: Playing with `reduce` Stream Operation

- In this section, we dive deeper into the `reduce` method in Java Streams and explore what happens in the background when performing a reduction operation.

- We look at how values are aggregated step-by-step and how we can replace custom functions with lambda expressions or method references.

### Key Objectives:

- Understand how the `reduce` method works internally.

- Explore how values are combined during the reduction process.

- Learn to replace custom functions with lambda expressions.

- Use method references as a more concise way to perform operations like summing.

### 1. Understanding How `reduce` Works Internally

- The `reduce` method aggregates a stream of elements into a single result by combining elements one by one.

- In this case, we are summing numbers from a list. To better understand how `reduce` works, we can print out the values being combined during the reduction process.

### Key Objectives:

- Print out the intermediate values during the reduction.

- Understand how the aggregate value and the next number are combined at each step.

#### Code: Printing Intermediate Values in `reduce`

```java

import java.util.List;

public class FP02DebuggingReduce {

public static void main(String[] args) {

List<Integer> numbers = List.of(12, 9, 13, 4, 6, 2, 4, 12, 15);

// Step 1: Use a custom sum function with a sysout to print the values being passed to a and b

int sum = numbers.stream()

.reduce(0, (a, b) -> {

System.out.println("a: " + a + ", b: " + b); // Print current values of a and b

return a + b; // Return the sum of a and b

});

System.out.println("Sum: " + sum); // Output the final sum

}

}

```

- The reduce method takes an initial value (0 in this case) and combines each element in the stream with an accumulator function.

- We print out the values of a (the aggregate) and b (the next number in the list) during each step of the reduction process.

- The output shows how values are accumulated step-by-step.

#### Output:

```

a: 0, b: 12

a: 12, b: 9

a: 21, b: 13

a: 34, b: 4

a: 38, b: 6

a: 44, b: 2

a: 46, b: 4

a: 50, b: 12

a: 62, b: 15

Sum: 77

```

- The reduction starts with an initial value of 0 (the identity).

- Each number from the list is added to the current aggregate value.

#### For example:

```

0 + 12 = 12

12 + 9 = 21

21 + 13 = 34, and so on.

```

- The final result is the sum of all numbers: 77.

### 2. Using Lambda Expressions in `reduce`

- Instead of using a custom function, we can simplify the code with lambda expressions.

- A lambda expression allows you to define the logic of the reduction in a more concise way.

### Key Objectives:

- Replace custom logic with lambda expressions.

- Simplify the reduce operation by specifying how two values (x and y) should be combined.

#### Code: Using Lambda Expressions

```java

import java.util.List;

public class FP02LambdaReduce {

public static void main(String[] args) {

List<Integer> numbers = List.of(12, 9, 13, 4, 6, 2, 4, 12, 15);

// Step 1: Use a lambda expression to sum the numbers

int sum = numbers.stream()

.reduce(0, (x, y) -> x + y); // Lambda to add two numbers

System.out.println("Sum: " + sum); // Output the final sum

}

}

```

- The lambda expression `(x, y) -> x + y` replaces the custom sum logic.

- Here, x represents the aggregate (current sum) and y represents the next number in the list.

- The reduce method combines x and y by adding them together, just like in the previous step, but with less code.

#### Output:

```

Sum: 77

```

### 3. Using Method References in reduce

- Java provides predefined methods for common operations like addition, which can be used with method references to make the code even simpler.

- In this case, we can use `Integer::sum` instead of writing our own sum logic.

### Key Objectives:

- Replace the lambda expression with a method reference for predefined operations.

- Use `Integer::sum` to add numbers instead of manually specifying the addition.

#### Code: Using Method References

```java

import java.util.List;

public class FP02MethodReferenceReduce {

public static void main(String[] args) {

List<Integer> numbers = List.of(12, 9, 13, 4, 6, 2, 4, 12, 15);

// Step 1: Use method reference Integer::sum to perform the reduction

int sum = numbers.stream()

.reduce(0, Integer::sum); // Method reference to sum numbers

System.out.println("Sum: " + sum); // Output the final sum

}

}

```

- `Integer::sum` is a method reference that points to the predefined sum method in the Integer class, which takes two integers and returns their sum.

- The method reference simplifies the reduce operation even further, making the code cleaner and easier to understand.

#### Output:

```

Sum: 77

```

### Benefits of Using Method References:

- \*\*Concise:\*\* Reduces the amount of code written.

- \*\*Readability:\*\* Improves code readability by directly using predefined methods for common operations.

---

### Step 03: Exploring Streams with Puzzles in JShell

- In this lecture, we will explore the powerful \*\*Stream API\*\* in Java, focusing on how to use the `reduce` method to aggregate values from a stream.

- We will also learn how to use \*\*JShell\*\* for interactive coding, allowing us to quickly test and understand how streams work.

### Key Objectives

- Understand how to use JShell for experimenting with Java code.

- Learn how the `reduce` method works for aggregating values.

- Discover how to find the maximum and minimum values in a list using streams.

- Practice using lambda expressions to manipulate stream operations.

- Develop hands-on experience with interactive coding using JShell.

### 1. Introduction to \*\*JShell\*\*

- \*\*JShell\*\* is a tool introduced in Java 9 that allows you to experiment with Java code interactively.

- This is especially useful for learners who want to see the results of their code without setting up an entire project.

- With \*\*JShell\*\*, you can type in and execute Java code line by line. This makes learning Java easier because you don't need to write a complete program just to test a small piece of code.

- You can immediately see the results of what you’ve written, helping you understand concepts more quickly.

- \*\*For example\*\*: If you want to print a simple message, you can do it directly in JShell:

#### Code:

```java

System.out.println("Hello, JShell!");

```

#### Output:

```

Hello, JShell!

```

- This is a quick way to test your Java code without the need for an IDE or a complete Java project.

### Key Objectives:

- Learn how to launch and use JShell for quick Java experiments.

- Understand the benefits of interactive coding for immediate feedback.

### 2. Creating a List

- Next, we create a list of integers. This list will be used to perform various stream operations throughout the lecture.

- In Java, you can use the `List.of(...)` method to create a list of values.

- This method is simple and efficient for creating immutable lists. In our case, we create a list of integers which we will then manipulate using stream operations.

#### Code:

```java

List<Integer> numbers = List.of(12, 9, 13, 4, 6);

```

- This line creates a list with the numbers 12, 9, 13, 4, and 6.

- This list will be used in our reduce operations later on.

### Key Objectives:

- Understand how to create a list of integers in Java.

Use `List.of(...)` to create an immutable list for stream operations.

### 3. Understanding the Reduce Method

- The reduce method in streams is a powerful operation that allows us to aggregate values in a list or stream.

- It takes two parameters: an initial value and a function that defines how to combine elements.

### Example 1: Summing Values

- In this example, we use reduce to sum the numbers in our list.

#### Code:

```java

int sum = numbers.stream().reduce(0, (x, y) -> x + y);

```

- \*\*Identity value:\*\* The first argument 0 is the initial value, which serves as the starting point for the summation.

- \*\*Lambda expression:\*\* The second argument `(x, y) -> x + y` is a lambda expression that specifies how to combine each element with the current result.

- In this case, it adds the two numbers together.

#### Output:

```

Sum: 77

```

- This result shows that the sum of the numbers in the list is 77.

### Key Objectives:

- Understand the purpose of the initial value (identity) in the reduce method.

- Learn how to use a lambda expression to combine values in a stream.

- Explore how the reduce method aggregates all elements in the list into a single result.

### 4. Exploring Different Reduce Operations

- Now that we understand the basic use of reduce, let's explore different ways we can manipulate the reduce operation to see how the results change.

### Example 2: Returning Zero

- In this example, we change the reduce operation to always return the initial value, which is 0.

#### Code:

```java

int result = numbers.stream().reduce(0, (x, y) -> x);

```

- In this case, the lambda expression `(x, y) -> x` always returns the value of x (the current aggregate).

- Since the initial value is 0 and the function keeps returning x, the result will be:

#### Output:

```

Result: 0

```

- This is because the lambda expression never updates the value of x, so the result remains the initial value of 0.

### Example 3: Returning the Second Number

- Next, we modify the lambda expression to return the second argument (y) instead of the first.

### Key Objectives:

- Understand how the return values in the lambda expression affect the output of the reduce method.

- Experiment with different lambda expressions to see how they change the result of the reduction.

#### Code:

```java

int result = numbers.stream().reduce(0, (x, y) -> y);

```

- Here, the lambda expression `(x, y) -> y` returns the second argument (y), which is the current element from the list.

- This means that the last element processed by the stream will be the final result. The output will be:

#### Output:

```

Result: 15

```

- This is because 15 is the last number in the list.

### 5. Finding Maximum Values

- Now, let's find the maximum value in the list using the reduce method.

### Key Objectives:

- Learn how to find the maximum value in a list using the reduce method.

- Understand why using `Integer.MIN\_VALUE` is important when dealing with negative numbers.

#### Code:

```java

int max = numbers.stream().reduce(0, (x, y) -> x > y ? x : y);

```

- In this case, we compare two numbers at a time and return the larger one using the condition `x > y ? x : y`. This ensures that as we iterate through the list, we keep track of the largest number we've seen so far.

- However, this approach has a limitation: it won't work properly if the list contains negative numbers.

- Since we are using 0 as the initial value, it will always be considered larger than any negative numbers in the list.

### Suggested Improvement

- To handle negative numbers, we can use `Integer.MIN\_VALUE` as the starting value.

- This ensures that any number in the list will be larger than the initial value.

#### Code:

```java

int max = numbers.stream().reduce(Integer.MIN\_VALUE, (x, y) -> x > y ? x : y);

```

### 6. Finding Minimum Values

- Just like finding the maximum value, we can use the reduce method to find the minimum value in the list.

### Key Objectives:

- Learn how to find the minimum value in a list using the reduce method.

- Understand how using Integer.MAX\_VALUE helps in finding the smallest number in a list.

#### Code:

```java

int min = numbers.stream().reduce(Integer.MAX\_VALUE, (x, y) -> x < y ? x : y);

```

- Here, we start with `Integer.MAX\_VALUE`, which is the largest possible integer value. As we compare each element in the list, the lambda expression `x < y ? x : y` ensures that the smaller of the two values is returned.

#### Output:

```

Minimum Value: 2

```

- This confirms that the smallest value in the list is 2.

---

### Step 04: Do Functional Programming Exercises with Streams and reduce

- In this lecture, we will work through several exercises related to Java Streams and the `reduce` function.

- We will focus on practical applications of functional programming by squaring numbers, cubing numbers, and filtering for odd numbers. This hands-on approach will solidify our understanding of streams and their operations.

### Key Objectives

- Understand how to use the `map` function to transform data in streams.

- Learn to use the `reduce` method for aggregating results.

- Apply filtering techniques to isolate specific elements in a stream.

- Practice using functional programming concepts in Java.

### Exercises

- \*\*Exercise 7\*\*: Square each number in a given list and find the total sum of these squares.

- \*\*Exercise 8\*\*: Cube each number in the list and determine the total sum of these cubes.

- \*\*Exercise 9\*\*: Identify and sum only the odd numbers from the list.

### Exercise 7: Sum of Squares

### Key Objectives

- Learn how to use the `map` function to transform elements in a stream.

- Understand the `reduce` method and how it can aggregate results from a stream.

- Let's start with \*\*Exercise 7\*\*, which requires us to square every number in the list and find the sum of these squares.

### Steps

1. \*\*Create a stream from the list of numbers\*\*.

2. \*\*Use `map` to square each number\*\*.

3. \*\*Use `reduce` to sum up the squared values\*\*.

#### Code:

```java

int sumOfSquares = numbers.stream().map(x -> x \* x).reduce(0, Integer::sum);

```

- The `map(x -> x \* x)` transformation applies the squaring operation to each number in the list.

- The `reduce(0, Integer::sum)` operation takes the squared values and sums them up, starting from an initial value of 0.

#### Output:

```

Sum of Squares: 835

```

### Exercise 8: Sum of Cubes

### Key Objectives

- Understand how to modify the mapping operation to perform cubing.

- Practice using reduce for aggregation with different mathematical operations.

#### Steps

- Use map to cube each number.

- Use reduce to sum the cubed values.

#### Code:

```java

int sumOfCubes = numbers.stream().map(x -> x \* x \* x).reduce(0, Integer::sum);

```

- Similar to Exercise 7, we apply the cubing operation with `map(x -> x \* x \* x)`.

- Again, we use reduce to aggregate the results into a single sum.

### Key Objectives:

- Practice using the map function for transformations in streams.

- Understand the flexibility of the reduce method for different operations.

### Exercise 9: Sum of Odd Numbers

### Key Objectives

- Learn how to filter elements in a stream based on specific criteria.

'

- Understand how to combine filtering with reduction to achieve the desired result.

#### Steps

- Use filter to isolate odd numbers.

- Use reduce to sum these filtered values.

#### Code:

```java

int sumOfOddNumbers = numbers.stream().filter(x -> x % 2 == 1).reduce(0, Integer::sum);

```

- The `filter(x -> x % 2 == 1)` condition checks each number to see if it is odd (i.e., it has a remainder of 1 when divided by 2).

- After filtering, we use reduce to sum the remaining numbers.

#### Output:

```

Sum of Odd Numbers: 37

```

---

### Step 05: Learn Stream Operations - distinct and sorted

- In this lecture, we will explore two important stream operations in Java, \*\*distinct\*\* and \*\*sorted\*\*.

- These operations allow us to manipulate collections of data effectively by filtering out duplicates and arranging elements in a specific order.

### Key Objectives

- Understand the purpose of the `distinct` method in streams.

- Learn how to use the `sorted` method to order elements.

- Explore the combination of `distinct` and `sorted` for more refined data manipulation.

### 1. Introduction to Stream Operations

### Key Objectives

- Familiarize yourself with basic stream operations such as `distinct` and `sorted`.

- In the previous step, we learned about the `reduce` method. Now, we will look at two additional methods that are essential for stream manipulation: `distinct` and `sorted`.

### 2. Using the Distinct Method

### Key Objectives

- Learn how to eliminate duplicate values from a stream using the `distinct` method.

- Understand how to print distinct elements from a stream.

Let's say we have a stream of numbers. To get only the distinct numbers from this stream, we can use the following approach:

#### Code:

```java

numbers.stream().distinct().forEach(System.out::println);

```

- The `distinct()` method filters the stream to only include unique elements.

- The `forEach(System.out::println)` prints each distinct number to the console.

- When executed, this code will print only the unique numbers from the list, omitting any duplicates.

- For instance, if the list contains duplicates of 4 and 12, those numbers will appear only once in the output.

### 3. Using the Sorted Method

### Key Objectives

- Understand how the sorted method can be used to arrange elements in ascending order.

- Explore how sorted values retain duplicates.

- To sort the numbers in a stream, we can use the `sorted()` method:

#### Code:

```java

numbers.stream().sorted().forEach(System.out::println);

```

- The `sorted()` method sorts the elements of the stream in their natural order (ascending).

- The duplicates are retained, meaning if a number appears multiple times, it will be printed each time.

- For example, if the list includes 2, 4, 4, 6, 9, 12, 12, 13, 15, the sorted

output will be:

```

2

4

4

6

9

12

12

13

15

```

### 4. Combining Distinct and Sorted

### Key Objectives

- Learn how to use both distinct and sorted together for more refined output.

If we want to get distinct values and have them sorted, we can combine both methods:

#### Code:

```java

numbers.stream().distinct().sorted().forEach(System.out::println);

```

- This combination first filters out duplicates with `distinct()` and then sorts the unique numbers with `sorted()`.

- The final output will be a list of distinct numbers in ascending order.

- For the same input list, the output will be:

```

2

4

6

9

12

13

15

```

### 5. Sorting Strings in Streams

### Key Objectives

- Understand how the sorted method can also be applied to collections of strings.

- In addition to numbers, we can also use the sorted method on a list of strings.

#### Code:

```java

List<String> courses = List.of("Java", "Python", "C++", "JavaScript");

courses.stream().sorted().forEach(System.out::println);

```

- This code sorts the list of course names in alphabetical order.

- The `sorted()` method arranges the strings, and `forEach(System.out::println)` prints each one.

#### Output:

```java

C++

Java

JavaScript

Python

```

---

### Step 06: Using Comparators to Sort Streams with sorted

- In this lecture, we will explore how to use \*\*Comparators\*\* in Java Streams to sort data according to custom criteria. We'll learn how to sort items in natural order, reverse order, and even by specific attributes such as string length.

### Key Objectives

- Understand how to use the `sorted` method with natural order.

- Learn to apply `Comparator` to sort items in reverse order.

- Create custom comparators for sorting based on specific criteria, such as string length.

- Practice using lambda expressions with comparators.

### 1. Introduction to Sorting with Comparators

### Key Objectives

- Familiarize yourself with the concept of natural order sorting using streams.

- In the previous lecture, we discussed how to sort a list of courses alphabetically. Now, we will explore how to define a custom sorting algorithm.

To sort the courses, we can use the following code:

#### Code:

```java

courses.stream().sorted().forEach(System.out::println);

```

- The `sorted()` method sorts the elements of the stream in their natural order. For strings, this means sorting alphabetically.

### 2. Sorting in Reverse Order

### Key Objectives

- Learn how to sort items in reverse order using Comparator.

- If we want to sort the courses in reverse alphabetical order, we can use a comparator:

#### Code:

```java

courses.stream()

.sorted(Comparator.naturalOrder().reversed())

.forEach(System.out::println);

```

- Here, `Comparator.naturalOrder()` specifies the natural order for sorting, while `reversed()` modifies this order to be descending.

- When executed, this will print the course names in reverse order, allowing you to see the courses sorted from Z to A.

- For example, if the original list is `["Java", "Python", "C++", "JavaScript", "Spring Boot"]`, the output will be:

```java

Spring Boot

Python

JavaScript

Java

C++

```

### 3. Creating Custom Comparators

### Key Objectives

- Understand how to define a custom comparator using lambda expressions.

- Sometimes, you may want to sort items based on specific criteria. For instance, let’s sort the courses by the length of their names.

#### Code:

```java

courses.stream()

.sorted(Comparator.comparing(str -> str.length()))

.forEach(System.out::println);

```

- In this example, `Comparator.comparing(str -> str.length())` creates a comparator that sorts the strings based on their lengths.

- This approach allows for flexible sorting based on different attributes.

- If the courses are `["Java", "Python", "C++", "JavaScript", "Spring Boot"]`, the output will be:

```java

C++

Java

Python

JavaScript

Spring Boot

```

---

### Step 07: Collecting Stream Elements to List using collect

- In this lecture, we will explore how to collect elements from a stream into a list using the `collect` method. We will see how to transform data, create new lists based on existing lists, and filter elements during the collection process.

### Key Objectives

- Understand how to use the `collect` method to gather stream elements into a list.

- Learn how to apply mapping to transform elements in a stream.

- Explore filtering techniques to create new lists based on specific criteria.

- Practice using lambda expressions in stream operations.

### 1. Introduction to Collecting Elements

### Key Objectives

- Familiarize yourself with stream manipulations and how to create new lists.

- Until now, we have manipulated elements of a list by performing operations like doubling, squaring, and cubing. We also learned how to use the `reduce()` function to produce a single result.

- Now, we will focus on creating new lists based on existing data.

### Example Scenario

Let’s say we have a list of numbers and we want to create a new list that contains the squares of those numbers.

### 2. Creating a Method to Double Values

### Key Objectives

- Learn how to define a method that utilizes streams for element transformation.

- We will start by creating a method called `doubleList` that takes a list of numbers and returns a new list containing their squares.

#### Code:

```java

List<Integer> squaredNumbers = doubleList(numbers);

```

- Here, `doubleList` is a method we will define to process the input list of numbers.

- Focusing on the `doubleList` Method

- Inside the `doubleList` method, we will implement the following steps:

### Convert the list to a stream.

- Use map to transform each number by squaring it.

- Collect the results into a new list.

#### Code:

```java

public List<Integer> doubleList(List<Integer> numbers) {

return numbers.stream()

.map(number -> number \* number) // Square each number

.collect(Collectors.toList()); // Collect to a list

}

```

- The map method applies the squaring operation, transforming each number to its square.

- The `collect(Collectors.toList())` method gathers the transformed elements into a new list.

- When this method is executed with a list of numbers, it returns a new list containing the squares.

### 3. Filtering Even Numbers

### Key Objectives

- Understand how to filter elements in a stream based on specific criteria.

Now, let’s create a list containing only the even numbers from the original list.

#### Code:

```java

List<Integer> evenNumbersOnly = numbers.stream().filter(x -> x % 2 == 0).collect(Collectors.toList());

```

- The `filter(x -> x % 2 == 0)` condition checks each number to see if it is even.

- The filtered results are then collected into a new list.

- Running this code will yield a list containing only the even numbers from the original list.

### 4. Creating a List of Lengths of Course Titles

### Key Objectives

- Learn how to transform strings and collect their lengths into a list.

In the next exercise, we will create a list that contains the lengths of all course titles.

#### Code:

```java

List<Integer> courseLengths = courses.stream().map(course -> course.length()).collect(Collectors.toList());

```

- The `map(course -> course.length())` operation transforms each course title into its length.

- The lengths are then collected into a new list.

- When executed, this code provides a list of integers representing the lengths of each course title.

---

### Step 08: Reviewing Streams - Intermediate and Stream Operations

- In this lecture, we will review various operations related to streams in Java, specifically focusing on \*\*intermediate\*\* and \*\*terminal\*\* operations.

- Understanding these concepts is crucial for effectively working with streams and manipulating data collections.

### Key Objectives

- Distinguish between intermediate and terminal operations in Java streams.

- Understand how intermediate operations transform streams and return new streams.

- Learn about terminal operations that produce a final result from a stream.

- Review practical examples of stream operations.

### 1. Introduction to Stream Operations

### Key Objectives

- Familiarize yourself with the different types of stream operations.

- So far in this course, we have performed numerous manipulations on lists, including sorting, filtering, and transforming elements.

- Now, let's categorize these operations into two main types: \*\*intermediate operations\*\* and \*\*terminal operations\*\*.

### 2. Intermediate Operations

### Key Objectives

- Learn about intermediate operations and their characteristics.

- Intermediate operations are those that transform a stream into another stream. They do not modify the original stream but instead return a new stream.

#### Some common intermediate operations include:

- \*\*distinct()\*\*: Filters out duplicate values from the stream.

- \*\*sorted()\*\*: Sorts the elements in natural order.

- \*\*map()\*\*: Transforms each element in the stream into a new value.

- \*\*filter()\*\*: Retains only those elements that match a specified condition.

### Code:

```java

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

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

List<Integer> squaredNumbers = numbers.stream().map(n -> n \* n).collect(Collectors.toList());

List<Integer> evenNumbers = numbers.stream().filter(n -> n % 2 == 0).collect(Collectors.toList());

```

- Each of these operations takes the input stream and performs a transformation, resulting in a new stream.

- For example, `distinct()` returns a stream containing only unique elements, while map() applies a function to each element and returns a stream of modified values.

### 3. Terminal Operations

### Key Objectives

- Understand terminal operations and their role in stream processing.

- Terminal operations are the final operations in a stream processing pipeline. Unlike intermediate operations, they do not return a stream; instead, they produce a result or a side effect.

#### Common terminal operations include:

- `forEach()`: Iterates over each element in the stream and performs an action.

- `collect()`: Gathers the elements of a stream into a collection, such as a list, set, or map.

- `reduce()`: Combines the elements of a stream into a single result, based on a specified function.

#### Code:

```java

numbers.stream().forEach(System.out::println);

List<Integer> collectedNumbers = numbers.stream().collect(Collectors.toList());

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

```

- `forEach()` consumes each element of the stream without returning anything (void).

- `collect()` transforms the stream into a collection type, such as a list or set.

- `reduce()` aggregates the stream's elements into a single value, like summing all the numbers.

- Terminal operations mark the end of the stream processing.

- Once a terminal operation is invoked, the stream is consumed, and no further operations can be performed on it.

---

## Exploring Java Functional Interfaces and Lambdas

### Step 01: Getting Started with Functional Interfaces - Predicate, Consumer and Function

### Key Objectives

- Understand what functional interfaces are and how they enable passing logic to methods.

- Learn about three key functional interfaces in Java: \*\*Predicate\*\*, \*\*Consumer\*\*, and \*\*Function\*\*.

- Explore how lambda expressions work under the hood and how functional interfaces simplify method logic.

- Refactor lambda expressions into explicit functional interface instances for better code clarity.

### Introduction to Functional Interfaces

\*\*Objective\*\*: Understand the role of functional interfaces in Java's functional programming.

- Functional interfaces are a core component of Java's approach to functional programming, introduced in Java 8. They allow us to pass logic to methods using lambda expressions.

- A functional interface is defined as an interface that contains exactly one abstract method. Java provides several built-in functional interfaces, such as `Predicate`, `Function`, and `Consumer`, which we will explore in this lecture.

### Refactoring with Lambda Expressions

\*\*Objective\*\*: Transition from using raw logic inside stream operations to assigning them to functional interfaces.

- Begin by looking at a common example using lambda expressions within stream operations to filter, map, and print numbers.

```java

List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6);

numbers.stream()

.filter(x -> x % 2 == 0) // Filter even numbers

.map(x -> x \* x) // Square the numbers

.forEach(System.out::println); // Print each number

```

Here, three operations are performed:

- \*\*Filter:\*\* Filters even numbers using a lambda expression `x -> x % 2 == 0`.

- \*\*Map\*\*: Maps each number to its square with `x -> x \* x`.

- \*\*ForEach\*\*: Prints the squared numbers using method reference `System.out::println`.

- In the background, each of these operations is associated with a functional interface (explored below).

### Introduction to Predicate Interface

- A Predicate is a functional interface that tests a condition on input and returns a boolean value.

- In the example above, the filtering logic is `x -> x % 2 == 0`, which checks if a number is even.

- This lambda expression can be rewritten using the Predicate interface:

```java

Predicate<Integer> isEvenPredicate = x -> x % 2 == 0;

```

- You can refactor the filter operation using the Predicate:

```java

numbers.stream()

.filter(isEvenPredicate) // Use Predicate to filter even numbers

.map(x -> x \* x)

.forEach(System.out::println);

```

- In essence, the Predicate interface provides a structure for filtering collections based on conditions.

#### Behind the Scenes:

A lambda expression like x -> x % 2 == 0 is internally equivalent to the following:

```java

Predicate<Integer> isEvenPredicate2 = new Predicate<>() {

@Override

public boolean test(Integer x) {

return x % 2 == 0;

}

};

```

- This shows how the lambda is actually turned into an object that implements the Predicate interface.

### Introduction to Function Interface

\*\*Objective\*\*: Understand how the Function interface handles input-output transformation logic.

- The Function interface represents a function that accepts one argument and returns a result. It is used to map one input to a corresponding output.

- In the stream example, the map operation squares each number, represented by the lambda `x -> x \* x`.

- We can refactor this lambda into a Function:

```java

Function<Integer, Integer> squareFunction = x -> x \* x;

```

- Now, you can refactor the map operation using this Function:

```java

numbers.stream()

.filter(isEvenPredicate)

.map(squareFunction) // Use Function to map numbers to their squares

.forEach(System.out::println);

```

#### Behind the Scenes:

The lambda `x -> x \* x` is transformed into an object of the Function interface:

```java

Function<Integer, Integer> squareFunction2 = new Function<>() {

@Override

public Integer apply(Integer x) {

return x \* x;

}

};

```

- The Function interface is essential for operations where we take an input and return an output, such as mapping elements in a stream.

### Introduction to Consumer Interface

\*\*Objective\*\*: Learn how the Consumer interface is used for operations that take input but return no result.

- A Consumer is a functional interface that accepts a single input and performs an action without returning any result.

- In our example, the method reference `System.out::println` is a Consumer that prints each number to the console.

- You can explicitly declare the Consumer for printing numbers:

```java

Consumer<Integer> sysoutConsumer = System.out::println;

```

- Refactor the forEach operation using the Consumer:

```java

numbers.stream()

.filter(isEvenPredicate)

.map(squareFunction)

.forEach(sysoutConsumer); // Use Consumer to print each number

```

#### Behind the Scenes:

The method reference System.out::println is internally represented as an implementation of the Consumer interface:

```java

Consumer<Integer> sysoutConsumer2 = new Consumer<>() {

@Override

public void accept(Integer x) {

System.out.println(x);

}

};

```

- The Consumer interface is widely used when you want to perform operations on data without returning a result (e.g., logging or saving data).

#### Full Code

```java

import java.util.List;

import java.util.function.Predicate;

import java.util.function.Function;

import java.util.function.Consumer;

public class FP03FunctionalInterfaces {

public static void main(String[] args) {

List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6);

Predicate<Integer> isEvenPredicate = x -> x % 2 == 0;

Function<Integer, Integer> squareFunction = x -> x \* x;

Consumer<Integer> sysoutConsumer = System.out::println;

numbers.stream()

.filter(isEvenPredicate) // Apply Predicate to filter even numbers

.map(squareFunction) // Apply Function to map numbers to squares

.forEach(sysoutConsumer); // Apply Consumer to print each number

}

}

```

---

### Step 02: Do Exercises with Functional Interfaces - BinaryOperator

### Key Objectives

- Understand how to use the `BinaryOperator` functional interface in Java.

- Learn how method references like `Integer::sum` work behind the scenes.

- Implement and refactor a `BinaryOperator` to perform binary operations on two numbers.

- Explore the relationship between `BinaryOperator` and `BiFunction` interfaces.

### Exercise

\*\*Objective\*\*: Explore how method references work and implement the corresponding functional interface.

- In this exercise, we will take a closer look at the following code snippet, which uses a method reference:

```java

int sum = numbers.stream()

.reduce(0, Integer::sum);

```

- This code reduces a list of numbers to their sum using `Integer::sum`.

- Our goal is to determine the functional interface behind this method reference (`Integer::sum`) and create a manual implementation of this interface.

### Identifying the Functional Interface Behind `Integer::sum`

\*\*Objective\*\*: Understand how `Integer::sum` is linked to the BinaryOperator functional interface.

- The method reference `Integer::sum` refers to a method that adds two integers. This is an example of a binary operation, where two values are combined to produce one result.

- The functional interface behind this is BinaryOperator `<T>`, a specialized version of the BiFunction `<T, T, T>` interface where both the arguments and the result are of the same type (Integer in this case).

```java

BinaryOperator<Integer> sumBinaryOperator = Integer::sum;

```

#### In this code:

- `BinaryOperator<Integer>` is a functional interface that operates on two Integer values and returns an Integer.

- `Integer::sum` is a method reference that implements this binary operation.

#### Behind the Scenes:

- The method reference `Integer::sum` is shorthand for a lambda expression like this:

```java

BinaryOperator<Integer> sumBinaryOperator = (a, b) -> a + b;

```

- This lambda expression performs the same addition operation, taking two integers as input and returning their sum.

### Implementing the BinaryOperator Interface

\*\*Objective\*\*: Create a manual implementation of the BinaryOperator interface and understand its structure.

- We can manually implement the BinaryOperator interface using the following approach:

```java

BinaryOperator<Integer> sumBinaryOperator2 = new BinaryOperator<>() {

@Override

public Integer apply(Integer a, Integer b) {

return a + b;

}

};

```

#### In this code:

- The apply method defines the logic for combining two Integer values, in this case, adding them.

- The result type is also Integer, which matches the input types.

- This manual implementation achieves the same result as `Integer::sum` but without using a method reference or lambda expression. It illustrates the structure of the functional interface behind the method reference.

### Exploring BinaryOperator and BiFunction Interfaces

\*\*Objective\*\*: Understand the relationship between BinaryOperator and BiFunction.

- The `BinaryOperator<T>` interface is a specialized version of the `BiFunction<T, T, R>` interface where both input arguments and the return type are of the same type (T).

- The apply method of BinaryOperator takes two parameters and returns a result of the same type:

```java

@FunctionalInterface

public interface BinaryOperator<T> extends BiFunction<T, T, T> {

T apply(T t1, T t2);

}

```

#### In the example we are working with:

- T is Integer.

- The method apply takes two integers as input (a and b) and returns their sum.

- This explains why `Integer::sum` can be used as a `BinaryOperator<Integer>` because it adds two integers and returns the result, all of type Integer.

### Refactoring BinaryOperator to a Lambda Expression

\*\*Objective\*\*: Refactor the BinaryOperator implementation into a lambda expression for conciseness.

- We can simplify the implementation of the BinaryOperator using a lambda expression, which is more concise and readable:

```java

BinaryOperator<Integer> sumBinaryOperatorLambda = (a, b) -> a + b;

```

- This lambda expression is equivalent to both the method reference (`Integer::sum`) and the manual implementation using the apply method.

#### Explanation of Lambda Expression:

- (a, b) represents two input integers.

- a + b is the operation that adds these integers.

- The result is returned directly from the lambda expression.

- By using lambda expressions, we achieve a more succinct way to express binary operations while adhering to the BinaryOperator interface.

#### Finding the Maximum:

```java

BinaryOperator<Integer> maxBinaryOperator = (a, b) -> a > b ? a : b;

```

#### Finding the Minimum:

```java

BinaryOperator<Integer> minBinaryOperator = (a, b) -> a < b ? a : b;

```

#### Multiplying Two Numbers:

```java

BinaryOperator<Integer> multiplyBinaryOperator = (a, b) -> a \* b;

```

- These operations demonstrate the versatility of the BinaryOperator interface for performing various types of operations on pairs of values.

#### Full Code:

```java

import java.util.List;

import java.util.function.BinaryOperator;

public class BinaryOperatorExample {

public static void main(String[] args) {

List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6);

// Using BinaryOperator with Integer::sum

BinaryOperator<Integer> sumBinaryOperator = Integer::sum;

int sum = numbers.stream()

.reduce(0, sumBinaryOperator);

System.out.println("Sum: " + sum);

// Using BinaryOperator as a lambda expression

BinaryOperator<Integer> sumLambda = (a, b) -> a + b;

int sum2 = numbers.stream()

.reduce(0, sumLambda);

System.out.println("Sum using Lambda: " + sum2);

}

}

```

---

### Step 03: Doing Behavior Parameterization with Functional Programming

### Key Objectives

- Understand the concept of behavior parameterization in functional programming.

- Learn how to pass behavior (logic) as an argument to methods.

- Implement a flexible method to filter and print numbers based on varying criteria.

- Explore how this technique enhances code reusability and maintainability.

### Introduction to Behavior Parameterization

\*\*Objective\*\*: Understand what behavior parameterization means and how it applies to functional programming.

- In this section, we are introduced to \*\*behavior parameterization\*\*, a technique that allows us to pass the logic (or behavior) of a method as an argument to another method.

- This enables us to write more flexible and reusable code.

- For instance, the expression `x % 2 == 0` defines the behavior for determining if a number is even. By extracting this logic, we can avoid repetition in our code and allow for easy modifications in the future.

- This approach is significant because it promotes the idea of treating behaviors as first-class citizens in programming. Instead of just passing data, we are passing logic, which opens up new possibilities for dynamic code execution.

### Initial Implementation of Filtering Numbers

\*\*Objective\*\*: Start with a basic implementation to filter even and odd numbers.

- Lets begin with a simple implementation that prints out even and odd numbers from a list:

```java

numbers.stream()

.filter(x -> x % 2 == 0) // Filter for even numbers

.forEach(System.out::println); // Print each even number

numbers.stream()

.filter(x -> x % 2 != 0) // Filter for odd numbers

.forEach(System.out::println); // Print each odd number

```

#### In this code:

- We use the filter method to separate even and odd numbers based on the provided logic.

- The first block filters for even numbers, while the second block filters for odd numbers.

- However, there is a clear problem, both blocks of code are nearly identical, differing only in the filtering condition.

- This duplication not only makes the code harder to maintain but also suggests that we can refactor it for greater clarity and efficiency.

### Identifying Code Duplication

\*\*Objective\*\*: Recognize the duplication in the existing code and prepare for refactoring.

- We recognize significant duplication, as both blocks perform similar operations with different filtering logic.

- The only difference lies in the condition used to filter numbers. This realization prompts us to consider a more efficient way to manage this behavior.

#### Extracting Predicate Logic

To address this issue, we can create predicates that encapsulate the filtering logic for even and odd numbers:

```java

Predicate<Integer> evenPredicate = x -> x % 2 == 0; // Predicate for even numbers

Predicate<Integer> oddPredicate = x -> x % 2 != 0; // Predicate for odd numbers

```

- By defining these predicates, we can refer to the logic in a more meaningful way and reduce code repetition.

- This refactoring is a fundamental practice in programming that enhances readability and maintainability.

### Creating a Generic Filter and Print Method

\*\*Objective\*\*: Develop a method that accepts a predicate for filtering numbers.

- Next, we create a generic method called `filterAndPrint` that accepts a list of numbers and a predicate as parameters.

- This allows us to filter the numbers based on any condition we choose:

```java

public void filterAndPrint(List<Integer> numbers, Predicate<Integer> predicate) {

numbers.stream()

.filter(predicate) // Use the passed predicate for filtering

.forEach(System.out::println); // Print each filtered number

}

```

- Now, we can use this method to filter even or odd numbers, significantly simplifying our code:

```java

filterAndPrint(numbers, evenPredicate); // Filter and print even numbers

filterAndPrint(numbers, oddPredicate); // Filter and print odd numbers

```

- This approach eliminates duplication and makes the code cleaner and more modular. By externalizing the logic, we are not only enhancing readability but also making our code more adaptable to changes in requirements.

### Enhancing Flexibility with Behavior Parameterization

\*\*Objective\*\*: Explore the benefits of passing behavior as parameters.

- One of the most exciting aspects of this method is that we can pass different filtering logic to filterAndPrint.

- \*\*For example\*\*: To filter multiples of 3, we can define a new predicate:

```java

Predicate<Integer> multipleOfThreePredicate = x -> x % 3 == 0; // Predicate for multiples of 3

filterAndPrint(numbers, multipleOfThreePredicate); // Filter and print multiples of 3

```

- This ability to change the filtering logic dynamically is powerful.

- It means that instead of hardcoding behaviors into our methods, we can now customize them at runtime based on the needs of our application.

- By passing behavior as parameters, we externalize the logic and give our methods the flexibility to adapt. This leads to cleaner code that is easier to maintain and extend.

#### Full Code:

```java

import java.util.List;

import java.util.function.Predicate;

public class FP03BehaviorParameterization {

public static void main(String[] args) {

List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6);

// Using behavior parameterization to filter and print numbers

filterAndPrint(numbers, x -> x % 2 == 0); // Print even numbers

filterAndPrint(numbers, x -> x % 2 != 0); // Print odd numbers

filterAndPrint(numbers, x -> x % 3 == 0); // Print multiples of 3

}

public static void filterAndPrint(List<Integer> numbers, Predicate<Integer> predicate) {

numbers.stream()

.filter(predicate) // Use the passed predicate for filtering

.forEach(System.out::println); // Print each filtered number

}

}

```

---

### Step 04: Do Exercise with Behavior Parameterization

### Key Objectives:

- Understand how to implement behavior parameterization in Java.

- Learn to create flexible methods that can apply different mathematical operations on a list of numbers.

- Explore the concept of first-class functions and their significance in Java programming.

### Exercise

\*\*Objective\*\*: Get familiar with behavior parameterization through a practical exercise.

- In this exercise, we will focus on behavior parameterization by creating a method that can handle different mathematical operations like squaring, cubing, doubling, or tripling numbers in a list.

- This enables us to pass the desired behavior as an argument, allowing for greater flexibility in our code.

- Previously, we created a list of squared numbers using the following code:

```java

numbers.stream()

.map(x -> x \* x) // Mapping x to its square

.collect(Collectors.toList()); // Collecting the results into a list

```

- Now, the goal is to refactor this code so that we can easily switch between different operations by passing different functions as parameters.

### Refactoring for Behavior Parameterization

\*\*Objective\*\*: Refactor the existing code to introduce behavior parameterization.

- To start, we can define a mappingFunction that encapsulates the behavior we want to apply to each number in the list. For example, to square the numbers:

```java

Function<Integer, Integer> mappingFunction = x -> x \* x; // Function to square numbers

```

- Next, we will extract this logic into a method that takes both the list of numbers and the mapping function as parameters:

```java

public List<Integer> mapAndCreateNewList(List<Integer> numbers, Function<Integer, Integer> mappingFunction) {

return numbers.stream()

.map(mappingFunction) // Using the passed mapping function

.collect(Collectors.toList()); // Collecting the results

}

```

- By extracting this method, we allow for any transformation logic to be passed in, making the method reusable for various operations.

### Implementing Different Operations

\*\*Objective\*\*: Use the new method to create lists of squared, cubed, and doubled numbers.

- With our `mapAndCreateNewList` method in place, we can now easily apply different mathematical transformations to our list of numbers.

- Here’s how we can use the method for squaring, cubing, and doubling:

```java

// Squaring numbers

List<Integer> squaredNumbers = mapAndCreateNewList(numbers, x -> x \* x);

// Cubing numbers

List<Integer> cubedNumbers = mapAndCreateNewList(numbers, x -> x \* x \* x);

// Doubling numbers

List<Integer> doubledNumbers = mapAndCreateNewList(numbers, x -> x + x);

```

- This flexibility allows us to use the same method with different logic just by changing the function we pass in.

- \*\*For example\*\*: To print the doubled numbers, we can do:

```java

System.out.println("Doubled Numbers: " + doubledNumbers);

```

- When we run this code, we see that we can seamlessly switch between different operations without duplicating code.

### The Concept of First-Class Functions

\*\*Objective\*\*: Understand the significance of first-class functions in Java.

- During this exercise, we observe that functions now have a first-class status in Java.

- This means that we can treat functions like any other variable, allowing us to:

- Pass functions as parameters to other methods.

- Store functions in variables.

- Return functions from other methods.

#### For instance, we can create a variable that holds a function:

```java

Function<Integer, Integer> square = x -> x \* x; // Function to square a number

```

- This flexibility is essential for writing more modular and reusable code. It allows us to externalize behavior and apply it dynamically, leading to clearer and more maintainable applications.

#### Full Code:

```java

import java.util.List;

import java.util.function.Function;

import java.util.stream.Collectors;

public class BehaviorParameterizationExample {

public static void main(String[] args) {

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

// Using behavior parameterization to create lists of squared, cubed, and doubled numbers

List<Integer> squaredNumbers = mapAndCreateNewList(numbers, x -> x \* x);

List<Integer> cubedNumbers = mapAndCreateNewList(numbers, x -> x \* x \* x);

List<Integer> doubledNumbers = mapAndCreateNewList(numbers, x -> x + x);

// Printing results

System.out.println("Squared Numbers: " + squaredNumbers);

System.out.println("Cubed Numbers: " + cubedNumbers);

System.out.println("Doubled Numbers: " + doubledNumbers);

}

public static List<Integer> mapAndCreateNewList(List<Integer> numbers, Function<Integer, Integer> mappingFunction) {

return numbers.stream()

.map(mappingFunction) // Using the passed mapping function

.collect(Collectors.toList()); // Collecting the results

}

}

```

---

### Step 05: Exploring Supplier and UnaryOperator Functional Interfaces

### Key Objectives:

- Understand the Supplier functional interface and its applications.

- Learn about the UnaryOperator functional interface and how it operates.

- Explore the significance of functional interfaces in Java and their role in functional programming.

### Introduction to Functional Interfaces

- In prior lessons, we explored several key functional interfaces:

- \*\*Predicate\*\*: Represents a boolean condition based on an input.

- \*\*Function\*\*: Takes an input and produces an output, potentially of a different type.

- \*\*Consumer\*\*: Accepts an input but does not return any output.

- Now, we will expand our knowledge by introducing two more functional interfaces: \*\*Supplier\*\* and \*\*UnaryOperator\*\*.

### Understanding the Supplier Interface

\*\*Objective\*\*: Learn about the Supplier interface, its purpose, and its usage.

- The \*\*Supplier\*\* interface provides a mechanism to generate or supply values without requiring any input. This makes it distinct from other interfaces we've discussed.

- A Supplier is useful in scenarios such as factory patterns, where you want to create objects or generate values without needing any external input.

Here's how we can define a Supplier of integers:

```java

Supplier<Integer> randomIntegerSupplier = () -> 2; // A simple supplier returning the number 2

```

However, a more practical use case would be to return a random number:

```java

Supplier<Integer> randomIntegerSupplier = () -> {

Random random = new Random();

return random.nextInt(10000); // Returns a random integer less than 10000

};

```

- In this example, we create a Supplier that returns a random integer between 0 and 9999. This is executed using:

```java

System.out.println(randomIntegerSupplier.get());

This allows us to retrieve random numbers each time we call get(),

```

- illustrating how Suppliers work in practice.

### Characteristics of Suppliers

- The Supplier interface is straightforward; it has one abstract method, `get()`, which does not take any parameters.

- This makes it ideal for scenarios where you want to create values or objects on demand without requiring any input.

- For instance, in a game, you might use a Supplier to generate random events, enemies, or power-ups without the need for specific parameters each time.

### Understanding the UnaryOperator Interface

- The UnaryOperator interface is a specialization of the Function interface. It represents a function that takes a single input and returns a result of the same type.

- Essentially, it operates on one element and produces a result of the same type.

#### Here’s an example of a UnaryOperator that triples an integer:

```java

UnaryOperator<Integer> tripleOperator = x -> 3 \* x; // Takes an integer and returns its triple

```

We can then use this operator to transform an integer:

```java

System.out.println(tripleOperator.apply(10)); // Outputs 30

```

- This showcases how UnaryOperators can be utilized to transform values while maintaining the same data type.

### Key Features of UnaryOperator

- The UnaryOperator interface is valuable because it simplifies operations on single values while ensuring that the input and output types remain consistent.

- It is particularly useful in scenarios where you need to apply transformations to data within streams or collections.

\*\*For example\*\*: When working with Java streams, you can easily apply UnaryOperators to modify data as it flows through the stream pipeline:

```java

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

List<Integer> tripledNumbers = numbers.stream()

.map(tripleOperator)

.collect(Collectors.toList());

```

- This makes it easy to transform collections with clean and readable code.

### Exploring Java's Functional Interfaces Package

- Both Supplier and UnaryOperator, along with other functional interfaces, are found in the `java.util.function package`.

- This package includes various interfaces designed to work with Java's functional programming features, such as:

- \*\*BiPredicate\*\*: A predicate that takes two inputs.

- \*\*BiFunction\*\*: A function that takes two inputs and produces an output.

- \*\*BiConsumer\*\*: A consumer that takes two inputs and does not return anything.

- These interfaces allow for more complex operations and are helpful when dealing with multiple inputs in a functional style.

#### Full Code:

```java

import java.util.Random;

import java.util.function.Supplier;

import java.util.function.UnaryOperator;

public class FunctionalInterfacesExample {

public static void main(String[] args) {

// Supplier example: generates random integers

Supplier<Integer> randomIntegerSupplier = () -> {

Random random = new Random();

return random.nextInt(10000); // Returns a random integer less than 10000

};

// Testing the supplier

System.out.println("Random Number: " + randomIntegerSupplier.get());

// UnaryOperator example: triples a number

UnaryOperator<Integer> tripleOperator = x -> 3 \* x;

// Testing the unary operator

System.out.println("Triple of 10: " + tripleOperator.apply(10)); // Outputs 30

}

}

```

---

### Step 06: Exploring BiPredicate, BiFunction, BiConsumer and Primitive Functional Interfaces

### Key Objectives:

- Understand the concepts and usage of BiPredicate, BiFunction, and BiConsumer functional interfaces.

- Learn about primitive functional interfaces and their advantages in Java.

- Explore the significance of using primitive types for improved performance.

### Introduction to BiPredicate

\*\*Objective\*\*: Explore the BiPredicate interface and its functionality.

- The \*\*BiPredicate\*\* interface is a specialized version of the Predicate interface that accepts two inputs instead of one.

- The result is still a boolean value, which allows it to evaluate conditions based on two different parameters.

#### For example:

```java

BiPredicate<Integer, String> biPredicate = (number, str) -> number < 10 && str.length() > 5;

```

- In this example, we define a BiPredicate that checks if a number is less than 10 and if the length of a given string is greater than 5.

- We can easily test this BiPredicate with various inputs:

```java

System.out.println(biPredicate.test(5, "in28minutes")); // Prints true

System.out.println(biPredicate.test(15, "test")); // Prints false

```

- This output illustrates how BiPredicate allows us to evaluate more complex conditions involving two variables, showcasing its flexibility in functional programming.

### Understanding BiFunction

- The BiFunction interface takes two inputs and produces a single output.

- Unlike BiPredicate, which always returns a boolean, BiFunction can return any type of output.

- This makes it useful for various scenarios where two inputs are combined into a single result.

#### For example:

```java

BiFunction<Integer, String, String> biFunction = (number, str) -> number + " " + str;

```

- In this case, the BiFunction concatenates an integer and a string into a single string.

- We can apply this BiFunction like so:

```java

System.out.println(biFunction.apply(15, "in28minutes")); // Prints "15 in28minutes"

```

- This demonstrates how BiFunction can be used to manipulate or combine multiple inputs into a meaningful output, providing flexibility in handling data transformations.

### Exploring BiConsumer

- The BiConsumer interface is similar to Consumer but accepts two inputs and does not return a result.

- It is typically used for operations where you want to process two inputs, such as logging or printing.

#### For example:

```java

BiConsumer<Integer, String> biConsumer = (s1, s2) -> {

System.out.println(s1);

System.out.println(s2);

};

```

- This BiConsumer simply prints both inputs.

- To use the BiConsumer:

```java

biConsumer.accept(15, "in28minutes"); // Prints 15 and then in28minutes

```

- This shows that BiConsumer is helpful in scenarios where you want to perform an action on multiple inputs without returning any value.

### Primitive Functional Interfaces

Java also provides primitive versions of functional interfaces, such as:

- IntPredicate

- IntFunction

- IntConsumer

- IntBinaryOperator

These interfaces are designed specifically to operate with primitive types, allowing for more efficient processing without the overhead of boxing and unboxing associated with wrapper classes.

#### For example, instead of using:

```java

BinaryOperator<Integer> intBinaryOperator = (x, y) -> x + y;

```

We can use:

```java

IntBinaryOperator intBinaryOperator = (x, y) -> x + y;

```

- This approach eliminates the need for boxing and improves performance, especially in scenarios involving large datasets or performance-critical applications.

### Advantages of Primitive Functional Interfaces

Using primitive functional interfaces allows for:

- \*\*Improved Performance\*\*: Eliminates boxing and unboxing, reducing memory overhead and improving speed.

- \*\*Simpler Code\*\*: Facilitates the handling of primitive data types directly, making the code cleaner and easier to read.

- For instance, using an IntBinaryOperator directly with primitive int types avoids unnecessary conversion, which can lead to more efficient code execution.

#### Full Code:

```java

import java.util.function.BiPredicate;

import java.util.function.BiFunction;

import java.util.function.BiConsumer;

public class FunctionalInterfacesExample {

public static void main(String[] args) {

// BiPredicate example

BiPredicate<Integer, String> biPredicate = (number, str) -> number < 10 && str.length() > 5;

System.out.println(biPredicate.test(5, "in28minutes")); // Prints true

// BiFunction example

BiFunction<Integer, String, String> biFunction = (number, str) -> number + " " + str;

System.out.println(biFunction.apply(15, "in28minutes")); // Prints "15 in28minutes"

// BiConsumer example

BiConsumer<Integer, String> biConsumer = (s1, s2) -> {

System.out.println(s1);

System.out.println(s2);

};

biConsumer.accept(15, "in28minutes"); // Prints 15 and in28minutes

}

}

```

---

### Step 07: Playing Puzzles with Functional Interfaces and Lambdas

### Key Objectives:

- Understand the syntax and compilation rules for functional interfaces in Java.

- Learn how to work with lambda expressions, including type inference and multi-statement expressions.

- Explore practical examples to clarify common misconceptions related to functional interfaces.

### Introduction to Functional Interfaces

\*\*Objective\*\*: Review the concepts of functional interfaces and introduce puzzles for exploration.

- In this session, we will engage with functional interfaces and lambda expressions through a series of coding puzzles. We'll focus on how these concepts behave during compilation and execution.

- This hands-on approach will help solidify our understanding of these fundamental aspects of functional programming in Java.

### Exploring Consumer Functional Interface

\*\*Objective\*\*: Analyze the Consumer interface and its syntax.

Begin with the \*\*Consumer\*\* interface, which represents an operation that takes a single input and returns no result. For instance, let's define a Consumer for a string:

```java

Consumer<String> consumer = str -> {};

```

- \*\*Empty Parameters\*\*: If you try to define a Consumer without parameters, it will not compile because the Consumer requires one input.

- \*\*Lambda Expression\*\*: You can omit parentheses for a single parameter. However, if there are no statements in the body, it will compile successfully even if the body is empty.

Now, if we replace the empty body with a print statement:

```java

Consumer<String> consumer = str -> System.out.println(str);

```

- This compiles correctly and will print the string passed to it.

### Working with BiConsumer

- Next, we will examine the BiConsumer interface, which accepts two inputs. Here’s how we define it:

```java

BiConsumer<String, String> biConsumer = (s1, s2) -> {

System.out.println(s1);

System.out.println(s2);

};

```

- You must specify both parameters. If you only define one, it will not compile.

- BiConsumer allows us to ignore one of the parameters while still meeting the method signature requirements.

#### Testing the BiConsumer:

```java

biConsumer.accept("Hello", "World"); // Prints "Hello" and "World"

```

- This demonstrates how BiConsumer can process two inputs, similar to Consumer but with added flexibility.

### Exploring Supplier Interface

The Supplier interface is defined as:

```java

Supplier<String> supplier = () -> "Ranga"; // Returns the string "Ranga"

```

If you add braces to create a block:

```java

Supplier<String> supplier = () -> {

return "Ranga"; // Explicit return needed in a block

};

```

- When using braces, an explicit return statement is required.

- If you attempt to define a Supplier without parameters, it should compile as long as you adhere to its method signature (i.e., no input parameters).

### Understanding Multi-Statement Lambda Expressions

If you want to execute multiple statements within a lambda expression, you must encapsulate them in braces:

```java

Consumer<String> consumer = str -> {

System.out.println(str);

System.out.println("Another line");

};

```

- Multi-statement lambdas require braces and a return statement if there’s a return type involved. This makes it essential to understand the structure of lambda expressions when creating more complex operations.

### Type Inference in Lambda Expressions

In Java, type inference automatically determines the types of parameters based on the context:

```java

BinaryOperator<Integer> sum = (x, y) -> x + y; // Type is inferred from the context

```

You can also specify types explicitly:

```java

BinaryOperator<Integer> sum = (Integer x, Integer y) -> x + y; // Explicit types

```

- If you specify the type for one parameter, you must do so for all parameters. If you provide an incorrect type, the compiler will throw an error.

#### For instance:

```java

// This will cause a compilation error

BinaryOperator<Integer> incorrectSum = (String x, Integer y) -> x + y;

```

- This highlights the importance of matching types correctly when using lambda expressions.

#### Full Code:

```java

import java.util.function.Consumer;

import java.util.function.BiConsumer;

import java.util.function.Supplier;

import java.util.function.BinaryOperator;

public class FunctionalInterfacesPuzzles {

public static void main(String[] args) {

// Consumer example

Consumer<String> consumer = str -> System.out.println(str);

consumer.accept("Hello, Consumer!");

// BiConsumer example

BiConsumer<String, String> biConsumer = (s1, s2) -> {

System.out.println(s1);

System.out.println(s2);

};

biConsumer.accept("Hello", "World");

// Supplier example

Supplier<String> supplier = () -> "Ranga";

System.out.println(supplier.get());

// BinaryOperator example

BinaryOperator<Integer> sum = (x, y) -> x + y;

System.out.println("Sum: " + sum.apply(10, 20)); // Outputs "Sum: 30"

}

}

```

---

### Step 08: Exploring Method References with Java

### Key Objectives:

- Understand the concept of method references in Java.

- Learn how to use method references for static methods, instance methods, and constructors.

- Explore practical examples to solidify knowledge of method references.

### Introduction to Method References

- In this session, we will explore method references, a powerful feature in Java that allows us to refer to methods without executing them.

- This provides a more concise and readable way to write code, especially when working with functional interfaces.

### Using Method References with Streams

- Let’s start by examining a basic example using a list of courses. We will create a stream and apply transformations using method references.

#### Here’s how it looks:

```java

courses.stream()

.map(str -> str.toUpperCase()) // Using a lambda expression

.forEach(System.out::println); // Printing each course name

```

#### In this code:

- `map` transforms each course name to uppercase.

- `forEach` prints the uppercase course names.

- Replacing Lambda with Method Reference

We can simplify the map operation using a method reference:

```java

map(String::toUpperCase);

```

This change makes our code cleaner. Method references point directly to existing methods without needing a lambda expression for simple cases.

### Using Static Method References

Next, we can create a static method in our class for printing.

#### Here’s how to define and use it:

```java

private static void print(String str) {

System.out.println(str);

}

```

- `forEach(FP03MethodReferences::print);` // Method reference to static print method

- This code showcases how we can use method references for static methods.

This method will be called for each element in the stream, similar to how we used a lambda expression earlier.

### Using Instance Method References

We can also call instance methods using method references. For example:

```java

map(str -> str.toUpperCase());

```

can be replaced with:

```java

map(String::toUpperCase);

```

Here, we refer directly to the instance method `toUpperCase()` of the String class.

This indicates that the method should be called on each string instance in the stream.

### Constructor References

Constructor references allow us to create new objects without the need for a lambda expression.

\*\*For example\*\*: If we want to create a Supplier for strings:

```java

Supplier<String> stringSupplier = () -> new String(); // Using a lambda

```

We can simplify this using a constructor reference:

```java

Supplier<String> stringSupplier = String::new; // Constructor reference

```

This creates a new instance of the String class whenever `get()` is called on the supplier.

Constructor references provide a clear and concise way to instantiate objects.

In this lecture, we covered the following types of method references:

- \*\*Static Method References\*\*: Referring to static methods of a class.

- \*\*Instance Method References\*\*: Referring to instance methods of an object.

- \*\*Constructor References\*\*: Referring to constructors for creating new objects.

These references enhance code readability and maintainability by reducing boilerplate code.

They also integrate seamlessly with Java’s functional programming capabilities, allowing for cleaner functional style code.

#### Full Code:

```java

import java.util.List;

import java.util.function.Supplier;

public class FP03MethodReferences {

public static void main(String[] args) {

List<String> courses = List.of("Java", "Python", "JavaScript");

// Using method references with streams

courses.stream()

.map(String::toUpperCase) // Method reference for instance method

.forEach(System.out::println); // Method reference for static method

// Using a Supplier with constructor reference

Supplier<String> stringSupplier = String::new; // Constructor reference

System.out.println(stringSupplier.get()); // Prints an empty string

}

private static void print(String str) {

System.out.println(str); // Static method for printing

}

}

```

---

## Java Functional Programming with Custom Classes

### Step 01: Creating Custom Class Course with some Test Data

### Key Objectives:

1. Understand the importance of creating custom classes in Java.

2. Learn how to define class attributes and methods.

3. Explore how to instantiate objects and manage them in a list.

4. Familiarize with using the `toString()` method for object representation.

5. Prepare for advanced operations using streams in future sessions.

### Introduction

Lets begins by transitioning from basic data structures to creating a custom class called `Course`. This class will help organize and manage course-related data effectively.

The focus will be on defining the class, its attributes, and how to create multiple objects from this class.

### Creating the Custom Class

- A new class named `Course` is created, which will encapsulate all relevant details about a specific course. This practice is fundamental in object-oriented programming, as it helps structure code and data into manageable entities.

### Defining Class Attributes

Inside the `Course` class, several attributes are defined:

- A `String` variable named `name` is added to hold the name of the course.

- A `String` variable for the course's `category` indicates the area it belongs to.

- An `int` variable called `reviewScore` is introduced to represent the course's review score, which can range from 0 to 100.

- An `int` variable for `numberOfStudents` is included to track how many students are enrolled in the course.

#### Code:

```java

private String name; // Course name

private String category; // Course category

private int reviewScore; // Review score (0 to 100)

private int numberOfStudents; // Number of students enrolled

```

### Generating Getters and Setters

- Next, getters and setters are generated for the attributes of the Course class.

- This allows easy access and modification of the class properties. Getters provide read access, while setters allow changes to the property values.

#### Code:

```java

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

// Repeat for other attributes

```

### Adding a Constructor

- A constructor is added to the Course class, which initializes the attributes of the course when a new object is created. The constructor takes parameters for each attribute, ensuring that every Course object is properly initialized with relevant data.

#### Code:

```java

public Course(String name, String category, int reviewScore, int numberOfStudents) {

this.name = name;

this.category = category;

this.reviewScore = reviewScore;

this.numberOfStudents = numberOfStudents;

}

```

### Overriding the toString Method

- The `toString()` method is overridden to provide a meaningful string representation of the Course objects.

- This method is crucial for displaying information about a course in a human-readable format when an object is printed.

#### Code:

```java

@Override

public String toString() {

return name + " - Students: " + numberOfStudents + ", Review Score: " + reviewScore;

}

```

### Instantiating Course Objects

- With the Course class defined, several instances of the Course class are created using the `List.of()` method.

- This allows for the management of multiple course objects in a list format, making it easier to manipulate and access course data.

#### Code:

```java

List<Course> courses = List.of(

new Course("Spring", "Framework", 98, 20000),

new Course("Spring Boot", "Framework", 95, 18000),

new Course("Microservices", "Microservices", 97, 22000),

new Course("FullStack", "FullStack", 96, 25000),

new Course("AWS", "Cloud", 92, 21000),

new Course("Azure", "Cloud", 91, 14000),

new Course("Docker", "Cloud", 90, 20000),

new Course("Kubernetes", "Cloud", 89, 20000)

);

```

- After creating the list of course objects, the details of each course can be accessed and manipulated easily.

---

### Step 02: Playing with allMatch, noneMatch and anyMatch

### Key Objectives:

1. Understand the functionality of `allMatch()`, `noneMatch()`, and `anyMatch()` in Java Streams.

2. Learn how to create predicates to check specific conditions on course data.

3. Explore practical examples to illustrate the use of these stream methods.

### Introduction

In this lecture, the focus is on using Java Stream methods: `allMatch()`, `noneMatch()`, and `anyMatch()`.

These methods allow for checking conditions across a collection of objects, specifically, the list of courses created in the previous lecture.

### Understanding `allMatch()`

- The first method explored is `allMatch()`. This method checks whether all elements in a stream match a given predicate.

- The goal is to find out if all courses have a review score greater than 90.

#### To implement this:

- The stream of courses is initiated using `courses.stream()`.

- A predicate is defined to check if the review score of each course is greater than 90 using `course.getReviewScore() > 90`.

- The result is printed to the console.

#### Code:

```java

boolean allGreaterThan90 = courses.stream().allMatch(course -> course.getReviewScore() > 90);

System.out.println(allGreaterThan90); // Expected output: true

```

- When executed, the output confirms that all courses meet this criterion.

- Next, the check is modified to see if all courses have a review score greater than 95.

#### Code:

```java

boolean allGreaterThan95 = courses.stream().allMatch(course -> course.getReviewScore() > 95);

System.out.println(allGreaterThan95); // Expected output: false

```

The output is `false`, indicating that not all courses have a review score exceeding 95.

### Creating Predicates

To make the code cleaner and more maintainable, predicates for the review score conditions are created:

- `reviewScoreGreaterThan90Predicate`: Checks if the review score is greater than 90.

- `reviewScoreGreaterThan95Predicate`: Checks if the review score is greater than 95.

#### Code:

```java

Predicate<Course> reviewScoreGreaterThan90Predicate = course -> course.getReviewScore() > 90;

Predicate<Course> reviewScoreGreaterThan95Predicate = course -> course.getReviewScore() > 95;

```

These predicates can then be reused in the stream methods for better readability.

### Understanding `noneMatch()`

- The next method covered is `noneMatch()`, which checks if no elements in the stream match a given condition.

- The implementation tests whether none of the courses have a review score greater than 95.

#### Code:

```java

boolean noneGreaterThan95 = courses.stream().noneMatch(reviewScoreGreaterThan95Predicate);

System.out.println(noneGreaterThan95); // Expected output: false

```

The output is `false`, indicating that at least one course has a review score greater than 95.

- To demonstrate a case where `noneMatch()` returns true, a new predicate is created to check if the review score is less than 90.

#### Code:

```java

Predicate<Course> reviewScoreLessThan90Predicate = course -> course.getReviewScore() < 90;

boolean noneLessThan90 = courses.stream().noneMatch(reviewScoreLessThan90Predicate);

System.out.println(noneLessThan90); // Expected output: true

```

This output is `true`, confirming that no courses have a review score below 90.

### Understanding `anyMatch()`

Finally, the method `anyMatch()` is explored. This method checks if at least one element in the stream matches the specified predicate.

A test is performed to see if any courses have a review score greater than 90.

#### Code:

```java

boolean anyGreaterThan90 = courses.stream().anyMatch(reviewScoreGreaterThan90Predicate);

System.out.println(anyGreaterThan90); // Expected output: true

```

This returns true because all courses meet this criterion. Similarly, checking against the `reviewScoreGreaterThan95Predicate` also yields true since some courses meet the condition.

#### Code:

```java

boolean anyGreaterThan95 = courses.stream().anyMatch(reviewScoreGreaterThan95Predicate);

System.out.println(anyGreaterThan95); // Expected output: true

```

- `allMatch()` returns true if every element in the stream matches the condition.

- `noneMatch()` returns true if no elements match the specified condition.

- `anyMatch()` returns true if at least one element matches the condition.

An exercise is suggested for learners to create additional predicates based on the number of students and practice using `allMatch()`, `noneMatch()`, and `anyMatch()` with these new conditions.

---

### Step 03: Sorting courses with sorted and creating Comparators

### Key Objectives:

1. Understand how to sort a list of objects using Java Streams.

2. Learn to create custom comparators to define sorting criteria.

3. Explore sorting with multiple criteria, including handling primitive data types efficiently.

### Introduction

In this lecture, the focus is on sorting a list of courses using predefined criteria.

The primary method utilized for sorting is the `sorted()` method from Java Streams, which requires a comparator to define how the sorting should be done.

### Sorting Courses

- To start sorting the courses, the `courses.stream().sorted()` method is called. This method will sort the list based on a provided comparator, which needs to be defined.

#### Code:

```java

courses.stream().sorted(/\* comparator here \*/);

```

### Creating a Comparator by Number of Students

- The next step involves defining a comparator that sorts the courses based on the number of students enrolled. The Comparator interface from `java.util` is used to facilitate this.

- A new comparator is created and named comparingByNumberOfStudents.

- The comparator is set up to compare courses by their number of students using the `Comparator.comparing()` method.

#### Code:

```java

Comparator<Course> comparingByNumberOfStudents = Comparator.comparing(Course::getNumberOfStudents);

```

- Once defined, this comparator is passed to the `sorted()` method to sort the courses.

#### Code:

```java

List<Course> sortedCourses = courses.stream()

.sorted(comparingByNumberOfStudents)

.collect(Collectors.toList());

```

- After running the code, the courses are displayed in increasing order based on the number of students enrolled, providing a clear output for verification.

### Sorting in Decreasing Order

- To sort the courses in decreasing order, the previous comparator is modified by using the `reversed()` method.

- A new comparator named `comparingByNumberOfStudentsDecreasing` is created for this purpose.

#### Code:

```java

Comparator<Course> comparingByNumberOfStudentsDecreasing = comparingByNumberOfStudents.reversed();

```

- The same sorting logic is applied, and running the code now displays the courses in decreasing order of the number of students.

### Sorting with Multiple Criteria

- When courses have the same number of students, a secondary sorting criterion is introduced: the review score.

- A new comparator is created that first compares by the number of students and then by the review score.

#### Code:

```java

Comparator<Course> comparingByNumberOfStudentsAndReviews =

Comparator.comparing(Course::getNumberOfStudents)

.thenComparing(Course::getReviewScore);

```

- By using `thenComparing()`, the courses are now sorted by the number of students, and in cases where there are ties, they are further sorted by the review score in ascending order.

- After running this code, courses with the same number of students are displayed with the one having the higher review score appearing first.

### Efficiently Handling Primitive Types

While sorting, it’s noted that when dealing with primitive types (like int), it’s more efficient to use specialized comparator methods such as `comparingInt()`.

This avoids the overhead of boxing and unboxing.

#### Code:

```java

Comparator<Course> comparingByNumberOfStudentsInt = Comparator.comparingInt(Course::getNumberOfStudents);

```

- The same efficiency can be applied to `thenComparing()` for secondary criteria as well.

---

### Step 04: Playing with skip, limit, takeWhile and dropWhile

### Key Objectives:

1. Understand the utility methods `skip()`, `limit()`, `takeWhile()`, and `dropWhile()` in Java Streams.

2. Learn how to filter and manipulate streams based on specific criteria.

3. Explore practical examples of using these methods to manage lists of data.

### Introduction

In this lecture, we will explore additional utility methods available in Java Streams: `skip`, `limit`, `takeWhile`, and `dropWhile`.

These methods allow us to control which elements we process from a stream based on defined criteria, making it easier to handle large datasets efficiently.

### Using the limit() Method

- Lets begins by introducing the `limit()` method. This method restricts the number of elements returned from a stream. For example, if we only want to focus on the top five courses after sorting them, we can utilize `limit()` to narrow down our results.

- By applying `limit(5)`, we can effectively filter out any unnecessary data and concentrate on the most relevant entries.

- This is particularly useful in scenarios where we want to display only a subset of results, such as in a user interface where displaying all courses might overwhelm the user.

### Code:

```java

List<Course> topFiveCourses = courses.stream()

.sorted(comparingByNumberOfStudentsAndReviews)

.limit(5)

.collect(Collectors.toList());

System.out.println(topFiveCourses);

```

When executed, this will output the top five courses based on the sorting criteria. For instance:

#### Output:

```java

[Microservices:25000:96, API:22000:97, Azure:21000:92, AWS:21000:92, Spring:20000:98]

```

### Using the `skip()` Method

- Next, we explore the `skip()` method. This method allows us to skip a specified number of elements from the beginning of the stream.

- For instance, if we want to ignore the top three results, we can easily achieve this with `skip(3)`.

- This is useful in cases where we might want to bypass certain entries, such as when displaying results that follow a pagination structure, where the first few items have already been viewed.

#### Code:

```java

List<Course> skippedCourses = courses.stream()

.skip(3)

.collect(Collectors.toList());

System.out.println(skippedCourses);

```

When this is run, the output will show that the first three courses are skipped:

#### Output:

```java

[AWS:21000:92, Spring:20000:98, Docker:20000:92, Kubernetes:20000:91, Spring boot:18000:95, FullStack:14000:91]

```

### Combining `skip()` and `limit()`

- It is also possible to combine `skip()` and `limit()` to create more complex queries.

- For instance, if you want to skip the first three courses and then limit the results to the next five, you can chain these methods together seamlessly.

- This combination provides a powerful way to control exactly which elements are included in your final output, giving you greater flexibility in how you present data.

#### Code:

```java

List<Course> nextFiveAfterSkipping = courses.stream()

.skip(3)

.limit(5)

.collect(Collectors.toList());

System.out.println(nextFiveAfterSkipping);

```

The output shows the next five courses starting from the fourth course:

#### Output:

```java

[AWS:21000:92, Spring:20000:98, Docker:20000:92, Kubernetes:20000:91, Spring boot:18000:95]

```

### Using `takeWhile()` Method

- Moving on, the `takeWhile()` method is introduced. This method allows you to take elements from the stream as long as a specified condition is true.

- \*\*For example\*\*: We might want to take courses until we find one with a review score less than 95.

- The utility of `takeWhile()` lies in its ability to stop processing as soon as a condition is no longer met.

- This can significantly improve performance by avoiding unnecessary checks after the first non-matching element is found.

#### Code:

```java

List<Course> coursesWithHighReviews = courses.stream()

.takeWhile(course -> course.getReviewScore() >= 95)

.collect(Collectors.toList());

System.out.println(coursesWithHighReviews);

```

When executed, this outputs the courses with review scores above 95, stopping at the first course that does not meet this condition:

#### output:

```java

[Spring:20000:98, Spring boot:18000:95, API:22000:97, Microservices:25000:96]

```

### Using dropWhile() Method

- The lecture then covers the `dropWhile()` method, which is the opposite of takeWhile(). It skips elements as long as the specified condition is true and takes all subsequent elements.

- This method is particularly useful when you want to ignore initial entries that do not meet certain criteria.

- \*\*For example\*\*: If you want to ignore all courses with a review score above 95 and start processing from the first course that meets this criterion, `dropWhile()` will help you achieve that.

#### Code:

```java

List<Course> droppedCourses = courses.stream()

.dropWhile(course -> course.getReviewScore() >= 95)

.collect(Collectors.toList());

System.out.println(droppedCourses);

```

When this is run, it will skip courses until it finds one that does not meet the condition and then return all following courses:

#### Output:

```java

[FullStack:14000:91, AWS:21000:92, Azure:21000:99, Docker:20000:92, Kubernetes:20000:91]

```

- `limit(n)`: Restricts the stream to the first n elements, allowing for a focused subset of data.

- `skip(n)`: Skips the first n elements of the stream, useful for pagination or excluding certain entries.

- `takeWhile(predicate)`: Takes elements as long as the predicate is true, stopping as soon as a condition is not met.

- `dropWhile(predicate)`: Drops elements as long as the predicate is true, returning all subsequent elements thereafter.

These methods can be very useful for managing collections of data effectively, especially when working with large datasets or when performance is a concern.

---

### Step 05: Finding top, max and min courses with max, min, findFirst and findAny

### Key Objectives:

1. Understand how to use `max()`, `min()`, `findFirst()`, and `findAny()` methods in Java Streams.

2. Learn about the significance of the `Optional` type and how to handle the absence of values.

3. Explore practical examples of finding specific courses based on defined criteria.

### Introduction

In this lecture, the focus shifts to retrieving individual elements from a list of courses. We will explore methods such as `max()`, `min()`, `findFirst()`, and `findAny()` to identify top courses based on specific criteria, such as the number of students or review scores.

Understanding how to leverage these methods can greatly enhance data retrieval and analysis in Java applications.

### Finding the Maximum Course

- Lets begins by discussing how to find the course with the maximum value based on defined criteria. The approach starts with invoking the `stream()` method on the list of courses.

- To find the maximum course, the `max()` method is utilized, which requires a comparator that defines how to evaluate the courses.

- The comparator will compare courses based on both the number of students and review scores.

#### Code:

```java

Optional<Course> maxCourse = courses.stream()

.max(comparingByNumberOfStudentsAndReviews);

System.out.println(maxCourse);

```

When executed, this code identifies the course with the maximum values according to the defined criteria.

#### Output:

```

Optional[FullStack:14000:91]

```

This indicates that FullStack is the course with the highest values according to the comparator used.

- It's important to note that the result of the `max()` operation is wrapped in an Optional.

- This is a container object which may or may not contain a value, providing a way to avoid NullPointerExceptions.

- If no course meets the criteria, `Optional.empty` is returned.

### Finding the Minimum Course

Next, lets explore how to find the course with the minimum value using the `min()` method. Similar to `max()`, the `min()` method also requires a comparator.

#### Code:

```java

Optional<Course> minCourse = courses.stream()

.min(comparingByNumberOfStudentsAndReviews);

System.out.println(minCourse);

```

When executed, this will return the course with the lowest values:

```java

Optional[Microservices:25000:96]

```

In this case, Microservices is the course with the minimum value based on the defined criteria.

### Handling Empty Results with Optional

To demonstrate the handling of an empty result, a predicate that filters out courses with review scores less than 90 is applied.

If no courses meet this condition, the `min()` operation returns Optional.empty.

#### Code:

```java

Optional<Course> noMinCourse = courses.stream()

.filter(course -> course.getReviewScore() < 90)

.min(comparingByNumberOfStudentsAndReviews);

System.out.println(noMinCourse); // Output: Optional.empty

```

In this case, the output confirms that there are no courses with a review score below 90:

#### Output:

```

Optional.empty

```

### Providing Default Values

- To address cases where a value may not be present, you can specify a default value using the `orElse()` method.

- This allows you to define what should be returned if no valid result is found.

#### Code:

```java

Course defaultCourse = new Course("Default Course", "N/A", 0, 0);

Course resultCourse = noMinCourse.orElse(defaultCourse);

System.out.println(resultCourse); // Output: Default Course

```

If no minimum course is found, the default course is returned instead.

### Finding the First Element

The `findFirst()` method is introduced as a way to retrieve the first element from a stream that matches a specific criterion.

For instance, if you want to find the first course with a review score below 90, you would use this method.

#### Code:

```java

Optional<Course> firstLowScoreCourse = courses.stream()

.filter(course -> course.getReviewScore() < 90)

.findFirst();

System.out.println(firstLowScoreCourse); // Output: Optional.empty

```

This operation returns an Optional containing the first course that meets the criterion or `Optional.empty` if none exists.

### Using `findAny()`

Finally, the `findAny()` method is discussed. This method returns any element from the stream that matches a specified condition.

Unlike `findFirst()`, which always returns the first match, `findAny()` may return any qualifying element in a non-deterministic manner, especially when working with parallel streams.

#### Code:

```java

Optional<Course> anyHighScoreCourse = courses.stream()

.filter(course -> course.getReviewScore() > 95)

.findAny();

System.out.println(anyHighScoreCourse); // Output: Optional[Spring:21000:96]

```

This demonstrates that `findAny()` can return any course that matches the criteria, providing flexibility in how you retrieve data.

- `max()`: Finds the maximum element according to a comparator.

- `min()`: Finds the minimum element according to a comparator.

- `findFirst()`: Retrieves the first element that matches a given predicate.

- `findAny()`: Returns any element that matches the criteria in a potentially non-deterministic manner.

Understanding the use of Optional is crucial for robust programming, helping to avoid null-related errors.

---

### Step 06: Playing with sum, average and count

### Key Objectives:

1. Learn how to use `sum()`, `average()`, and `count()` methods in Java Streams.

2. Understand how to aggregate data from a collection of objects.

3. Explore the use of primitive mapping methods (`mapToInt`) for efficient performance.

4. Learn how to handle optional results when calculating averages.

### Introduction

In this lecture, we explore methods that return single aggregate results from a stream of data.

Specifically, we will look at how to calculate the sum, average, and count of students in courses that meet a specific condition—courses with a review score greater than 95.

### Calculating the Total Number of Students with `sum()`

- The first operation demonstrated is how to calculate the total number of students in courses that have a review score greater than 95.

- To achieve this, the process begins by filtering the courses using a predicate, and then mapping the filtered courses to the number of students using the `mapToInt()` method.

- This method is chosen because it efficiently handles primitive data types without boxing or unboxing.

Once the courses are mapped to the number of students, the `sum()` method is applied to get the total number of students.

#### Code:

```java

int totalStudents = courses.stream()

.filter(course -> course.getReviewScore() > 95)

.mapToInt(Course::getNumberOfStudents)

.sum();

System.out.println(totalStudents);

```

This code returns the total number of students in the courses that meet the condition:

#### Output:

```java

88000

```

### Finding the Average Number of Students with `average()`

- Next, the `average()` method is used to find the average number of students in courses with a review score greater than 95.

- Since `average()` might not always return a result (e.g., if there are no courses that meet the condition), it returns an `OptionalDouble`

- This helps avoid `NullPointerException` and provides a way to handle missing values.

#### Code:

```java

OptionalDouble averageStudents = courses.stream()

.filter(course -> course.getReviewScore() > 95)

.mapToInt(Course::getNumberOfStudents)

.average();

System.out.println(averageStudents);

```

The output shows the average number of students per course:

#### Output:

```java

OptionalDouble[22000.0]

```

### Handling Optional Results

If there are no matching courses, `average()` would return an `Optional.empty`.

This can be handled using methods like `orElse()` to provide a default value in such cases:

```java

double avgStudents = averageStudents.orElse(0.0);

System.out.println(avgStudents);

```

### Counting the Number of Courses with `count()`

- To find out how many courses meet the condition of having a review score greater than 95, the `count()` method is used.

- This method is straightforward and returns the total number of elements that match the criteria.

#### Code:

```java

long count = courses.stream()

.filter(course -> course.getReviewScore() > 95)

.count();

System.out.println(count);

```

The output shows the number of courses that meet the condition:

#### Output:

```java

4

```

### Finding Maximum and Minimum Number of Students

- The lecture continues by showing how to find the course with the maximum and minimum number of students using `max()` and `min()`.

- These methods return the maximum and minimum values as `OptionalInt`.

To find the course with the highest number of students in courses with a review score greater than 95, the `max()` method is used:

```java

OptionalInt maxStudents = courses.stream()

.filter(course -> course.getReviewScore() > 95)

.mapToInt(Course::getNumberOfStudents)

.max();

System.out.println(maxStudents);

```

The output will show the maximum number of students in such courses:

#### Output:

```java

OptionalInt[25000]

```

Similarly, the `min()` method can be used to find the course with the minimum number of students:

```java

OptionalInt minStudents = courses.stream()

.filter(course -> course.getReviewScore() > 95)

.mapToInt(Course::getNumberOfStudents)

.min();

System.out.println(minStudents);

```

### Flexibility with Stream Methods

- With Java Streams, it's possible to apply various filters, map data to any attribute (such as category or review score), and retrieve aggregate results like sum, average, and count.

- For example, instead of mapping to the number of students, we could map the stream to review scores and calculate their sum, average, or count. The possibilities are endless when working with streams and custom filters.

- `sum()`: Calculates the total of a mapped stream of numbers.

- `average()`: Returns the average of the numbers in the stream as an OptionalDouble.

- `count()`: Returns the number of elements in the stream that meet a certain condition.

- `max()` and `min()`: Return the maximum and minimum values in the stream as `OptionalInt`.

Understanding and applying these methods allows for powerful and flexible data analysis in Java.

---

### Step 07: Grouping courses into Map using groupingBy

### Key Objectives:

1. Learn how to group a list of objects into a `Map` using the `Collectors.groupingBy()` method in Java Streams.

2. Understand how to perform advanced grouping operations such as counting and finding the maximum value within each group.

3. Explore how to map specific properties (like course names) rather than entire objects to the grouped results.

### Introduction

In this lecture, we explore how to group a list of courses into a `Map` based on specific criteria using the `Collectors.groupingBy()` method.

Grouping allows us to organize data into categories and perform various aggregate operations on each group, such as counting the number of items or finding the maximum value.

### Grouping Courses by Category

- The first task is to group courses by their category.

- For example, we want to group all courses related to "Framework," "Microservices," "Cloud," etc., and store them in a `HashMap` where the category is the key and the list of courses under that category is the value.

To do this, we use the `groupingBy()` method in the `Collectors` utility class.

#### Code:

```java

Map<String, List<Course>> coursesByCategory = courses.stream()

.collect(Collectors.groupingBy(Course::getCategory));

System.out.println(coursesByCategory);

```

This code groups courses by category, and when executed, the output will show something like:

#### Output:

```java

{

Cloud=[AWS:21000:92, Azure:21000:99, Docker:20000:92, Kubernetes:20000:91],

FullStack=[FullStack:14000:91],

Microservices=[API:22000:97, Microservices:25000:96],

Framework=[Spring:20000:98, Spring boot:18000:95]

}

```

### Counting Courses in Each Category

- Next, instead of listing all the courses under each category, we want to count how many courses belong to each category.

- The `Collectors.counting()` method can be used to count the number of courses for each key (category) in the map.

#### Code:

```java

Map<String, Long> courseCountByCategory = courses.stream()

.collect(Collectors.groupingBy(Course::getCategory, Collectors.counting()));

System.out.println(courseCountByCategory);

```

The output will show the number of courses in each category:

```java

{

Cloud=4,

FullStack=1,

Microservices=2,

Framework=2

}

```

### Finding the Course with the Highest Review Score in Each Category

- The next step is to find the course with the highest review score within each category. Instead of counting, we use `Collectors.maxBy() to identify the course with the maximum review score in each group.

- We need to specify a comparator that compares courses based on their review scores.

#### Code:

```java

Map<String, Optional<Course>> highestRatedCourseByCategory = courses.stream()

.collect(Collectors.groupingBy(

Course::getCategory,

Collectors.maxBy(Comparator.comparing(Course::getReviewScore))

));

System.out.println(highestRatedCourseByCategory);

```

When executed, the output will display the course with the highest review score for each category:

```java

{

Cloud=Optional[Azure:99],

FullStack=Optional[FullStack:91],

Microservices=Optional[API:97],

Framework=Optional[Spring:98]

}

```

### Extracting Only the Course Names

- Rather than returning the entire Course object, we might only want the name of the highest-rated course for each category.

- This can be achieved using `Collectors.mapping()`, which allows us to transform the grouped elements before collecting them. Here, we map the courses to their names.

#### Code:

```java

Map<String, List<String>> courseNamesByCategory = courses.stream()

.collect(Collectors.groupingBy(

Course::getCategory,

Collectors.mapping(Course::getName, Collectors.toList())

));

System.out.println(courseNamesByCategory);

```

The output will show the course names grouped by category:

```java

{

Cloud=[AWS, Azure, Docker, Kubernetes],

FullStack=[FullStack],

Microservices=[API, Microservices],

Framework=[Spring, Spring Boot]

}

```

### Mapping Highest Rated Course Name by Category

- To refine the previous example, we can combine `mapping()` and `maxBy()` to return only the names of the highest-rated courses in each category.

- This example shows how to chain different collectors to get specific results.

#### Code:

```java

Map<String, Optional<String>> highestRatedCourseNameByCategory = courses.stream()

.collect(Collectors.groupingBy(

Course::getCategory,

Collectors.mapping(Course::getName,

Collectors.maxBy(Comparator.comparing(Course::getReviewScore))

)

));

System.out.println(highestRatedCourseNameByCategory);

```

The output will show only the names of the highest-rated courses by category:

```java

{

Cloud=Optional[Azure],

FullStack=Optional[FullStack],

Microservices=Optional[API],

Framework=Optional[Spring]

}

```

---

## Playing with Java Functional Programming

### Step 01: Creating Streams using Stream of method and for Arrays

### Key Objectives:

1. Learn how to create streams directly using `Stream.of()`.

2. Understand the difference between streams created from lists and those created from arrays.

3. Explore the benefits of using primitive streams for performance efficiency.

4. Practice common operations like sum, average, min, and max on both wrapper class and primitive streams.

### Introduction

In this lecture, we focus on how to create streams directly without needing to rely on pre-existing lists. There are multiple options in Java for creating streams, including `Stream.of()` and `Arrays.stream()`.

These options provide flexibility in how we create and manipulate streams, particularly when working with primitive data types, which can offer performance benefits.

### Creating Streams using `Stream.of()`

- To begin, we explore how to create a stream directly from a set of values using the `Stream.of()` method.

- This allows us to quickly initialize a stream with specific elements without first needing to create a list.

#### Code:

```java

Stream<Integer> numberStream = Stream.of(12, 9, 13, 4, 6, 2, 4, 12, 15);

System.out.println(numberStream.count());

```

This code snippet creates a stream from a series of integers and then counts the number of elements in the stream. The output will be:

```java

9

```

### Summing Elements with Reduce

- In addition to counting, we can perform operations on the stream.

- For example, using the `reduce()` method, we can calculate the sum of the stream’s elements. The `reduce()` method applies a binary operator to combine all elements into a single result.

#### Code:

```java

int sum = Stream.of(12, 9, 13, 4, 6, 2, 4, 12, 15)

.reduce(0, Integer::sum);

System.out.println(sum);

```

This code sums all the elements in the stream, and the output will be:

```java

77

```

### Wrapper Classes in Streams

- One important thing to note is that streams created using `Stream.of()` store elements as instances of wrapper classes (e.g., Integer for int values).

- This can lead to boxing and unboxing during operations, which can be inefficient when dealing with large datasets.

- \*\*Boxing\*\*: Converting a primitive type (e.g., int) into a wrapper class object (e.g., Integer).

- \*\*Unboxing\*\*: Converting a wrapper class object back into a primitive type.

- While `Stream.of()` is simple to use, there are more efficient ways to create streams of primitive types that avoid boxing and unboxing overhead.

### Creating Streams from Arrays

- To avoid boxing and unboxing inefficiencies, Java provides an option to create streams of primitive values directly from arrays using `Arrays.stream()`.

- This creates streams that work with primitive types (e.g., int, double, long) and allows for more efficient operations.

Let’s look at how to create a stream from an array of primitive int values.

#### Code:

```java

int[] numberArray = {12, 9, 13, 4, 6, 2, 4, 12, 15};

IntStream intStream = Arrays.stream(numberArray);

System.out.println(intStream.sum());

```

This code creates a stream from an array of integers and calculates the sum of the elements.

The output will be:

```java

77

```

### Primitive vs. Wrapper Class Streams

A key distinction between streams created from arrays and those created from `Stream.of()` is the type of stream generated:

- Streams created from arrays (e.g., IntStream) are primitive streams that directly hold primitive values (int, long, etc.).

- Streams created from `Stream.of()` are streams of wrapper classes (e.g., `Stream<Integer>`).

- Primitive streams (like IntStream) are more efficient because they avoid the overhead of boxing and unboxing.

### Performing Operations on Primitive Streams

- Primitive streams, such as IntStream, provide several useful operations like `sum()`, `average()`, `min()`, and `max()` directly, without needing extra conversions.

- These operations return results wrapped in optional types to handle cases where there might not be any values in the stream.

#### Code:

```java

int[] numberArray = {12, 9, 13, 4, 6, 2, 4, 12, 15};

IntStream intStream = Arrays.stream(numberArray);

// Perform common operations

int sum = intStream.sum();

OptionalDouble average = Arrays.stream(numberArray).average();

OptionalInt min = Arrays.stream(numberArray).min();

OptionalInt max = Arrays.stream(numberArray).max();

System.out.println("Sum: " + sum); // Sum: 77

System.out.println("Average: " + average); // Average: OptionalDouble[8.555555555555555]

System.out.println("Min: " + min); // Min: OptionalInt[2]

System.out.println("Max: " + max); // Max: OptionalInt[15]

```

The output will show the sum, average, minimum, and maximum values:

```java

Sum: 77

Average: OptionalDouble[8.555555555555555]

Min: OptionalInt[2]

Max: OptionalInt[15]

```

---

### Step 02: Creating Streams for First 100 Numbers, Squares of Numbers and More

### Key Objectives:

1. Learn how to create streams of primitive values dynamically using `IntStream`.

2. Understand the difference between `range()`, `rangeClosed()`, and `iterate()` methods for generating sequences.

3. Explore the use of `peek()` for inspecting stream elements.

4. Perform advanced exercises, such as generating even numbers and powers of two.

5. Learn how to convert primitive streams to collections using the `boxed()` method.

### Introduction

In this lecture, we explore how to create dynamic streams of primitive values using `IntStream`. We will cover techniques for generating streams of numbers, performing operations on those streams, and converting primitive streams to more general collections like lists.

This approach is useful for generating sequences like the first 100 numbers, even numbers, odd numbers, and powers of two.

### Creating Streams with `IntStream.range()`

- To generate a sequence of numbers, we can use the `IntStream.range()` method, which allows us to create a stream of numbers within a specified range.

- This method is useful when you want to generate a range of sequential integers.

```java

int sum = IntStream.range(1, 10).sum();

System.out.println(sum);

```

- `IntStream.range(1, 10)` generates a stream of integers from 1 to 9 (excluding 10).

- The `sum()` method adds all the elements in the stream.

#### Output:

```java

45

```

### Using `rangeClosed()` to Include the Last Element

- If you want to include the last element in the range, you can use `IntStream.rangeClosed()`.

- This method is similar to `range()` but includes the last number in the sequence.

```java

int sum = IntStream.rangeClosed(1, 10).sum();

System.out.println(sum);

```

- `IntStream.rangeClosed(1, 10)` generates a stream of integers from 1 to 10.

#### Output:

```java

55

```

### Creating Dynamic Streams with `IntStream.iterate()`

- Sometimes, we need to generate numbers based on a specific algorithm rather than a sequential range.

- The `IntStream.iterate()` method allows you to specify a starting value and a function to generate subsequent values.

- This is particularly useful for generating even numbers, odd numbers, or any other pattern.

\*\*Example\*\*: Generating an Infinite Stream of Odd Numbers

```java

IntStream.iterate(1, n -> n + 2)

.limit(10)

.forEach(System.out::println);

```

- `IntStream.iterate(1, n -> n + 2)` starts with 1 and increments by 2, generating odd numbers.

- The `limit(10)` method limits the stream to the first 10 odd numbers.

#### Output:

```java

1

3

5

7

9

11

13

15

17

19

```

### Using `peek()` to Inspect Stream Elements

- The `peek()` method allows you to inspect the elements of a stream without modifying the stream itself.

- It’s useful for debugging or for understanding what values are being processed in the stream.

\*\*Example\*\*: Printing the Elements of a Stream Using peek()

```java

IntStream.iterate(1, n -> n + 2)

.limit(10)

.peek(System.out::println)

.sum();

```

- `peek(System.out::println)` prints the elements in the stream before performing the sum operation.

- The sum is calculated, but it is secondary to the peek output, which is the same list of odd numbers as before:

```java

1

3

5

7

9

11

13

15

17

19

```

### \*\*Exercise\*\*: Printing the First 10 Even Numbers

- As an exercise, let’s generate the first 10 even numbers by modifying the `IntStream.iterate()` method.

\*\*Solution\*\*:

```java

IntStream.iterate(2, n -> n + 2)

.limit(10)

.peek(System.out::println)

.sum();

```

- This stream starts at 2 and increments by 2, producing even numbers.

#### Output:

```java

2

4

6

8

10

12

14

16

18

20

```

### \*\*Exercise\*\*: Calculating the Powers of 2

- Another exercise involves generating the first 10 powers of 2 (2^1, 2^2, 2^3, etc.).

\*\*Solution\*\*:

```java

IntStream.iterate(2, n -> n \* 2)

.limit(10)

.peek(System.out::println)

.sum();

```

- The stream starts at 2 and multiplies each number by 2, producing powers of 2.

#### Output:

```java

2

4

8

16

32

64

128

256

512

1024

```

### Converting Primitive Streams to Lists Using `boxed()`

- While working with primitive streams (IntStream, DoubleStream, LongStream), it is important to note that you cannot directly collect primitive streams into a List.

- To do this, you need to convert the primitive stream into a stream of wrapper objects (e.g., Integer for IntStream). This can be done using the `boxed()` method.

\*\*Example\*\*: Collecting Powers of 2 into a List

```java

List<Integer> powersOfTwo = IntStream.iterate(2, n -> n \* 2)

.limit(10)

.boxed()

.collect(Collectors.toList());

System.out.println(powersOfTwo);

```

- The `boxed()` method converts the primitive IntStream into a `Stream<Integer>`, which can then be collected into a List.

#### Output:

```java

[2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]

```

---

### Step 03: Doing Big Number calculations with BigInteger

### Key Objectives:

1. Understand the limitations of primitive numeric types like `int` and `long` in Java.

2. Learn how to perform calculations involving extremely large values using `BigInteger`.

3. Explore how to use `BigInteger` in conjunction with Java Streams to calculate large factorials and other big number operations.

### Introduction

In this lecture, we explore how to handle calculations involving very large numbers that exceed the limits of Java’s primitive numeric types such as `int` and `long`.

Java provides the `BigInteger` class for such cases, allowing us to perform arithmetic operations on arbitrarily large integers.

### Understanding the Limits of `int` and `long`

- Primitive types like `int` and `long` have predefined maximum values:

- \*\*`Integer.MAX\_VALUE`\*\*: The largest value that an `int` can hold.

- \*\*`Long.MAX\_VALUE`\*\*: The largest value that a `long` can hold.

### Example: Checking Integer and Long Limits

```java

System.out.println("Max value of Integer: " + Integer.MAX\_VALUE);

System.out.println("Max value of Long: " + Long.MAX\_VALUE);

```

#### Output:

```java

Max value of Integer: 2147483647

Max value of Long: 9223372036854775807

```

- These limits restrict the range of calculations that can be performed with int and long.

- If calculations exceed these limits, the results can wrap around and become incorrect, such as returning negative values for large numbers.

### Calculating Factorials and Exceeding Limits

- Factorials grow very large quickly.

- \*\*For example\*\*, calculating 50! (50 factorial) involves multiplying numbers from 1 to 50.

- When using int or long, the result will overflow the maximum value and produce incorrect results.

### \*\*Example\*\*: Factorial Calculation Using intStream.rangeClosed

```java

int factorial = IntStream.rangeClosed(1, 50)

.reduce(1, (x, y) -> x \* y);

System.out.println(factorial);

```

#### Output:

```java

0

```

- In this case, the result is 0 because the product exceeds the int limit, resulting in an overflow. Similarly, using LongStream will also overflow at high values.

### Introducing `BigInteger` for Large Calculations

- To handle large number calculations like factorials, Java provides the `BigInteger` class.

- Unlike `int` and `long`, `BigInteger` can handle arbitrarily large values without overflow.

- It also provides methods for common arithmetic operations like addition, multiplication, etc.

### Calculating Large Factorials with `BigInteger`

- We can solve the factorial problem by converting the stream of numbers into `BigInteger` objects and using `BigInteger` for the multiplication.

- Step-by-Step: Using `BigInteger` to Calculate 50 Factorial

- Convert Each Number to `BigInteger`: Use `BigInteger``.valueOf()` to convert each int into a `BigInteger`.

- Perform Multiplication Using BigInteger: Use the `BigInteger.multiply()` method to multiply the numbers.

### \*\*Example\*\*: Calculating 50 Factorial with `BigInteger`

```java

BigInteger factorial50 = IntStream.rangeClosed(1, 50)

.mapToObj(BigInteger::valueOf)

.reduce(BigInteger.ONE, BigInteger::multiply);

System.out.println(factorial50);

```

- `IntStream.rangeClosed(1, 50)`: Generates a stream of integers from 1 to 50.

- `mapToObj(BigInteger::valueOf)`: Converts each integer into a BigInteger.

- `reduce(BigInteger.ONE, BigInteger::multiply)`: Multiplies all the BigInteger values, starting from BigInteger.ONE.

#### Output:

```java

30414093201713378043612608166064768844377641568960512000000000000

```

- This produces the correct value for 50! without any overflow issues.

---

## Playing further with Java Functional Programming

### Step 01: Joining Strings with joining and Playing with `flapMap`

### Key Objectives:

1. Learn how to join a list of strings into a single string using the `Collectors.joining()` method.

2. Understand the purpose and use of `flatMap()` to flatten streams of complex data structures.

3. Explore practical examples of using `flatMap()` for operations like splitting strings into characters and creating tuples from two lists.

### Introduction

In this lecture, we focus on two powerful stream operations in Java:

- \*\*`Collectors.joining()`\*\*: Used for joining multiple strings with a separator into a single string.

- \*\*`flatMap()`\*\*: A versatile method for flattening nested streams, useful when dealing with arrays or complex collections.

### Joining Strings Using `Collectors.joining()`

- To start, we explore how to combine a list of course names into a single string, where each course name is separated by a comma.

- This is easily achieved using the `Collectors.joining()` method.

### \*\*Example\*\*: Joining a List of Course Names

```java

List<String> courses = List.of("Spring", "Spring Boot", "API", "Microservices", "AWS", "PCF");

String joinedCourses = courses.stream()

.collect(Collectors.joining(", "));

System.out.println(joinedCourses);

```

- `Collectors.joining(", ")`: Joins all elements in the stream, separated by a comma and space.

#### Output:

```java

Spring, Spring Boot, API, Microservices, AWS, PCF

```

- The `joining()` method allows you to specify any separator (e.g., commas, spaces) to combine the strings.

### Working with `flatMap()`

- Next, we introduce `flatMap()`, a method that is useful when dealing with nested structures such as arrays or lists of lists. `flatMap()` helps flatten these structures into a single stream.

- In the first example, we want to split each course name into its individual characters.

- A direct use of `map()` would return a stream of arrays, but `flatMap()` allows us to flatten the arrays into a stream of characters.

### \*\*Example\*\*: Flattening a List of Strings into Characters

```java

List<String> courses = List.of("Spring", "Spring Boot", "API");

List<String> characters = courses.stream()

.map(course -> course.split(""))

.flatMap(Arrays::stream)

.collect(Collectors.toList());

System.out.println(characters);

```

- `map(course -> course.split(""))`: Splits each course name into an array of characters.

- `flatMap(Arrays::stream)`: Converts the array of characters into a flattened stream of individual characters.

#### Output:

```java

[S, p, r, i, n, g, S, p, r, i, n, g, B, o, o, t, A, P, I]

```

### Using `flatMap()` to Remove Duplicates

- We can further manipulate the stream by removing duplicate characters using the `distinct()` method.

### \*\*Example\*\*: Finding Distinct Characters

```java

List<String> distinctCharacters = courses.stream()

.map(course -> course.split(""))

.flatMap(Arrays::stream)

.distinct()

.collect(Collectors.toList());

System.out.println(distinctCharacters);

```

- `distinct()`: Removes duplicate characters from the stream.

#### Output:

```java

[S, p, r, i, n, g, B, o, t, A, P, I]

```

### Advanced Use of `flatMap()`: Generating Tuples

- In more advanced scenarios, `flatMap()` can be used to generate combinations of elements from two lists.

- \*\*For example\*\*, we can use `flatMap()` to create pairs (tuples) of course names that have the same number of characters.

### \*\*Example\*\*: Creating Tuples from Two Lists

```java

List<String> courses = List.of("Spring", "API", "AWS", "PCF", "Docker");

List<String> courses2 = List.of("Spring", "API", "AWS", "PCF", "Docker");

List<List<String>> coursePairs = courses.stream()

.flatMap(course -> courses2.stream()

.filter(course2 -> course.length() == course2.length() && !course.equals(course2))

.map(course2 -> List.of(course, course2)))

.collect(Collectors.toList());

System.out.println(coursePairs);

```

- `flatMap()`: For each course in the first list, flatMap() generates a stream of courses from the second list.

- `filter(course2 -> course.length() == course2.length())`: Filters the second list to match courses with the same length.

- `List.of(course, course2)`: Creates a pair (tuple) of matching courses.

#### Output:

```java

[[API, AWS], [API, PCF], [AWS, API], [AWS, PCF], [PCF, API], [PCF, AWS]]

```

### Explanation of `flatMap()` Use Cases

- The `flatMap()` method can be used in a variety of scenarios, especially when:

- You need to flatten nested collections (e.g., a stream of lists into a single stream of elements).

- You want to combine multiple sources of data into a single stream.

- In the examples above, we used `flatMap()` to flatten arrays of characters and to generate combinations (pairs) of matching courses.

---

### Step 02: Creating Higher Order Functions

### Key Objectives:

1. Understand the concept of Higher Order Functions (HOFs) in functional programming.

2. Learn how to create and use higher-order functions in Java by returning functions from methods.

3. Explore practical examples of using HOFs, including the use of predicates.

### Introduction

In this lecture, we explore the concept of \*\*Higher Order Functions (HOFs)\*\*, an important feature in functional programming.

A \*\*Higher Order Function\*\* is a function that either takes one or more functions as parameters or returns a function as its result.

In this case, we will focus on creating functions that return other functions, allowing us to generate reusable logic dynamically.

### What is a Higher Order Function?

A \*\*Higher Order Function\*\* in programming is a function that either:

- Takes a function as a parameter, or

- Returns a function as a result.

In this lecture, the focus is on functions that \*\*return functions\*\*. This is useful when you want to generate customized logic dynamically at runtime, such as creating predicates with varying conditions.

### Creating Predicates with Different Cutoff Values

Let's start by revisiting some predicates that filter courses based on their review scores. Initially, we define two separate predicates that check if a course has a review score greater than 90 or 95.

### \*\*Example\*\*: Predicates with Different Cutoff Values

```java

Predicate<Course> reviewScoreGreaterThan95 = course -> course.getReviewScore() > 95;

Predicate<Course> reviewScoreGreaterThan90 = course -> course.getReviewScore() > 90;

```

- These predicates work, but they are quite repetitive. The only difference between them is the cutoff value (90 or 95).

- Instead of hardcoding separate predicates, we can make this more flexible by creating a method that returns a predicate with any cutoff value.

### Refactoring to Use Higher Order Functions

- We will refactor the code to create a higher-order function that takes the cutoff value as a parameter and returns a predicate based on that value.

- This approach eliminates repetition and allows us to generate predicates dynamically.

### \*\*Example\*\*: Creating a Higher Order Function to Return a Predicate

```java

public static Predicate<Course> createPredicateWithCutoffReviewScore(int cutoffReviewScore) {

return course -> course.getReviewScore() > cutoffReviewScore;

}

```

- `createPredicateWithCutoffReviewScore(int cutoffReviewScore)`: This method accepts a cutoff value and returns a predicate that checks if a course's review score is greater than the cutoff.

- The method returns a function (a `Predicate<Course>`) based on the cutoff value.

- Now, we can generate predicates for any cutoff dynamically:

```java

Predicate<Course> reviewScoreGreaterThan90 = createPredicateWithCutoffReviewScore(90);

Predicate<Course> reviewScoreGreaterThan95 = createPredicateWithCutoffReviewScore(95);

```

### Applying the Higher Order Function

- Once we have created our higher-order function, we can use it to generate predicates and apply them to our courses.

- For example, we can filter courses with a review score greater than 90 or 95 using the dynamically generated predicates.

### \*\*Example\*\*: Using the Generated Predicates

```java

List<Course> courses = // Assume this is a list of courses

List<Course> highRatedCourses = courses.stream()

.filter(createPredicateWithCutoffReviewScore(95))

.collect(Collectors.toList());

System.out.println(highRatedCourses);

```

- We generate the predicate using the `createPredicateWithCutoffReviewScore(95)` method.

- The `filter()` method applies the generated predicate to filter out courses with a review score greater than 95.

- This dynamic approach allows for flexibility, making it easier to reuse the logic for different cutoff values.

### Higher Order Functions in Functional Programming

- The use of higher-order functions like the one we created is a hallmark of functional programming.

- They allow functions to be treated like first-class citizens:

- \*\*Functions as data\*\*: We can pass functions around as arguments, return them from methods, or store them in variables just like any other data.

- \*\*Dynamic logic generation\*\*: Higher-order functions allow us to create custom logic on the fly, reducing code duplication and making the program more flexible.

In functional programming, you can store methods as local variables, pass methods as parameters, and return methods from other methods—allowing greater control over the behavior of your code.

---

### Step 03: FP and Performance - Intermediate Stream Operations are Lazy

### Key Objectives:

1. Understand the concept of laziness in intermediate stream operations.

2. Learn how Java Streams execute operations efficiently by delaying execution until a terminal operation is invoked.

3. Explore how functional programming in Java helps improve performance, particularly with operations like filtering and mapping.

### Introduction

In this lecture, we explore the performance advantages of functional programming in Java, particularly focusing on the \*\*laziness\*\* of intermediate stream operations.

Laziness means that intermediate operations like `filter()`, `map()`, and `peek()` are not executed until a terminal operation (such as `findFirst()` or `collect()`) is invoked.

This allows Java to execute operations more efficiently by only doing the minimum amount of work required to get the result.

### Functional Programming and Performance

- Functional programming has existed for over 50 years, but its performance benefits have gained more attention recently, especially in the context of multicore processors.

- Java Streams, introduced in Java 8, enable efficient parallelization of code, making it easier to write performant programs.

- One key aspect of this performance improvement is how intermediate operations in Java Streams are \*\*lazy\*\*.

- Instead of processing the entire stream eagerly, intermediate operations are only evaluated when a terminal operation is called.

### Example of Stream Operations

- Let's start with an example where we apply a series of operations on a list of course names.

- The goal is to filter out courses with names longer than 11 characters, convert them to uppercase, and return the first match.

### Example: Filter, Map, and Find First

```java

List<String> courses = List.of("Spring", "Spring Boot", "API", "Microservices", "AWS");

String result = courses.stream()

.filter(course -> course.length() > 11)

.map(String::toUpperCase)

.findFirst()

.orElse("No match");

System.out.println(result);

```

- `filter()`: Filters the courses based on the length of their names.

- `map()`: Converts the course names to uppercase.

- `findFirst()`: Finds the first course that satisfies the conditions.

#### Output:

```java

MICROSERVICES

```

### Understanding Laziness in Intermediate Operations

- The key point about streams is that intermediate operations (like `filter()` and `map()`) are lazy.

- They are not executed immediately when encountered. Instead, Java "chains" these operations and waits for a terminal operation (like `findFirst()`) to trigger their execution.

### Demonstrating Lazy Execution with `peek()`

- To understand the lazy nature of intermediate operations, we can use the peek() method to see when each step of the stream is executed.

- The `peek()` method allows us to inspect the elements as they flow through the pipeline without modifying them.

### Example: Using `peek()` to Observe Stream Execution

```java

String result = courses.stream()

.peek(System.out::println) // Inspect the courses before filtering

.filter(course -> course.length() > 11)

.peek(System.out::println) // Inspect the courses after filtering

.map(String::toUpperCase)

.peek(System.out::println) // Inspect the courses after mapping to uppercase

.findFirst()

.orElse("No match");

System.out.println("Result: " + result);

```

- The first `peek()` will print all the course names before they are filtered.

- The second `peek()` will only print the course names that pass the filter.

- The third `peek()` will print the course names after they are converted to uppercase.

#### Output:

```java

Spring

Spring Boot

API

Microservices

Microservices

MICROSERVICES

Result: MICROSERVICES

```

- As soon as the `findFirst()` operation finds the first course that meets the conditions ("Microservices"), the stream processing stops.

- The remaining courses ("AWS") are not even evaluated, demonstrating how efficient functional programming can be.

### Laziness of Intermediate Operations

- The intermediate operations (`filter()`, `map()`, `peek()`) in the stream pipeline are lazy:

- They don’t execute immediately: Java doesn’t process these operations until a terminal operation (like `findFirst()` or `collect()`) is called.

- \*\*Short-circuiting\*\*: As shown in the example, once findFirst() finds a match, the rest of the stream is not processed, saving computational resources.

- This lazy evaluation ensures that Java only performs the minimum amount of work necessary to get the result, which leads to performance optimizations.

### Understanding Terminal Operations

- A terminal operation triggers the execution of the intermediate operations in the stream pipeline.

Examples of terminal operations include:

- `findFirst()`: Returns the first element that matches the conditions.

- `collect()`: Collects the results into a collection like a list or set.

- `forEach()`: Applies an action to each element in the stream.

Once a terminal operation is invoked, Java evaluates the stream pipeline just enough to fulfill the terminal operation.

### Example: Removing the Terminal Operation

- If we remove the terminal operation (`findFirst()`), the intermediate operations will not be executed.

```java

Stream<String> courseStream = courses.stream()

.filter(course -> course.length() > 11)

.map(String::toUpperCase);

```

- No terminal operation is called, so nothing is printed or processed. The stream operations are only defined, but not executed.

---

### Step 04: Improving Performance with Parallelization of Streams

### Key Objectives:

1. Understand how to leverage multicore processors to improve the performance of functional code.

2. Learn how to parallelize streams in Java with the `parallel()` method.

3. Explore the benefits and mechanisms of parallelizing stream operations and how Java efficiently handles them.

### Introduction

- Modern laptops and computers are equipped with multicore processors, allowing programs to execute tasks in parallel and improve performance.

- In functional programming, \*\*parallelization\*\* is a straightforward and effective way to take advantage of this.

- In this lecture, we explore how to parallelize streams in Java and measure the performance improvements.

### Setting Up a Sequential Stream Operation

- To begin, let’s create a basic example where we sum a large range of numbers using Java Streams.

- First, we perform this operation sequentially (without parallelization).

### Example: Summing a Range of Numbers

```java

LongStream.range(0, 1000000000L).sum();

```

- This creates a stream of long values from 0 to 999,999,999 and calculates the sum. While this works, it can take time to execute.

### Measuring the Performance of Sequential Streams

- To measure how long it takes to sum these numbers, we can use the `System.currentTimeMillis()` method to track the start and end times.

### Example: Measuring Execution Time

```java

long startTime = System.currentTimeMillis();

long result = LongStream.range(0, 1000000000L).sum();

long endTime = System.currentTimeMillis();

System.out.println("Sum: " + result);

System.out.println("Time taken: " + (endTime - startTime) + " ms");

```

- `System.currentTimeMillis()`: Captures the current system time in milliseconds.

- We calculate the sum and then measure the total time taken by subtracting the start time from the end time.

### Introducing Parallelization with Parallel Streams

- Java allows us to easily parallelize stream operations with the `parallel()` method.

- This enables Java to split the stream into multiple chunks and process them concurrently across multiple cores, speeding up operations.

### Example: Parallelizing the Stream

```java

long startTime = System.currentTimeMillis();

long result = LongStream.range(0, 1000000000L)

.parallel()

.sum();

long endTime = System.currentTimeMillis();

System.out.println("Sum: " + result);

System.out.println("Time taken with parallelization: " + (endTime - startTime) + " ms");

```

- `parallel()`: This method tells Java to process the stream in parallel, allowing the operation to take advantage of multiple cores.

- Java will split the stream into chunks and process each chunk on different cores, improving the overall execution time.

### Comparing Sequential vs. Parallel Execution

- When we run both the sequential and parallel versions of the code, we will typically see a performance improvement in the parallel version.

- This is because Java can distribute the workload across multiple processor cores, reducing the total time required to complete the task.

### Understanding Parallel Stream Mechanics

- Parallel streams work by splitting the stream into smaller parts, processing each part separately, and then merging the results.

- This is possible because of the stateless nature of functional operations, which do not depend on shared data between iterations.

### Key Concepts:

- \*\*Stateful Operations\*\*: Operations that rely on changing or maintaining a state (like a loop variable) are difficult to parallelize because they require synchronization.

- \*\*Stateless Functional Programming\*\*: In contrast, functional programming emphasizes stateless operations, making it easier to split the work across multiple cores.

### Why Functional Programming is Easy to Parallelize

- In traditional structured code, loops and stateful operations make parallelization difficult because multiple threads would need to coordinate access to shared variables (e.g., a sum variable in a loop).

- In functional programming, however, streams are stateless. Instead of manually controlling how each element is processed, we describe what operations to apply, and Java handles the parallelization automatically.

#### The functional approach allows Java to:

- Divide the stream into chunks.

- Process each chunk independently across multiple cores.

- Merge the results from each chunk at the end.

- This makes functional code naturally parallelizable and more efficient.

---

## Functional Programming makes Java Easy

### Step 01: Modifying lists with `replaceAll` and `removeIf`

### Key Objectives:

1. Learn how to modify elements in a list using the `replaceAll()` method.

2. Understand how to remove elements from a list based on a condition using the `removeIf()` method.

3. Explore the advantages of using functional programming to simplify list modifications in Java.

### Introduction

Functional programming in Java introduces new methods that make list manipulation much easier and more expressive.

In this lecture, we explore two important methods for modifying lists: \*\*`replaceAll()`\*\* and \*\*`removeIf()`\*\*.

These methods allow us to update and filter list elements by passing functions directly to them, reducing the need for manual loops and enhancing code readability.

### Modifying List Elements Using `replaceAll()`

- Traditionally, to modify all elements in a list, we would loop through the list and update each element manually.

- However, with the `replaceAll()` method, we can easily modify all the elements in a list by passing a function that specifies how each element should be updated.

### \*\*Example\*\*: Converting All Elements to Uppercase

- Suppose we have a list of course names, and we want to convert all the course names to uppercase.

```java

List<String> courses = List.of("Spring", "Spring Boot", "API", "Microservices", "AWS");

courses.replaceAll(str -> str.toUpperCase());

```

- `replaceAll()`: This method applies the specified function (in this case, `str.toUpperCase()`) to each element of the list.

- However, because `List.of()` creates an immutable list, this operation will throw an error. Immutable lists cannot be modified.

### Handling Immutable Lists

- To perform modifications, we need to work with a modifiable list.

- One way to achieve this is by creating a new `ArrayList` from the original list.

### \*\*Example\*\*: Creating a Modifiable List

```java

List<String> modifiableCourses = new ArrayList<>(courses);

modifiableCourses.replaceAll(str -> str.toUpperCase());

System.out.println(modifiableCourses);

```

- `new ArrayList<>(courses)`: Creates a new modifiable list from the original courses list.

- `replaceAll(str -> str.toUpperCase())`: Converts each course name to uppercase.

#### Output:

```java

[SPRING, SPRING BOOT, API, MICROSERVICES, AWS]

```

- Now, the list is successfully modified, and all course names are converted to uppercase.

### Removing Elements from a List Using `removeIf()`

- Another powerful method introduced in Java is `removeIf()`, which allows us to remove elements from a list based on a condition.

- Instead of looping through the list manually and removing elements, we can pass a predicate (a condition) directly to the `removeIf()` method.

### \*\*Example\*\*: Removing Courses with Short Names

- Suppose we want to remove all courses from the list that have names with fewer than six characters.

```java

modifiableCourses.removeIf(course -> course.length() < 6);

System.out.println(modifiableCourses);

```

- `removeIf()`: This method takes a predicate (a condition) and removes all elements from the list that satisfy the condition.

- In this case, `course.length() < 6` removes all course names with fewer than six characters.

#### Output:

```java

[SPRING, SPRING BOOT, MICROSERVICES]

```

- In this example, "API" and "AWS" are removed from the list because their names have fewer than six characters.

### Benefits of Functional Programming in List Manipulation

Using methods like `replaceAll()` and `removeIf()` makes list manipulation in Java much simpler and more expressive:

- \*\*Cleaner code\*\*: We can apply transformations and filters directly, without the need for explicit loops.

- \*\*Higher readability\*\*: The logic is easier to understand since it focuses on what needs to be done rather than how to do it.

- \*\*Functional approach\*\*: By passing functions (or predicates) directly to these methods, we can encapsulate logic in a more reusable and declarative way.

---

### Step 02: Playing with Files using Functional Programming

### Key Objectives:

1. Learn how to read and manipulate files using functional programming constructs in Java.

2. Understand how to process the contents of a file using streams, including operations like reading lines, finding unique words, and sorting.

3. Explore how to list and filter directories using functional programming.

### Introduction

Functional programming in Java has made file handling more intuitive and efficient.

In this lecture, we will explore how to perform common file operations using streams and functional programming constructs.

By the end of the lecture, you will know how to read file contents, manipulate text, and work with directories—all in a functional and streamlined way.

### Reading a File with `Files.lines()`

- To start, we will learn how to read the contents of a text file using the `Files.lines()` method.

- This method allows us to read all lines in a file and process them as a stream.

### \*\*Example\*\*: Reading and Printing File Contents

```java

Path path = Paths.get("file.txt");

Files.lines(path)

.forEach(System.out::println);

```

- `Files.lines(path)`: Reads the file line by line and returns a `Stream<String>`.

- `forEach(System.out::println)`: Prints each line of the file to the console.

- This makes reading files simple and efficient

- However, you must handle exceptions (like IOException), so be sure to include proper error handling or a throws Exception clause in your method.

### Manipulating File Content

- Now that we can read the file, let's go one step further. Suppose you want to find the unique words in the file.

- To achieve this, we can split each line into words, flatten the results, and filter for distinct words.

### \*\*Example\*\*: Finding Unique Words in a File

```java

Files.lines(path)

.flatMap(line -> Arrays.stream(line.split(" "))) // Split lines into words

.distinct() // Find distinct words

.forEach(System.out::println); // Print unique words

```

- `split(" ")`: Splits each line into words based on spaces.

- `flatMap()`: Flattens the stream of arrays into a stream of words.

- `distinct()`: Filters out duplicate words to show only unique ones.

### Sorting Unique Words

- Once we have the unique words, we may want to sort them in alphabetical order.

- We can easily achieve this by adding the `sorted()` method to our stream.

### \*\*Example\*\*: Sorting Unique Words Alphabetically

```java

Files.lines(path)

.flatMap(line -> Arrays.stream(line.split(" "))) // Split lines into words

.distinct() // Find distinct words

.sorted() // Sort words alphabetically

.forEach(System.out::println); // Print sorted words

```

- `sorted()`: Sorts the words in natural (alphabetical) order.

- The sorted result will list all the unique words in alphabetical order, with uppercase words coming before lowercase ones.

### Listing Files in a Directory

- Next, we explore how to list the files in a directory.

- Using `Files.list()`, we can get a stream of files and directories in a given path.

### \*\*Example\*\*: Listing Files and Directories

```java

Files.list(Paths.get("."))

.forEach(System.out::println);

```

- `Files.list(Paths.get("."))`: Lists all files and directories in the current project root.

- `forEach(System.out::println)`: Prints the names of all files and directories.

- This method is useful when you want to inspect the contents of a directory programmatically.

### Filtering Directories

- Sometimes, we may want to filter out only directories from the list of files.

- This can be done easily using the filter() method and the `Files.isDirectory()` function.

### \*\*Example\*\*: Filtering Only Directories

```java

Files.list(Paths.get("."))

.filter(Files::isDirectory) // Filter only directories

.forEach(System.out::println); // Print directory names

```

- `filter(Files::isDirectory)`: Filters the stream to only include directories.

- This filters out non-directory files, leaving only the directories in the list.

### Other File Operations

- Java’s Files API offers many other methods that can be used in combination with streams for more specific operations.

\*\*For example\*\*:

- `Files.isRegularFile()`: Check if a path is a regular file.

- `Files.isReadable()`: Check if a file is readable.

- `Files.isExecutable()`: Check if a file is executable.

- `Files.isHidden()`: Check if a file is hidden.

These methods allow you to fine-tune your file operations based on different conditions.

---

### Step 03: Playing with Threads using Functional Programming

### Key Objectives:

1. Learn how to create and manage threads in Java using both traditional and functional approaches.

2. Understand how functional programming simplifies the creation of threads by using lambda expressions.

3. Explore how functional programming improves readability and reduces boilerplate code when working with concurrency in Java.

### Introduction

In this lecture, we explore how functional programming simplifies working with threads in Java.

Traditionally, creating threads required implementing the `Runnable` interface or extending the `Thread` class.

With functional programming, we can streamline this process using \*\*lambda expressions\*\*, making the code more concise and easier to understand.

### Creating Threads with the Traditional Approach

- In Java, a common way to create a thread is by implementing the `Runnable` interface.

- This involves defining the `run()` method, which contains the logic that will be executed by the thread. We then pass this `Runnable` to a `Thread` object and start the thread.

### \*\*Example\*\*: Creating Threads Using `Runnable`

```java

Runnable runnable = new Runnable() {

@Override

public void run() {

for (int i = 0; i < 10000; i++) {

System.out.println(Thread.currentThread().getId() + ": " + i);

}

}

};

Thread thread1 = new Thread(runnable);

Thread thread2 = new Thread(runnable);

thread1.start();

thread2.start();

```

- \*\*Runnable interface\*\*: This defines a task to be executed in a separate thread. We implement the `run()` method to specify the task.

- \*\*Thread class\*\*: We create new Thread objects, passing in the Runnable, and start them using `thread1.start()` and `thread2.start()`.

- The code will print the thread ID and the counter from 0 to 9999 for each thread running in parallel.

### Simplifying Thread Creation with Functional Programming

- In Java, functional interfaces are interfaces that contain exactly one abstract method.

- The Runnable interface is a functional interface because it has only one method: `run()`.

- This means we can use lambda expressions to provide an implementation, making the code shorter and more readable.

### \*\*Example\*\*: Using Lambda Expressions to Create Threads

```java

Runnable runnable2 = () -> {

for (int i = 0; i < 10000; i++) {

System.out.println(Thread.currentThread().getId() + ": " + i);

}

};

Thread thread3 = new Thread(runnable2);

Thread thread4 = new Thread(runnable2);

thread3.start();

thread4.start();

```

- \*\*Lambda Expression\*\*: The `runnable2` variable is defined as a lambda expression, which provides the implementation of the `run()` method without needing to explicitly define the method inside an anonymous class.

- \*\*Simplification\*\*: By using the lambda syntax, we eliminate the need for boilerplate code like the Runnable interface implementation and focus directly on the logic to be executed in the thread.

### Understanding the Benefits of the Functional Approach

Using lambda expressions in functional programming simplifies thread management by:

- \*\*Reducing boilerplate\*\*: We no longer need to write out the entire Runnable implementation, making the code more concise.

- \*\*Improving readability\*\*: The code is easier to understand because it focuses on the core logic rather than the ceremony of implementing an interface.

- \*\*Leveraging functional interfaces\*\*: Since Runnable is a functional interface, we can provide its implementation with a lambda expression, simplifying how we define tasks for threads.

---

### Step 04: Using Functional Programming in Java Applications

### Key Objectives:

1. Understand how functional programming fits into Java's object-oriented structure.

2. Learn where to apply functional programming constructs like streams, lambdas, and functional interfaces in real-world applications.

3. Recognize the challenges teams face when adopting functional programming and strategies to overcome them.

### Introduction

Java is primarily an \*\*object-oriented programming (OOP)\*\* language, where applications are built by creating classes, objects, and managing interactions between these objects.

However, with the introduction of \*\*functional programming (FP)\*\* features in Java, such as \*\*streams\*\*, \*\*lambda expressions\*\*, and \*\*method references\*\*, developers can now blend object-oriented and functional paradigms to write more concise, readable, and maintainable code.

### Integrating Functional Programming into Java Applications

- While Java remains object-oriented, \*\*functional programming\*\* can be highly useful in specific contexts.

- The best place to start incorporating functional programming is wherever you are processing collections of data, such as \*\*lists\*\*, \*\*sets\*\*, or \*\*maps\*\*.

### \*\*Example\*\*: Using Streams to Process Collections

```java

List<String> courses = List.of("Spring", "Java", "Kubernetes", "Microservices");

// Using functional programming to filter and process the list

courses.stream()

.filter(course -> course.length() > 6)

.map(String::toUpperCase)

.forEach(System.out::println);

```

- \*\*Streams\*\*: Functional programming makes it easier to process collections with declarative operations like `filter()`, `map()`, and `forEach()`.

- \*\*Lambdas and Method References\*\*: We use lambda expressions and method references to apply operations without explicitly defining loops.

- Functional programming fits perfectly into these contexts, where collections and data processing tasks are common.

### Functional Programming as a Complement to Object-Oriented Programming

- Functional programming does not replace object-oriented programming in Java. Instead, it complements OOP by simplifying specific tasks like:

- Data manipulation and transformations.

- Working with large collections using streams.

- Writing cleaner and more readable code when performing repetitive tasks.

- For instance, when you need to perform operations on collections (filtering, sorting, mapping), functional programming is a powerful tool.

- However, Java still relies on OOP principles for class design, object relationships, and larger application architecture.

### Challenges in Adopting Functional Programming

- One of the biggest challenges when introducing functional programming into existing Java projects is the learning curve for developers.

- Not all developers may be familiar with functional programming constructs like lambda expressions, method references, and stream operations.

### \*\*Example\*\*: Complex Functional Code

```java

List<String> courses = List.of("Spring", "Java", "Kubernetes", "Microservices");

courses.stream()

.filter(course -> course.startsWith("S"))

.map(course -> course.length())

.forEach(System.out::println);

```

- For developers unfamiliar with functional programming, this code can be difficult to understand at first.

### The Paradigm Shift in Problem Solving

- The transition from structured programming to functional programming requires a paradigm shift.

- While structured programming focuses on explicit loops, iteration, and state management, functional programming focuses on declarative problem solving:

- \*\*Declarative programming\*\*: You describe what you want to achieve (e.g., filtering a list) without detailing how it should be done (e.g., using a for loop).

- \*\*Immutability and statelessness\*\*: Functional programming encourages avoiding state mutation, reducing the risk of bugs and making code easier to reason about.\

### The Benefits of Functional Programming in Java

- Despite the learning curve, functional programming provides several benefits:

- Concise and readable code: It reduces boilerplate code, allowing you to write more expressive and declarative operations.

- Easier data processing: Functional constructs like streams make it easier to manipulate data, especially large datasets.

- \*\*Parallelism\*\*: Functional programming simplifies parallel processing by removing the need for manual thread management.

- By understanding these benefits, teams can slowly adopt functional programming in their daily Java development workflow.

---