In Java 8, the Optional class was introduced in the java.util package to help avoid NullPointerException and make your code more readable and robust by explicitly handling the possibility of null.

**✅ What is Optional?**

Optional<T> is a container object which may or may not contain a non-null value. If a value is present, isPresent() returns true and get() returns the value.

**✅ Why use Optional?**

To represent **optional (nullable) values** in a clean, functional style, instead of returning null.

**✅ Commonly Used Methods:**

| **Method** | **Description** |
| --- | --- |
| empty() | Returns an empty Optional. |
| of(value) | Returns an Optional with the specified non-null value. Throws NullPointerException if the value is null. |
| ofNullable(value) | Returns an Optional describing the value, or empty if null. |
| isPresent() | Returns true if a value is present, otherwise false. |
| ifPresent(Consumer) | Executes the given block if a value is present. |
| orElse(other) | Returns the value if present, otherwise returns other. |
| orElseGet(Supplier) | Similar to orElse, but lazily computes the value. |
| orElseThrow() | Throws NoSuchElementException if no value is present. |
| map(Function) | If a value is present, applies the mapping function. |
| flatMap(Function) | Same as map, but avoids nested Optional. |

**✅ Example 1: Basic Usage**

Optional<String> name = Optional.of("Vishnu");

System.out.println(name.get()); // Output: Vishnu

**✅ Example 2: Avoiding Null Checks**

Optional<String> name = Optional.ofNullable(getName());

name.ifPresent(n -> System.out.println(n.toUpperCase()));

**✅ Example 3: Providing Default Values**

String value = Optional.ofNullable(getName()).orElse("Default Name");

**✅ Example 4: Using map()**

Optional<String> name = Optional.of("vishnu");

Optional<String> upper = name.map(String::toUpperCase);

System.out.println(upper.get()); // Output: VISHNU

**❌ Things to Avoid**

* Do not use Optional for fields in entity classes (e.g., JPA).
* Do not serialize Optional (it's not intended for that).
* Avoid calling get() without checking isPresent()—use safer alternatives.

**functional interfaces**

* **Consumer<T>**
* **Supplier<T>**

Bahut accha sawal hai Vishnu! 💡 Java 8 ne java.util.function package me **functional interfaces** diye jaise:

* Consumer<T>
* Supplier<T>

Yeh dono **lambda expressions** ke sath use hote hain, aur functional-style programming ko support karte hain.

## ✅ 1. Consumer<T> Interface

### 🔹 Meaning:

Consumer ek functional interface hai jo:

* **Ek input leta hai**
* **Kuch return nahi karta** (i.e., void return type)

### 🔹 Functional Method:

void accept(T t)

### ✅ Example:

import java.util.function.Consumer;

public class ConsumerDemo {

public static void main(String[] args) {

Consumer<String> printer = name -> System.out.println("Hello " + name);

printer.accept("Vishnu"); // Output: Hello Vishnu

}

}

### ✅ Real-World Use:

List<String> names = List.of("Vishnu", "Amit", "Rahul");

names.forEach(name -> System.out.println(name.toUpperCase()));

Ya:

names.forEach(System.out::println); // Method reference

## ✅ 2. Supplier<T> Interface

### 🔹 Meaning:

Supplier ek functional interface hai jo:

* **Kuch return karta hai**
* **Koi input nahi leta**

### 🔹 Functional Method:

T get()

### ✅ Example:

import java.util.function.Supplier;

public class SupplierDemo {

public static void main(String[] args) {

Supplier<Double> randomValue = () -> Math.random();

System.out.println(randomValue.get()); // Example Output: 0.8463

}

}

### ✅ Real-World Use:

Supplier<String> defaultName = () -> "Guest";

String name = Optional.ofNullable(null).orElseGet(defaultName);

System.out.println(name); // Output: Guest

## 🔁 Summary Table:

| **Interface** | **Takes Input** | **Returns Output** | **Use Case** |
| --- | --- | --- | --- |
| Consumer<T> | ✅ Yes | ❌ No | Use when you want to **process** something (like printing/logging) |
| Supplier<T> | ❌ No | ✅ Yes | Use when you want to **generate** or **supply** data (like random number or default values) |

## 🔁 Intermediate Operations

(These return another Stream and are **lazy**; only executed when a terminal operation is invoked.)

Great question! Let's break it down simply:

**🔁 “These return another Stream”**

When you call an **intermediate operation** like filter(), map(), or sorted(), it doesn’t do anything immediately.  
Instead, it **returns a new Stream** — a kind of “pipeline” with the logic you’ve added.

For example:

Stream<String> s = list.stream().filter(name -> name.startsWith("A"));

Here, filter() creates a new stream with the filter logic, but **no filtering actually happens yet**.

**💤 “They are lazy; only executed when a terminal operation is invoked”**

"Lazy" means the intermediate operations **don't run immediately**. They are **just recorded**.

The actual execution of the operations happens **only when** a **terminal operation** like collect(), forEach(), count(), etc., is called.

🧠 Think of it like setting up a recipe — nothing is cooked until you press **Start**.

**🔚 Terminal Operation triggers everything**

List<String> result = list.stream() // creates stream

.filter(name -> name.startsWith("A")) // intermediate (lazy)

.map(String::toUpperCase) // intermediate (lazy)

.collect(Collectors.toList()); // terminal (execution starts!)

👉 When .collect() is called, **then**:

* The stream is traversed
* filter() is applied
* Then map() is applied
* And finally, the results are collected

**📌 Summary**

| **Concept** | **Meaning** |
| --- | --- |
| Intermediate Ops | Define what to do (but don’t execute). |
| Terminal Ops | Start actual execution of the pipeline. |
| Lazy | Stream waits until a terminal operation to start working. |

Let me know if you'd like a **real-world analogy or diagram**!

**Intermediate** **Operation :**

| **Method** | **Description** |
| --- | --- |
| filter(Predicate) | Filters elements based on a condition. |
| map(Function) | Transforms each element. |
| mapToInt, mapToDouble, mapToLong | Specialized mapping for primitive streams. |
| flatMap(Function) | Flattens nested streams. |
| flatMapToInt, etc. | Primitive versions of flatMap. |
| distinct() | Removes duplicates (uses equals()). |
| sorted() | Sorts elements using natural order. |
| sorted(Comparator) | Sorts using custom comparator. |
| peek(Consumer) | Performs an action on each element (mainly for debugging). |
| limit(long n) | Limits the stream to n elements. |
| skip(long n) | Skips the first n elements. |
| takeWhile(Predicate) 🔹 (Java 9+) | Takes elements while the condition is true. |
| dropWhile(Predicate) 🔹 (Java 9+) | Drops elements while the condition is true. |

## 🔚 Terminal Operations

(These produce a result or a side-effect and **consume** the stream.)

| **Method** | **Description** |
| --- | --- |
| forEach(Consumer) | Performs an action for each element. |
| forEachOrdered(Consumer) | Maintains encounter order (in parallel streams). |
| toArray() | Returns an array. |
| reduce(BinaryOperator) | Reduces elements to a single value. |
| reduce(identity, BinaryOperator) | With identity. |
| reduce(identity, BiFunction, BinaryOperator) | For parallel reduction. |
| collect(Collector) | Converts stream to a collection, map, etc. |
| collect(Supplier, BiConsumer, BiConsumer) | Custom collect logic. |
| min(Comparator) | Finds the minimum element. |
| max(Comparator) | Finds the maximum element. |
| count() | Counts number of elements. |
| anyMatch(Predicate) | Returns true if any element matches. |
| allMatch(Predicate) | Returns true if all match. |
| noneMatch(Predicate) | Returns true if none match. |
| findFirst() | Returns the first element (Optional). |
| findAny() | Returns any element (Optional, faster in parallel). |

## 🔀 Primitive Stream Specializations

(For performance: IntStream, LongStream, DoubleStream)

| **Method** | **Description** |
| --- | --- |
| range(start, end) | Range of numbers (exclusive). |
| rangeClosed(start, end) | Range (inclusive). |
| sum(), average() | Common numeric ops. |
| summaryStatistics() | Min, max, sum, count, average together. |

## 📄 Example Code Snippet

List<String> names = Arrays.asList("John", "Jane", "Jack", "Jill");

List<String> filteredSortedNames = names.stream()

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

.map(String::toUpperCase)

.sorted()

.collect(Collectors.toList());

System.out.println(filteredSortedNames); // [JACK, JANE, JILL, JOHN]

### 🧺 What is Collectors in Java?

Collectors is a **utility class** in the **Java Stream API** (part of java.util.stream package) that provides **predefined implementations** of the Collector interface — used to collect the results of a stream into **collections**, **strings**, **maps**, **summary statistics**, and more.

### 📦 Import Statement

import java.util.stream.Collectors;

### 🔚 Used with collect() Terminal Operation

Stream<T>.collect(Collector<T, A, R> collector)

You use Collectors inside .collect() to tell **how** to gather the final stream result.

### 🧩 Common Collectors Methods with Examples

| **Method** | **Description** | **Example** |
| --- | --- | --- |
| toList() | Collects elements into a List | collect(Collectors.toList()) |
| toSet() | Collects into a Set | collect(Collectors.toSet()) |
| toMap(keyMapper, valueMapper) | Collects into a Map | collect(Collectors.toMap(...)) |
| joining() | Joins strings (like String.join()) | collect(Collectors.joining(", ")) |
| groupingBy(classifier) | Groups elements by classifier function | collect(Collectors.groupingBy(...)) |
| partitioningBy(predicate) | Splits elements into two groups (true/false) | collect(Collectors.partitioningBy(...)) |
| counting() | Counts elements | collect(Collectors.counting()) |
| summarizingInt(ToIntFunction) | Summary stats (count, sum, min, max, avg) | collect(Collectors.summarizingInt(...)) |
| mapping(mapper, downstream) | Maps and collects | collect(Collectors.mapping(...)) |

### 🧪 Example: Collect into List

List<String> names = Stream.of("Alice", "Bob", "Charlie")

.collect(Collectors.toList());

System.out.println(names); // Output: [Alice, Bob, Charlie]

### 🧪 Example: Join Strings

String csv = Stream.of("Java", "Python", "C++")

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

System.out.println(csv); // Output: Java, Python, C++

### 🧪 Example: Group by Length

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

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

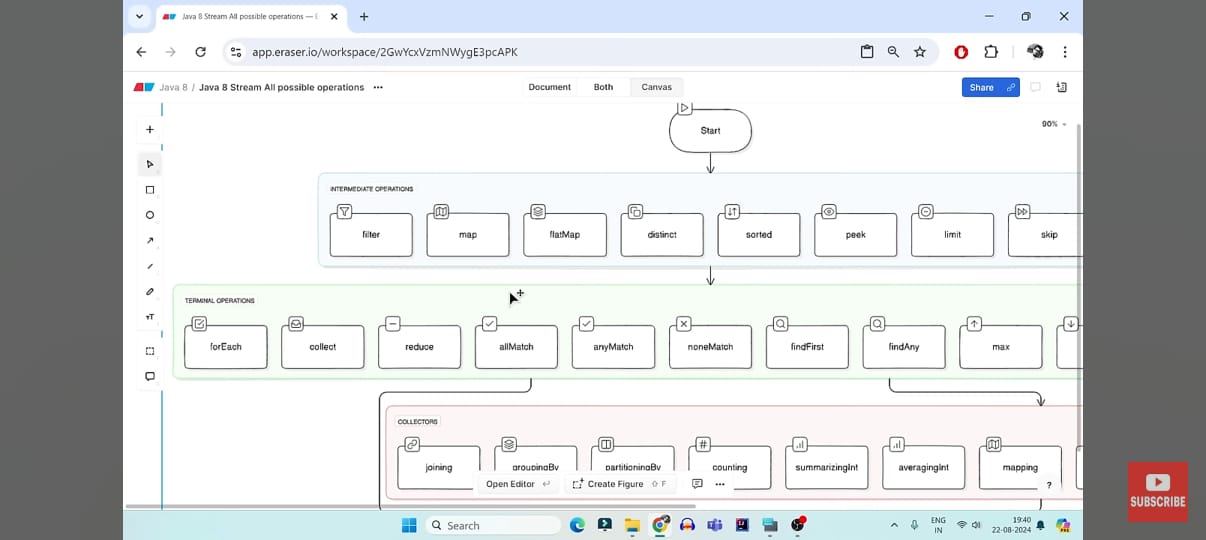
.collect(Collectors.groupingBy(String::length));

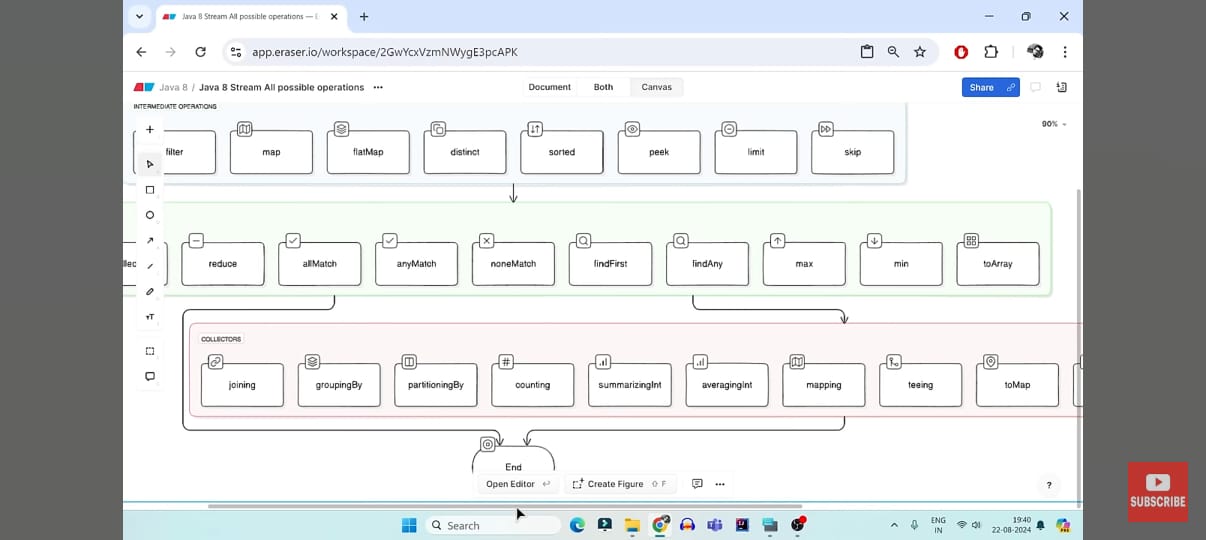
System.out.println(grouped);

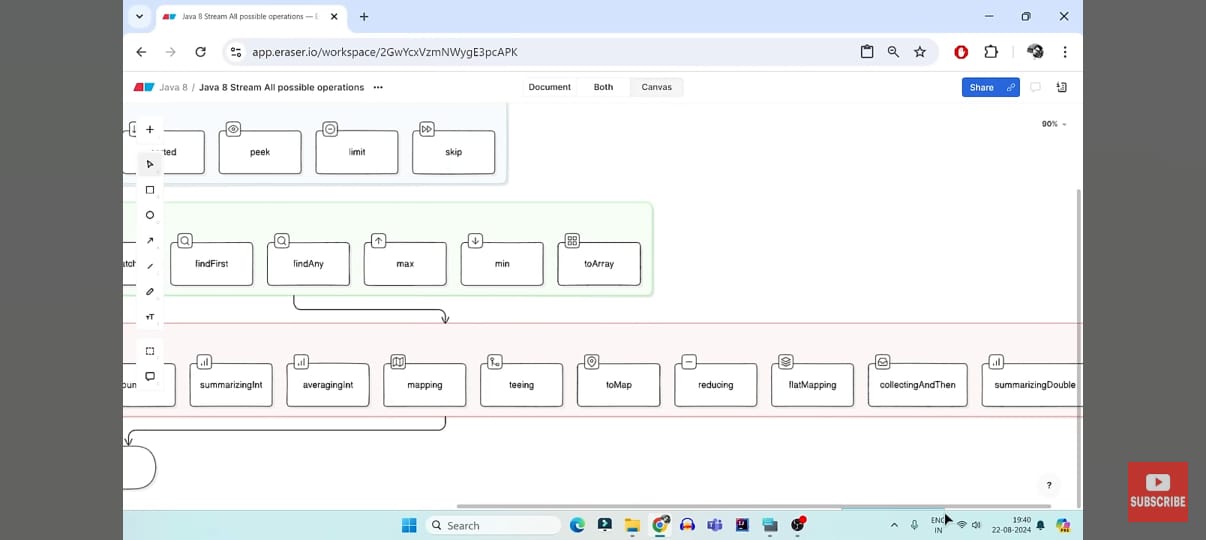
// Output: {2=[hi], 3=[hey, bye], 5=[hello]}

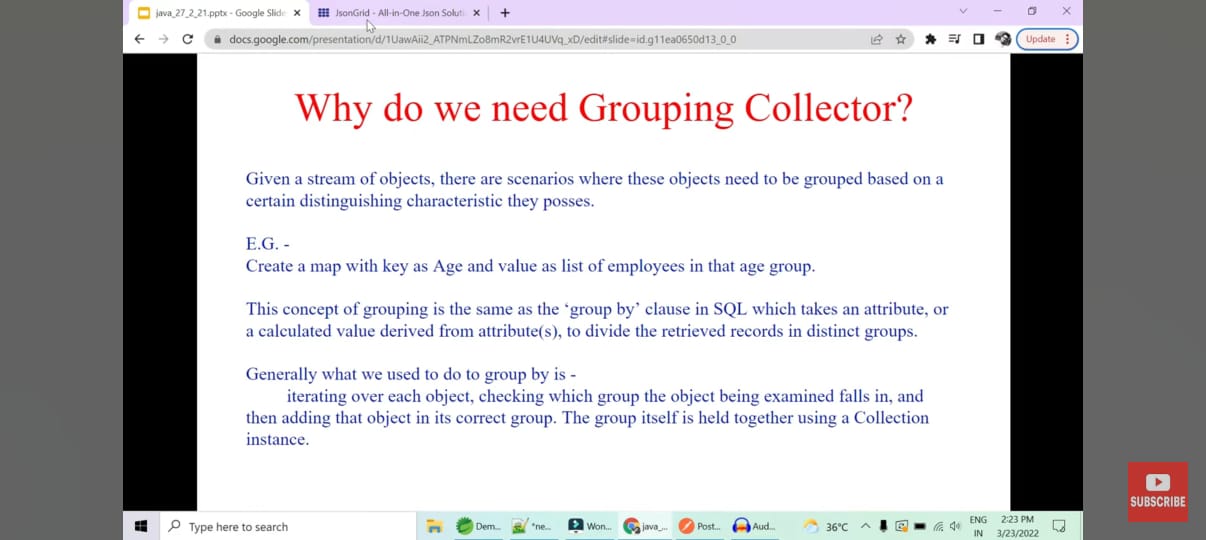
### ✅ Summary

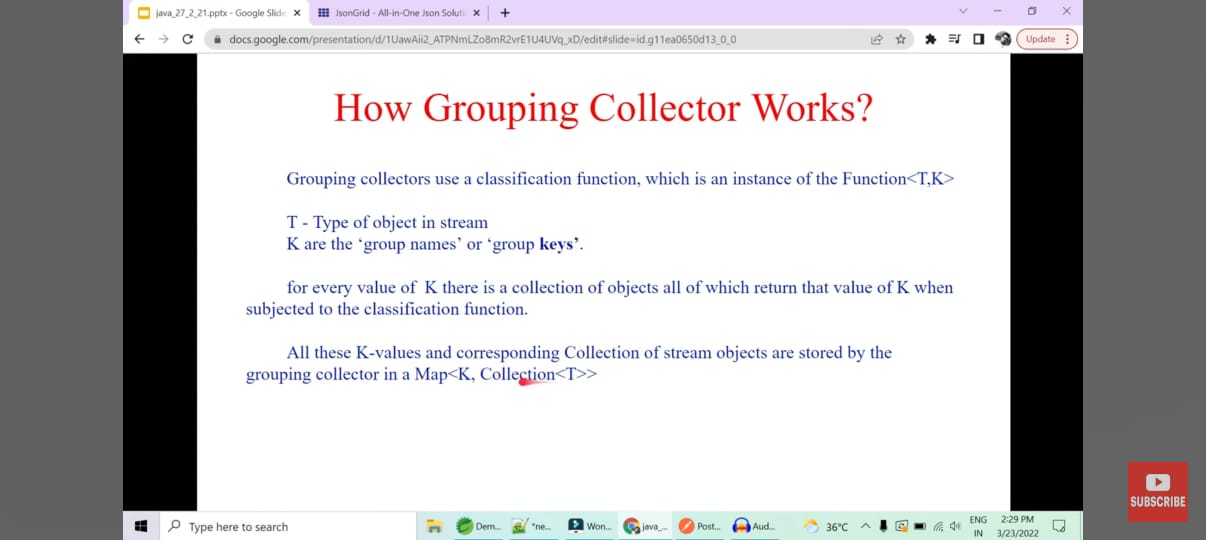
| **Collectors is used with .collect() to...** |
| --- |
| Gather stream elements into a collection. |
| Transform or summarize the result. |
| Group or partition data. |
| Build complex structures from stream data. |

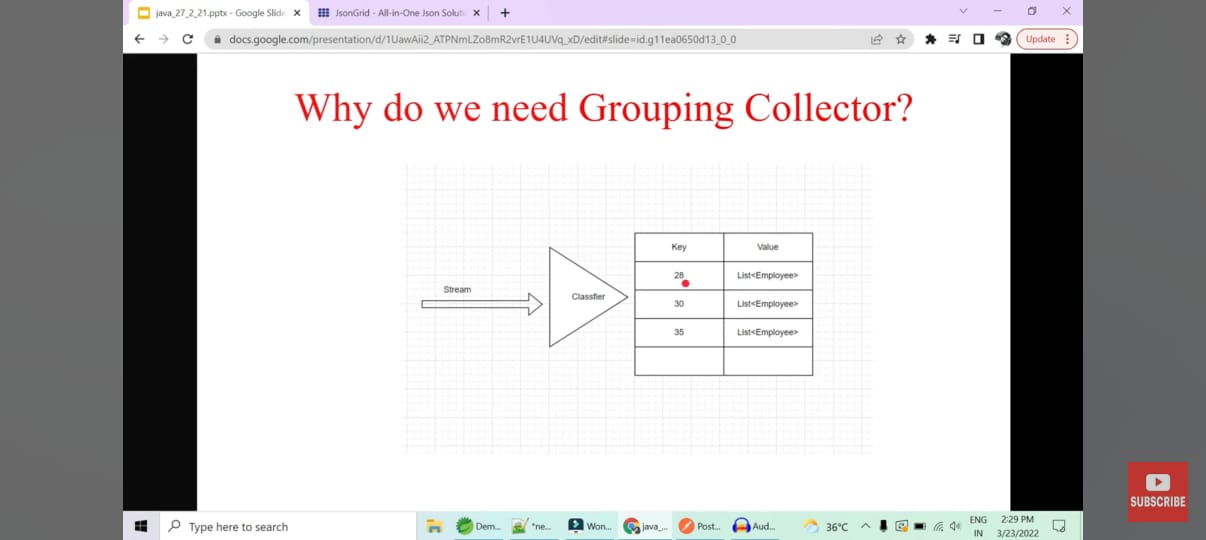


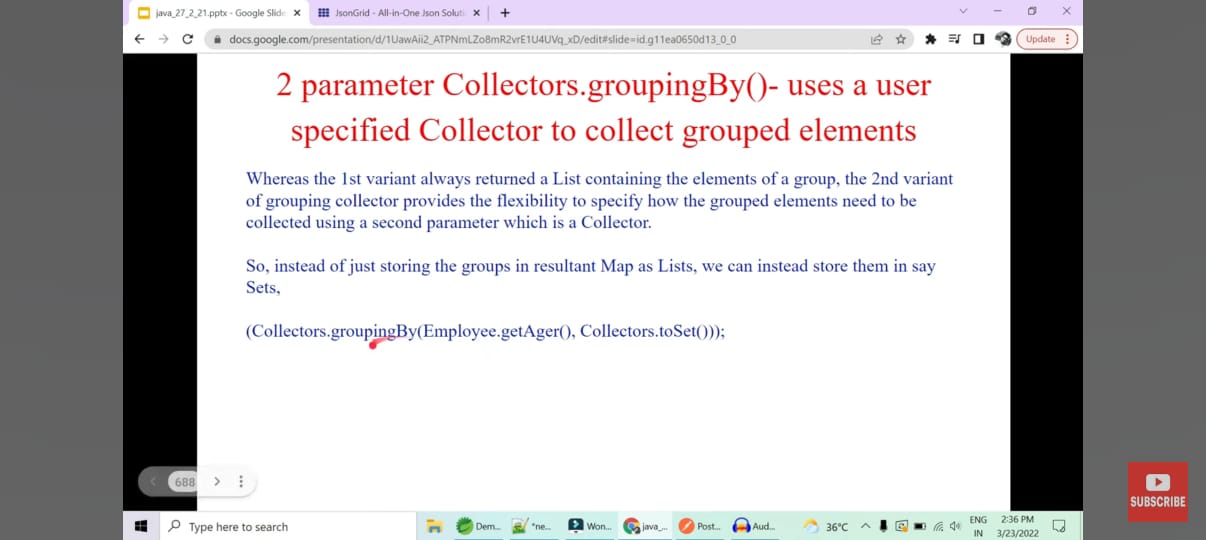


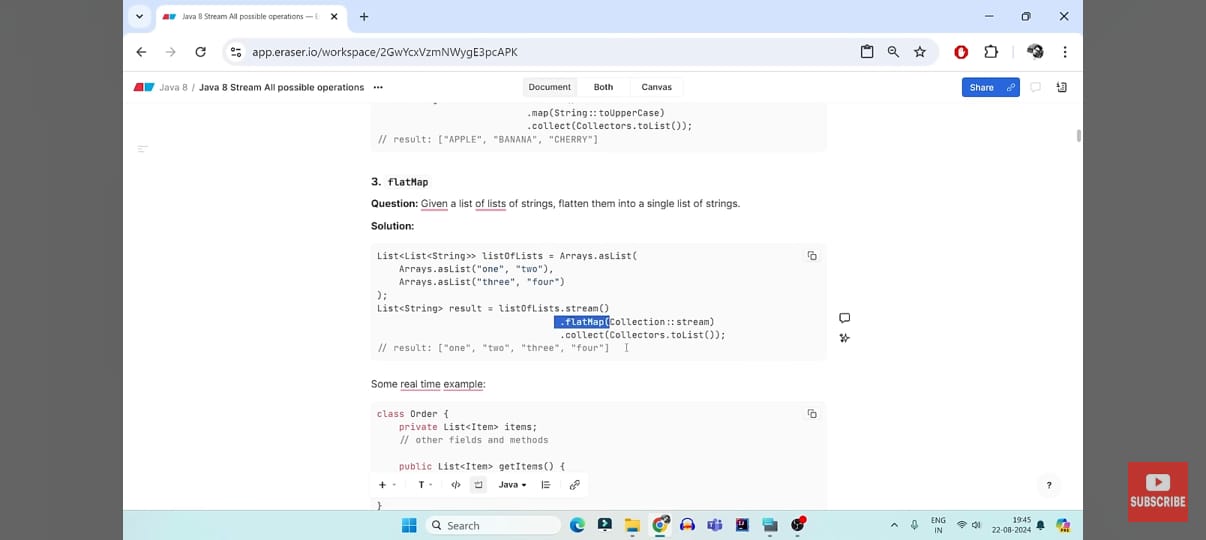


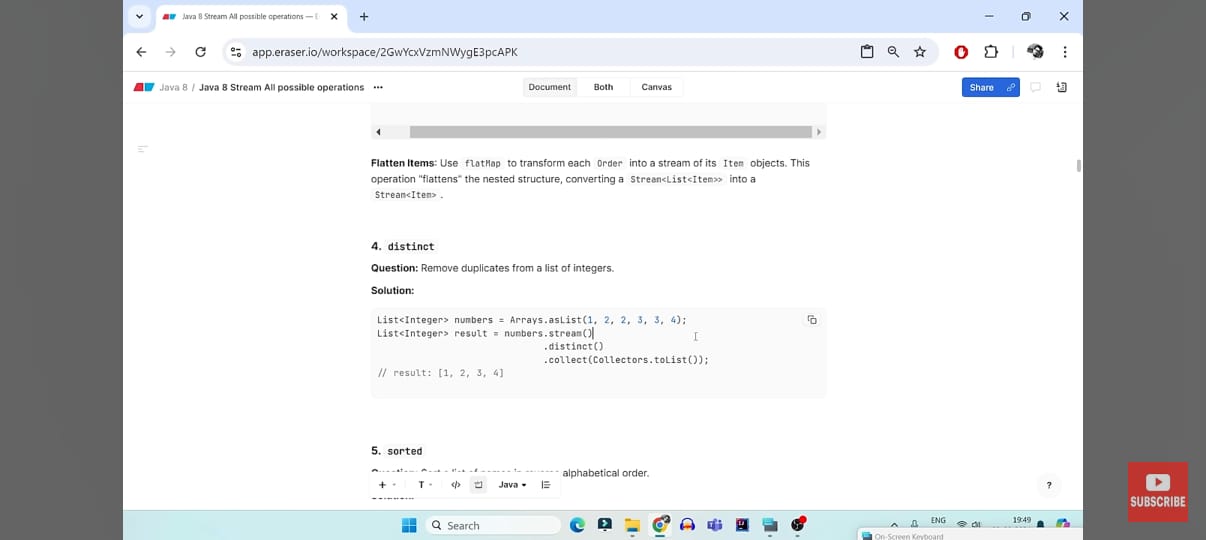


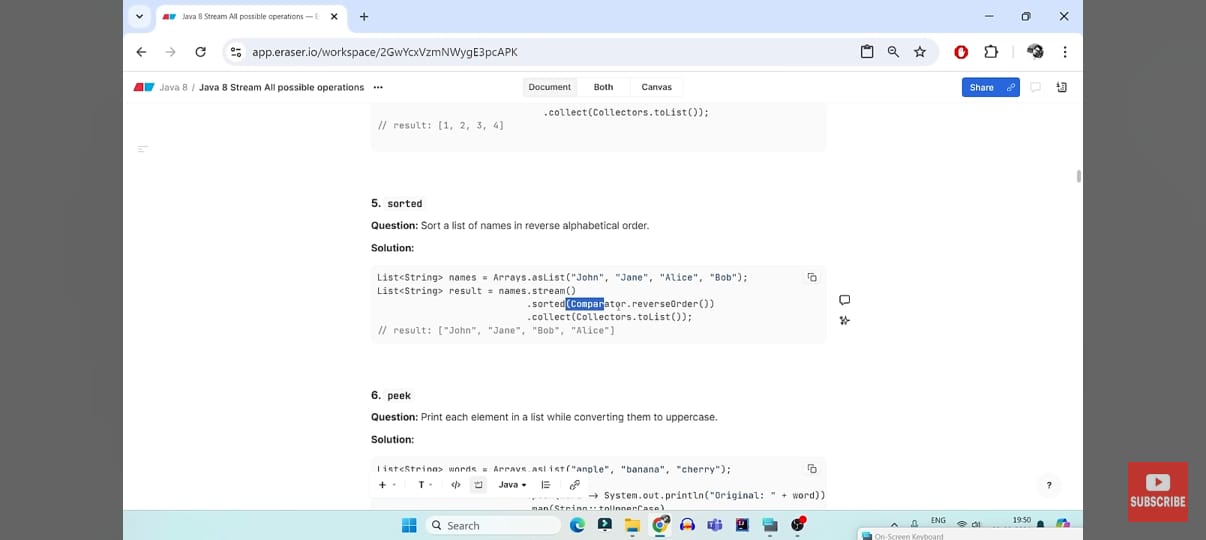


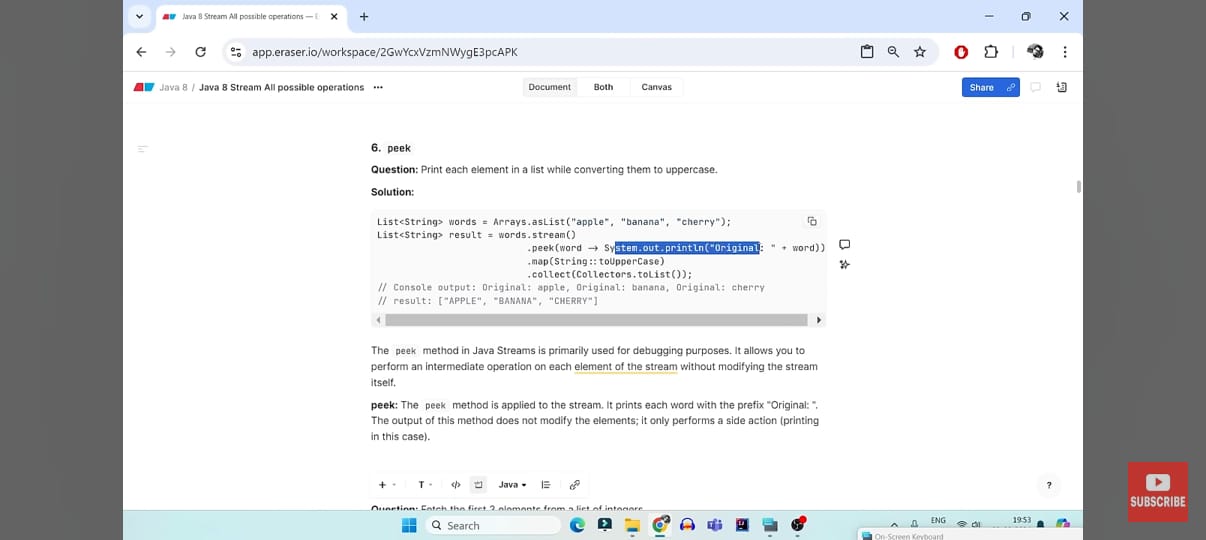


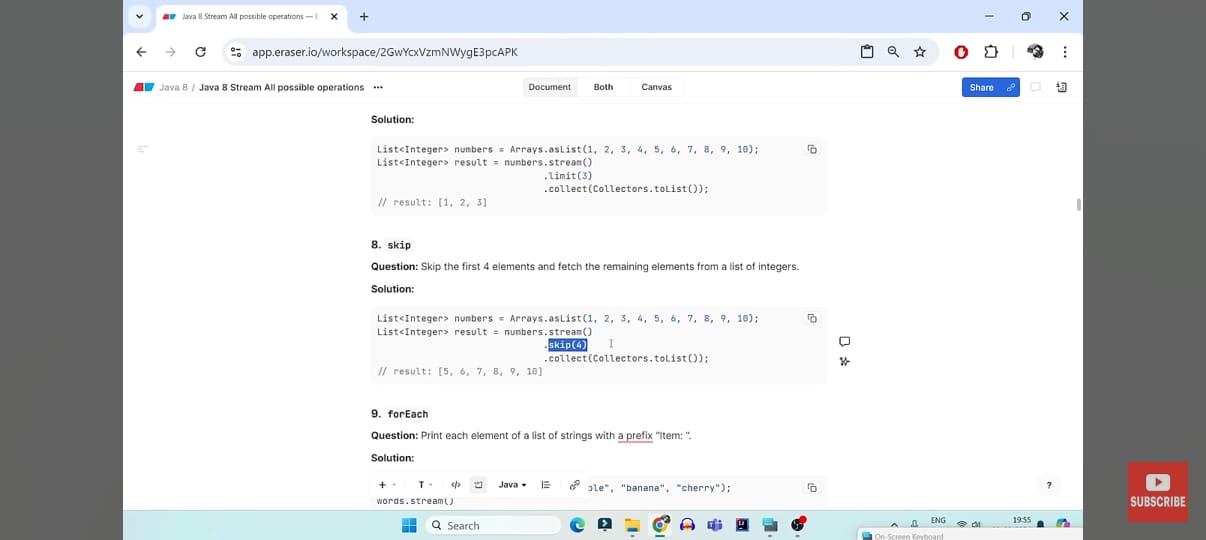


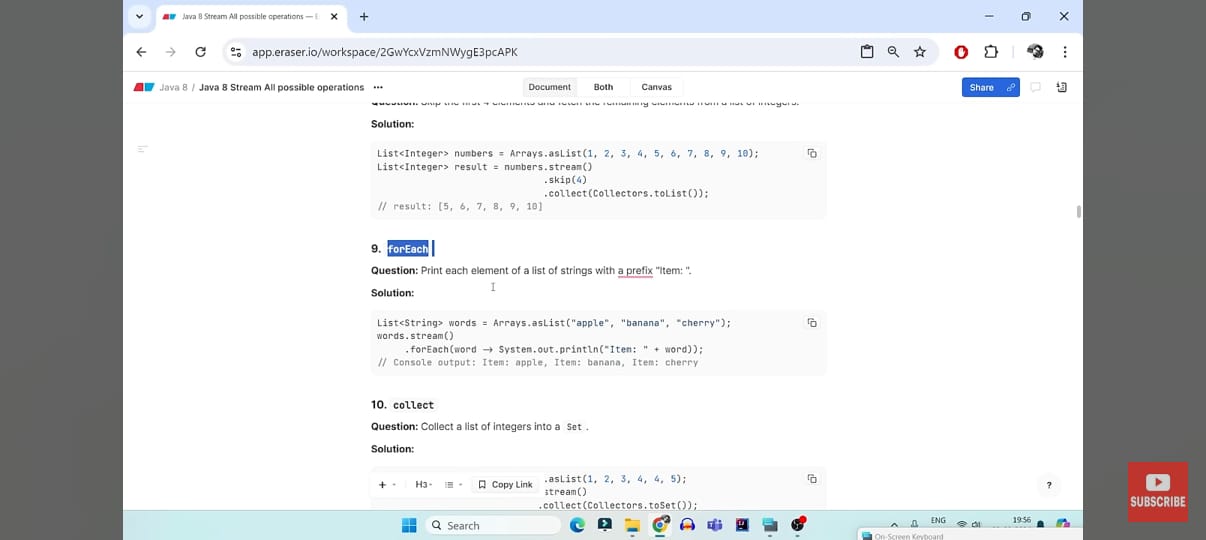


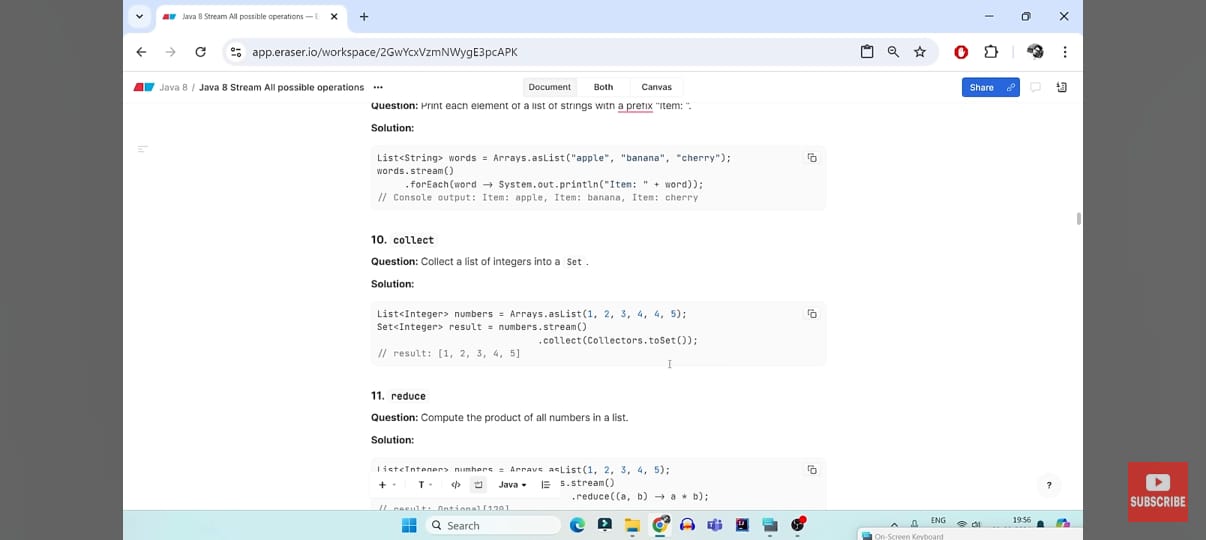


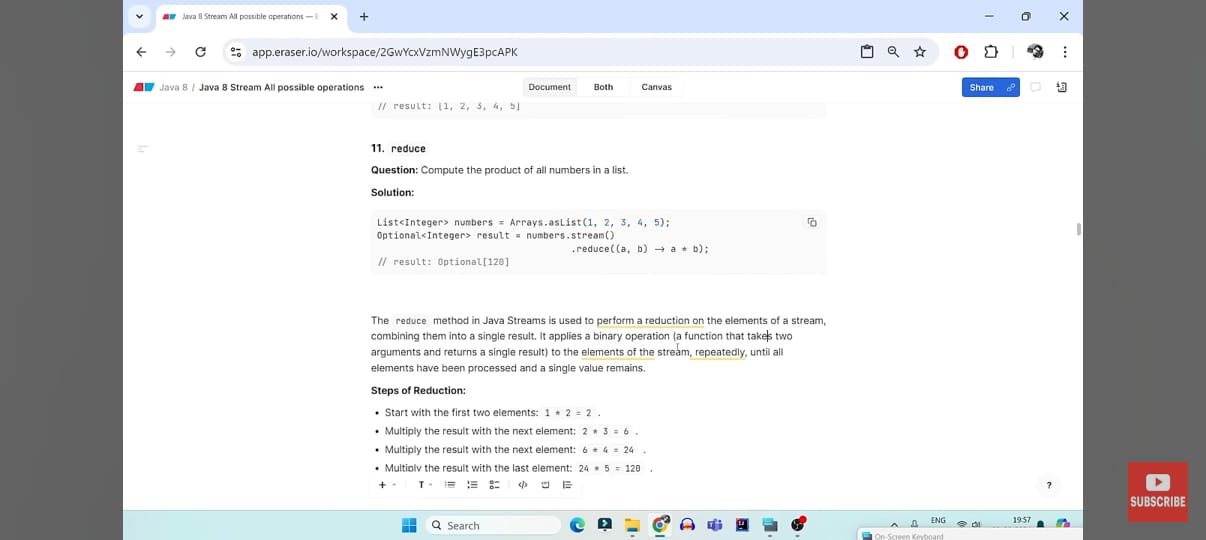


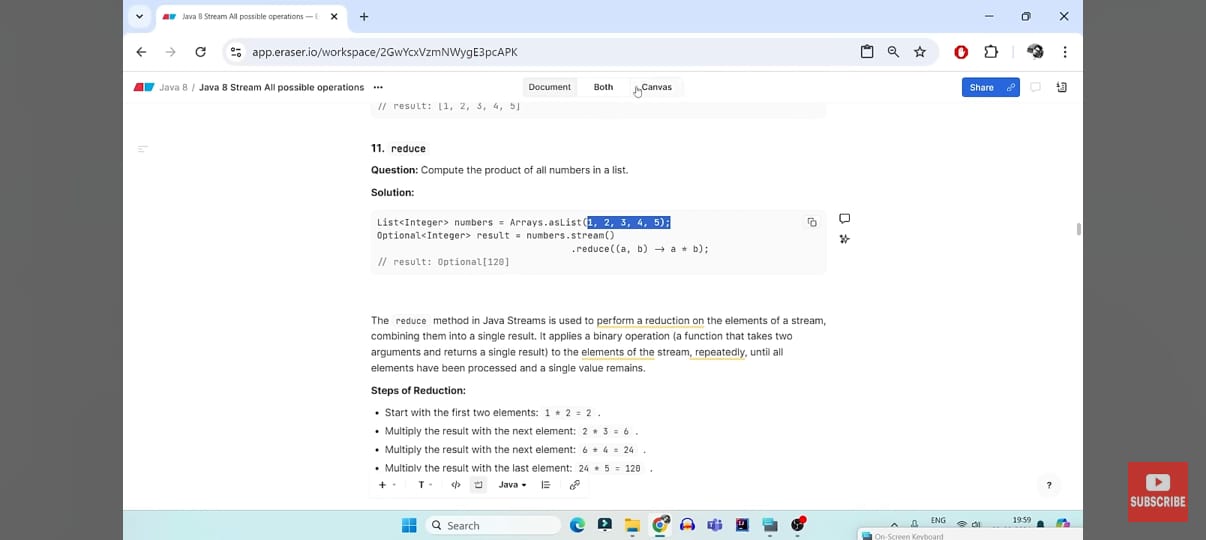


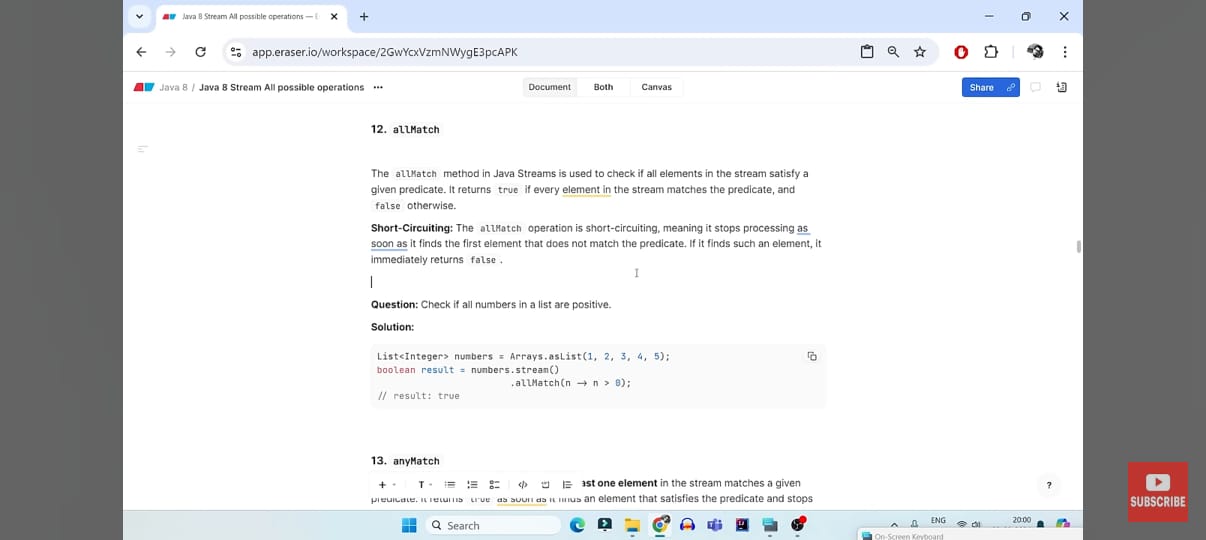


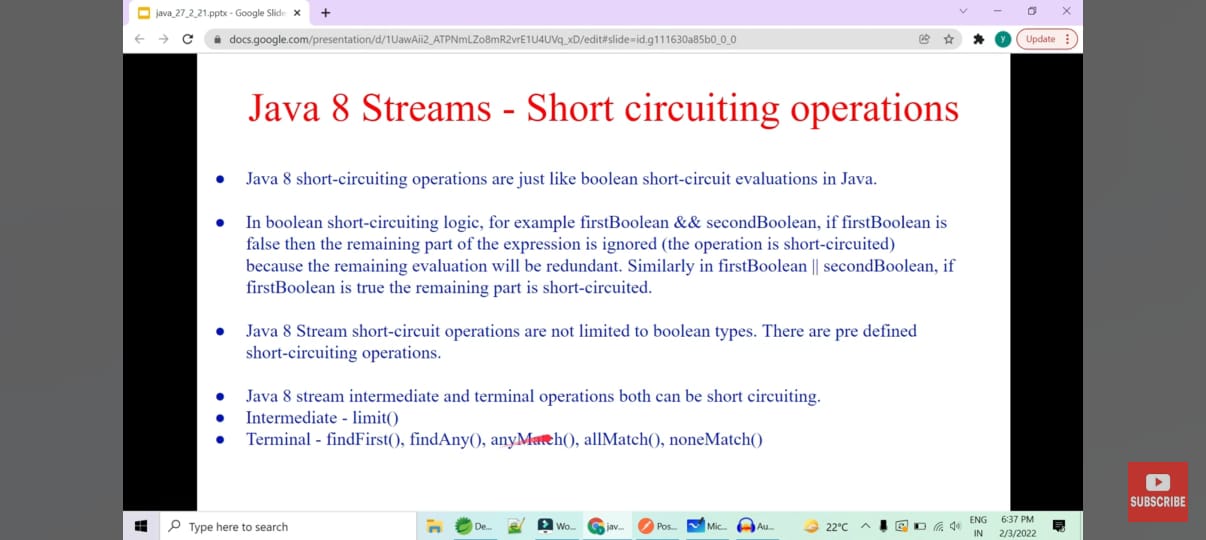


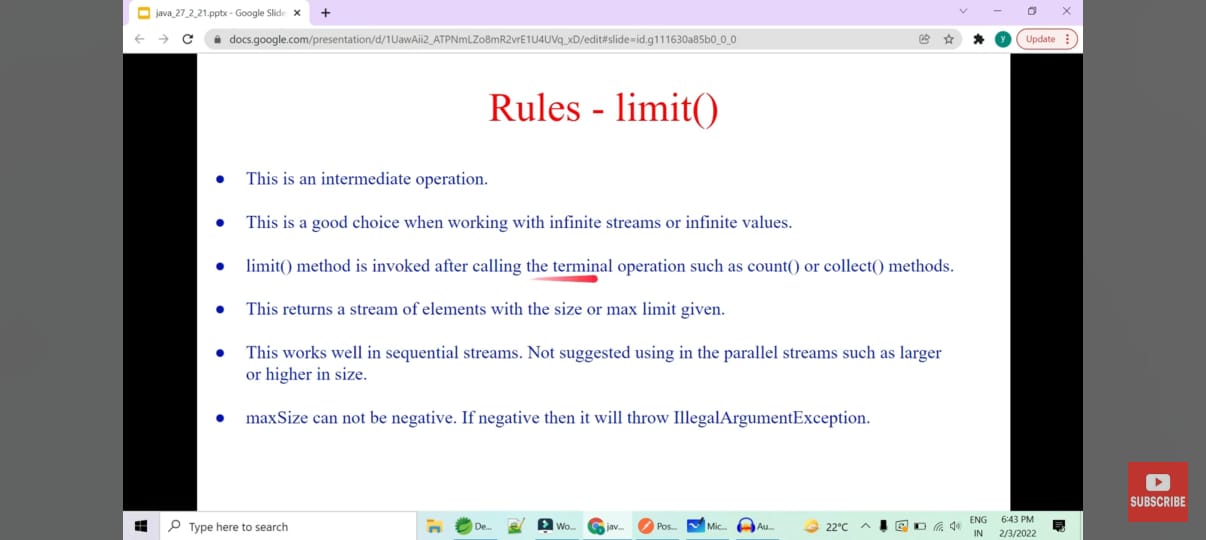


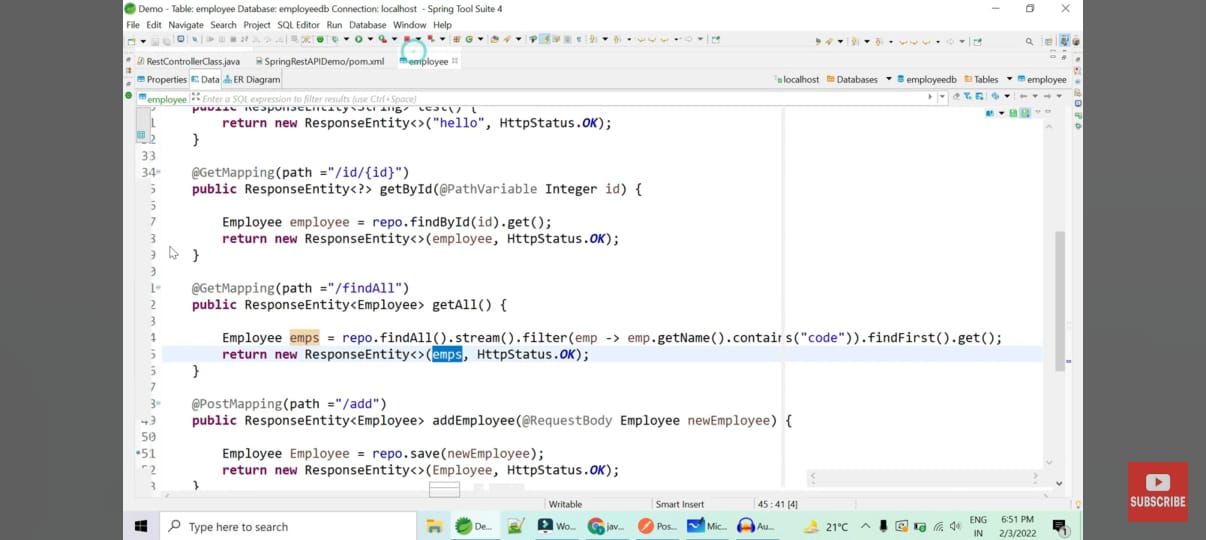


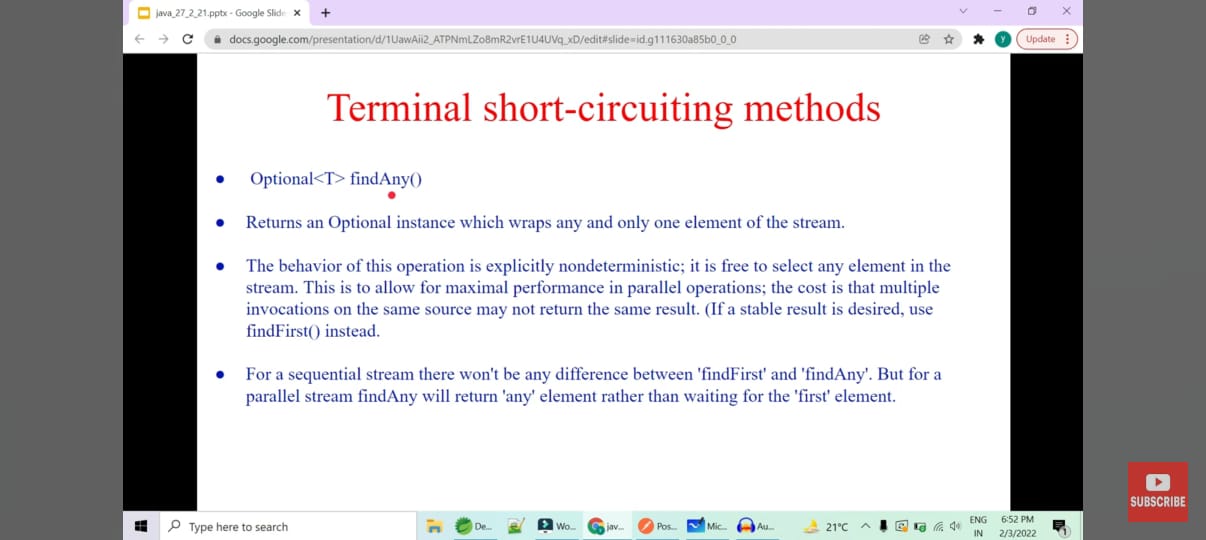


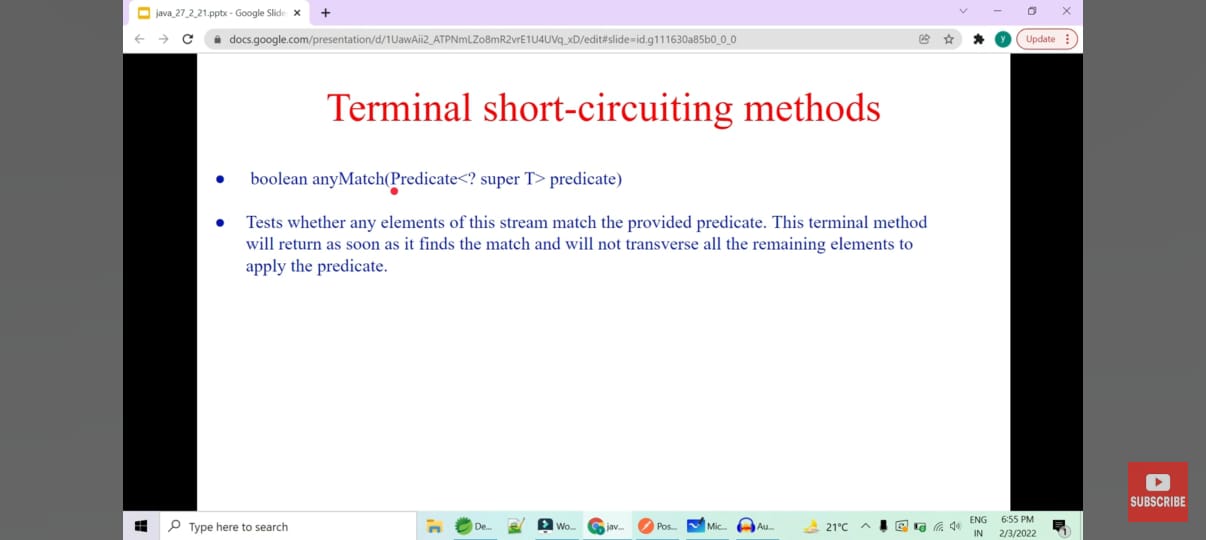


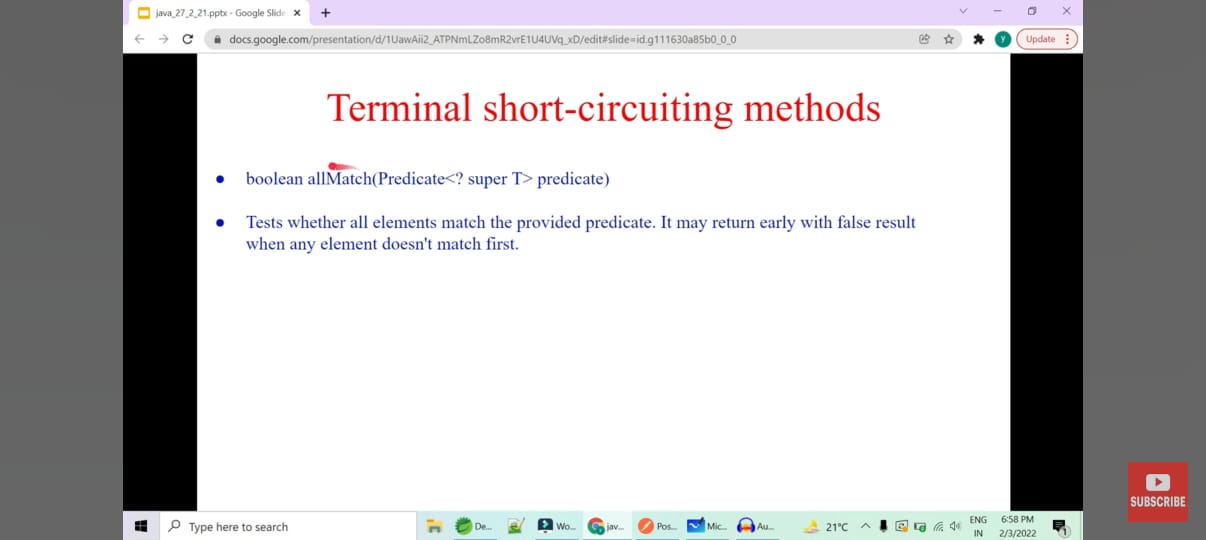


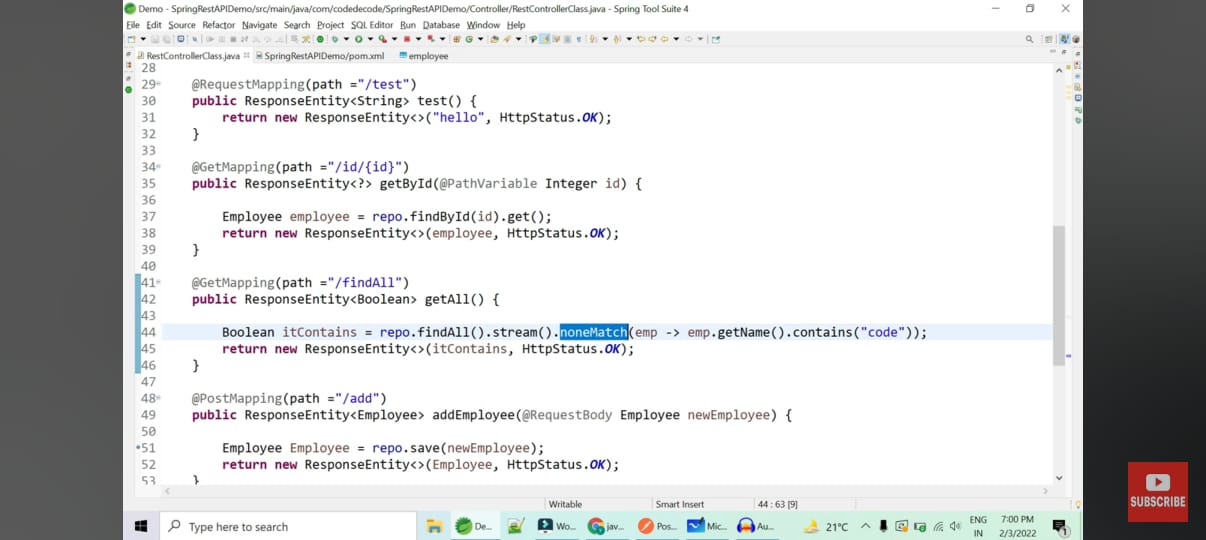


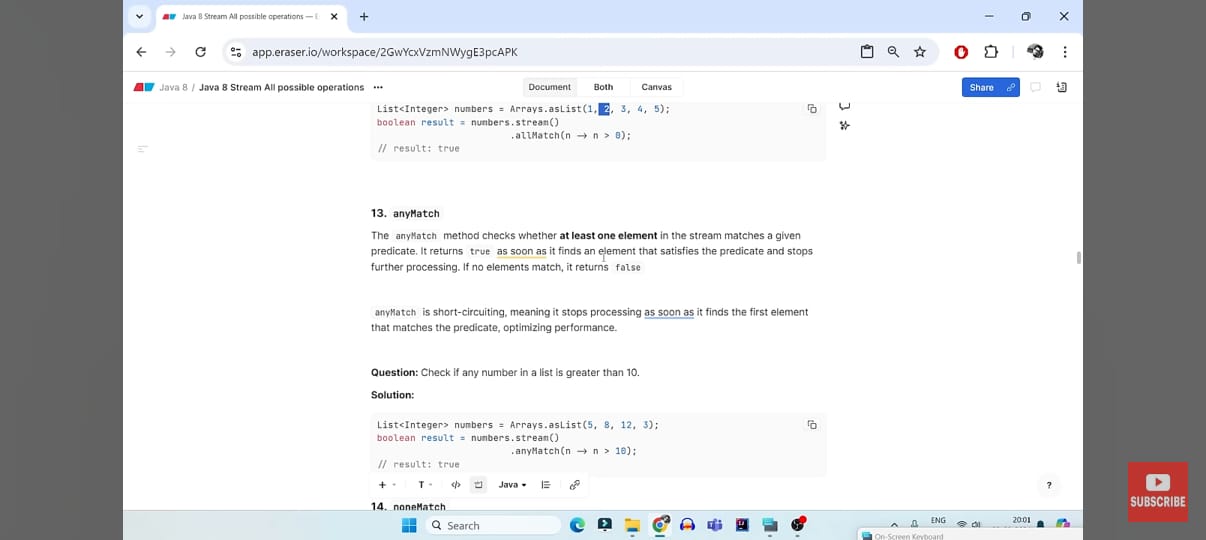


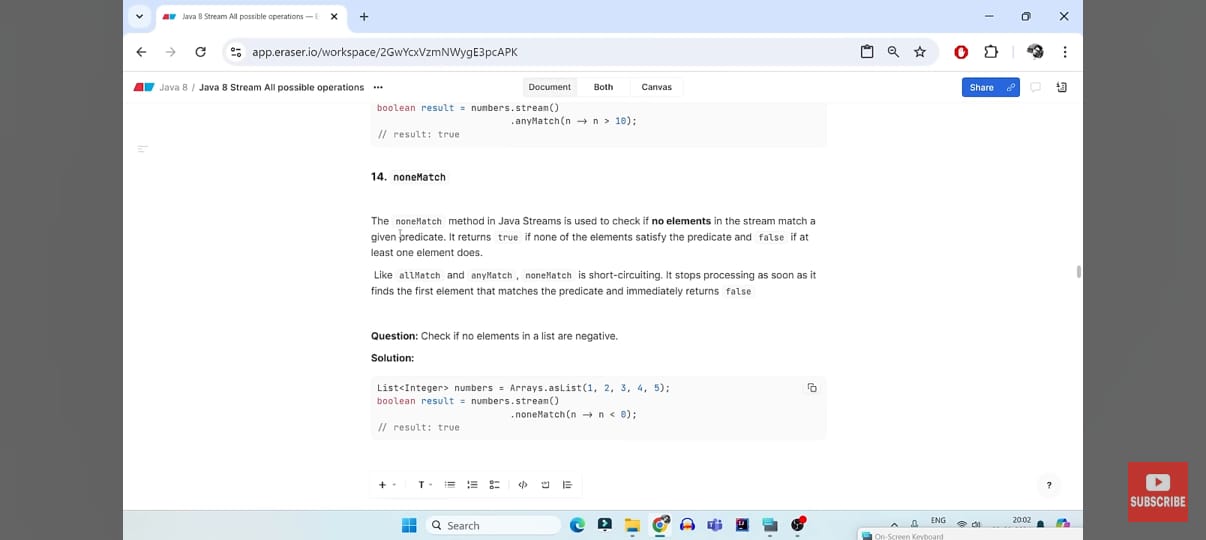


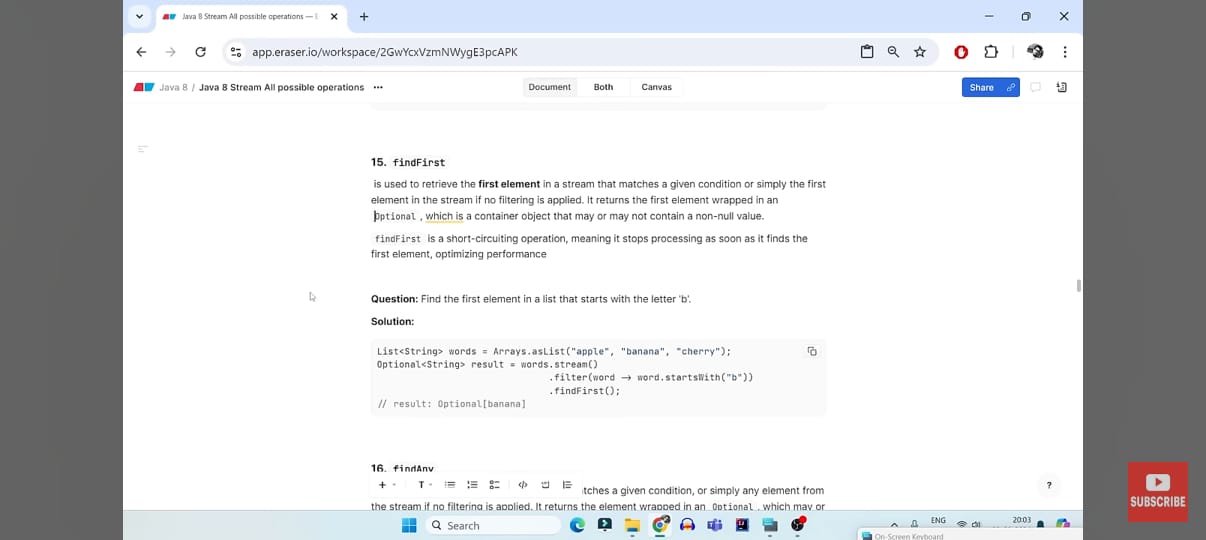


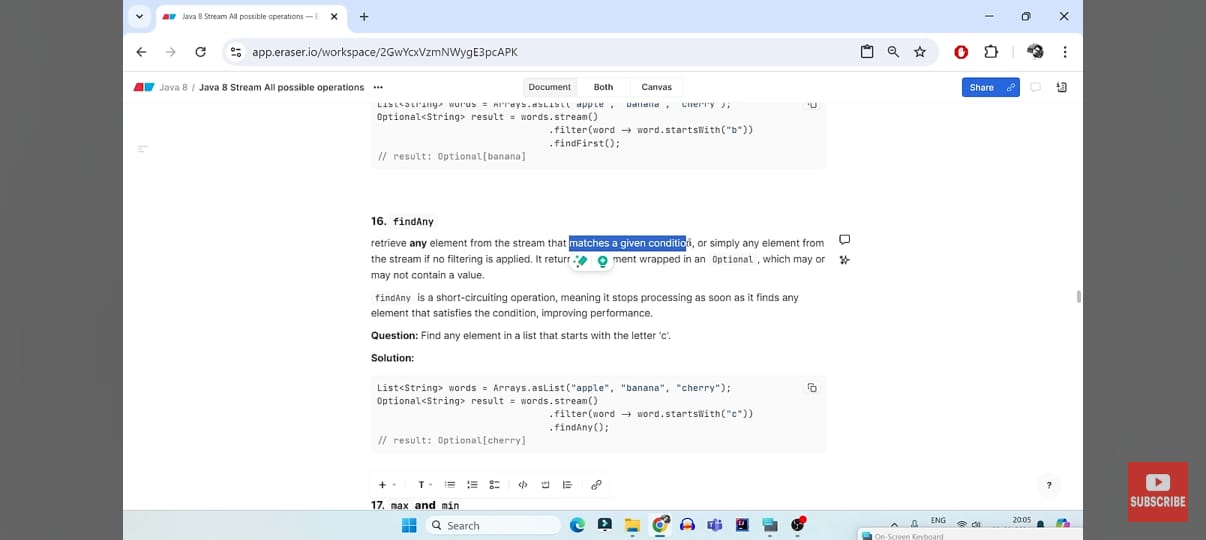


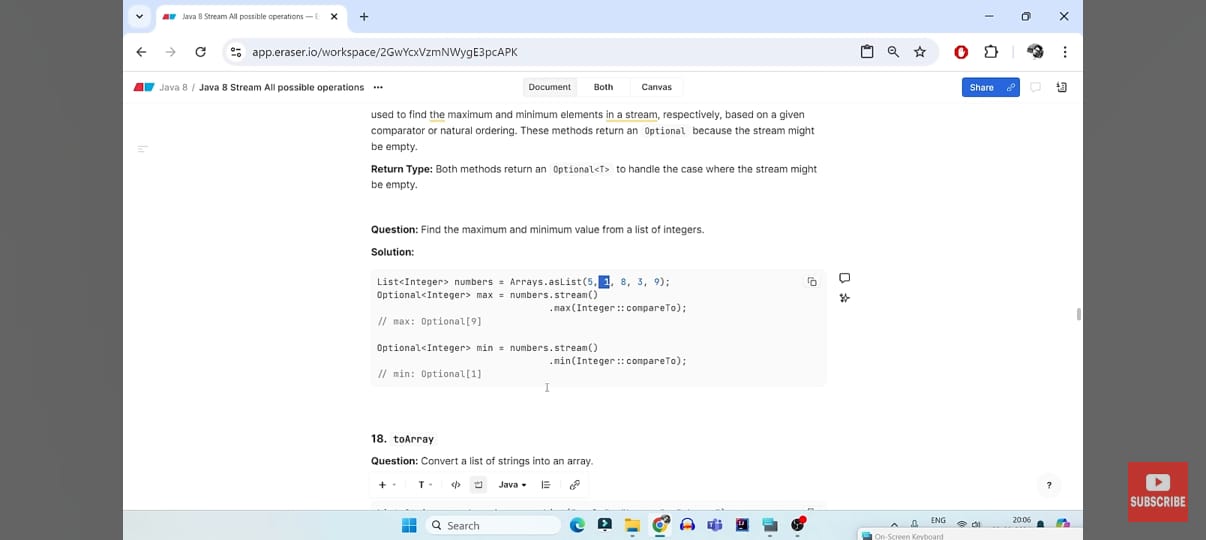


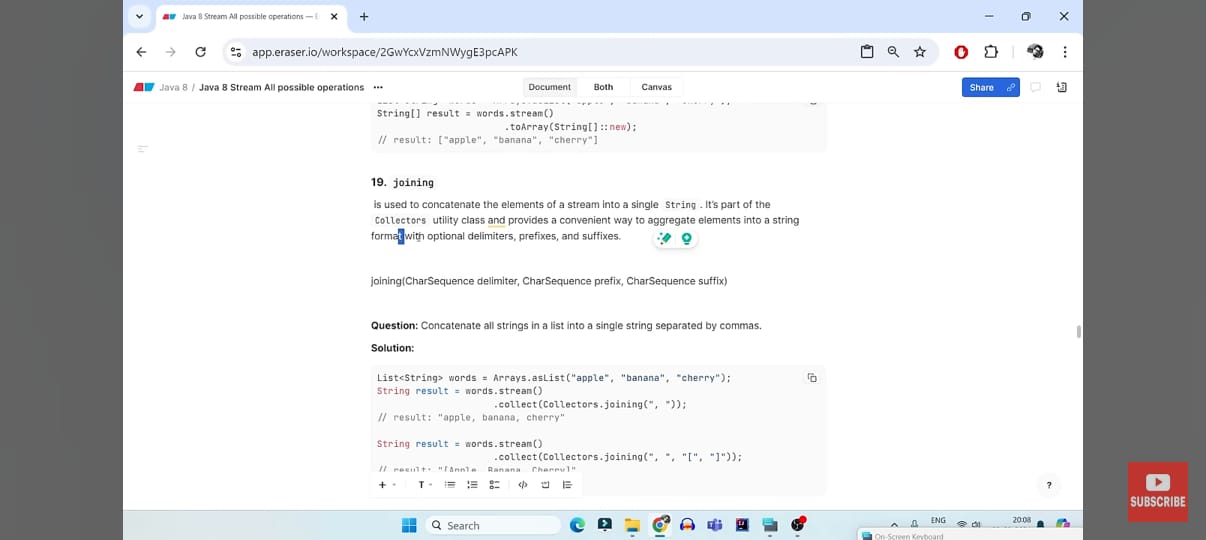


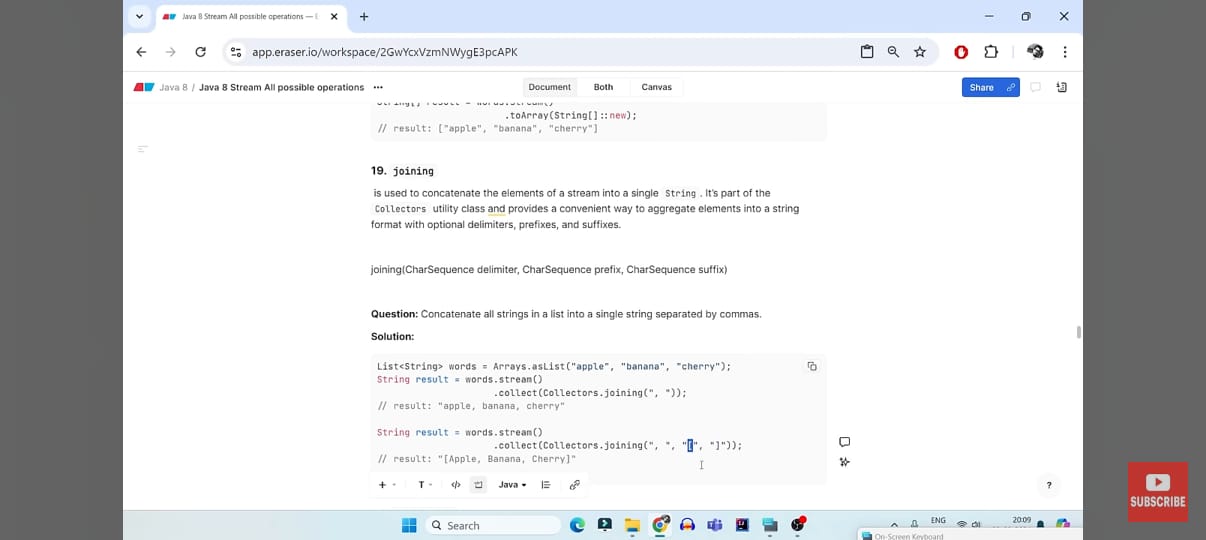


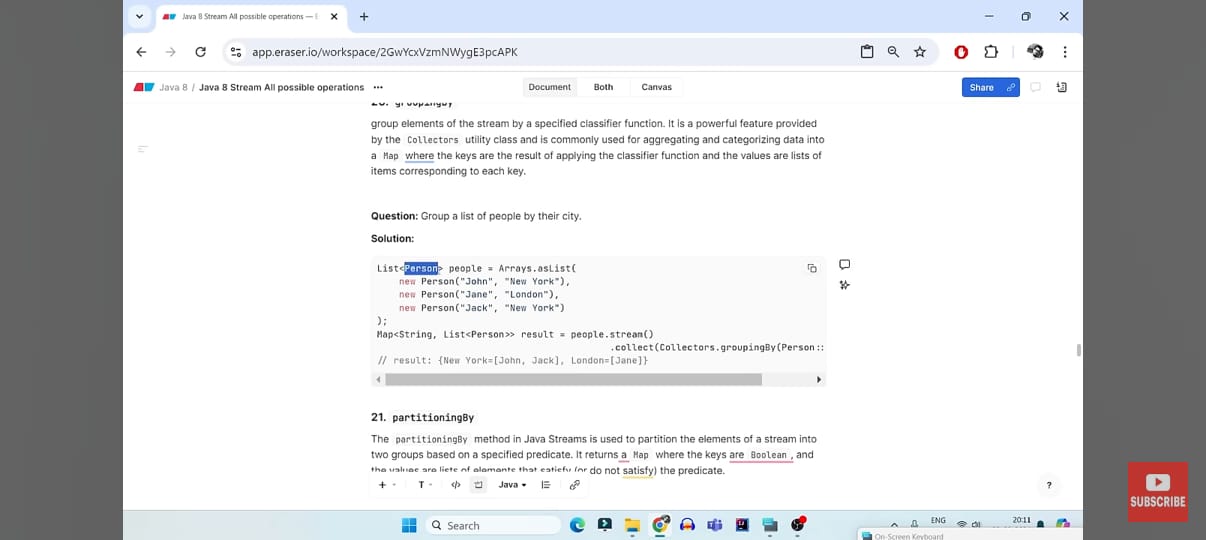


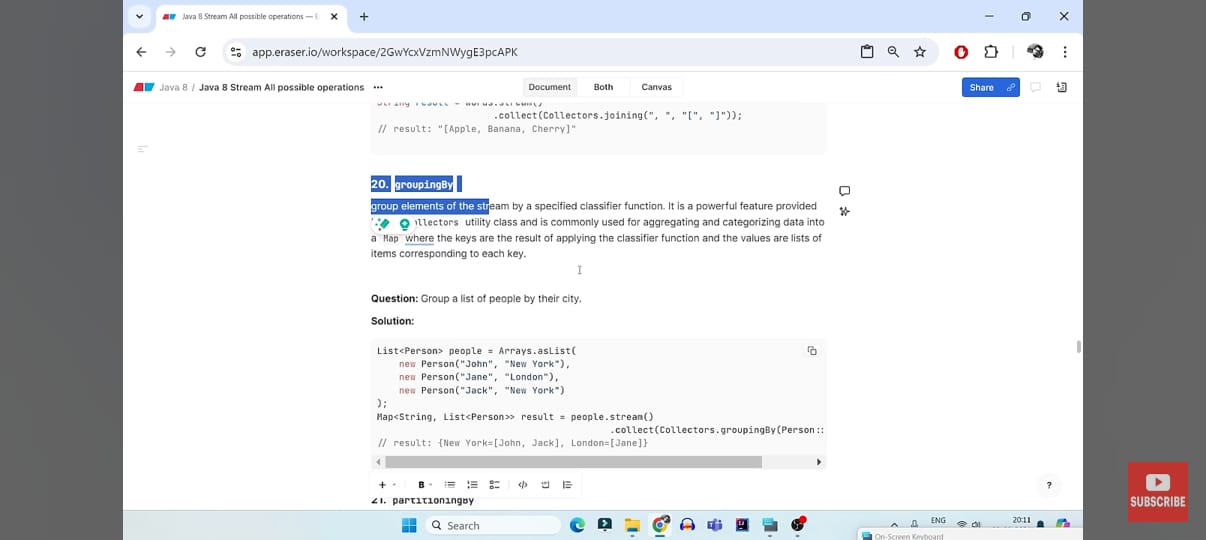














In Java, the Comparator interface is part of the java.util package and is used to **compare two objects** for ordering. It’s often used in sorting collections and streams.

## ✅ Commonly Used **Class Methods** in Comparator Interface (Java 8+)

These are **static** and **default** methods added in Java 8 to the Comparator interface:

### 🔹 Comparator.naturalOrder()

Returns a comparator that compares Comparable objects in **ascending (natural) order**.

Comparator<Integer> cmp = Comparator.naturalOrder();

List<Integer> list = Arrays.asList(4, 2, 7, 1);

Collections.sort(list, cmp); // [1, 2, 4, 7]

### 🔹 Comparator.reverseOrder()

Returns a comparator that imposes the **reverse of natural order**.

Comparator<String> cmp = Comparator.reverseOrder();

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

Collections.sort(names, cmp); // [John, Bob, Alice]

### 🔹 Comparator.comparing(Function<T, U>)

Creates a comparator that compares objects by a specified **key extractor function**.

class Person {

String name;

int age;

Person(String name, int age) { this.name = name; this.age = age; }

}

List<Person> people = Arrays.asList(

new Person("Alice", 30),

new Person("Bob", 25)

);

// Compare by age

people.sort(Comparator.comparing(p -> p.age));

### 🔹 thenComparing(...)

Used to chain comparators. If two elements are equal by the first comparator, this will use the next one.

people.sort(Comparator

.comparing((Person p) -> p.name)

.thenComparing(p -> p.age));

### 🔹 Comparator.nullsFirst(Comparator) / Comparator.nullsLast(Comparator)

Handles null values when comparing.

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

names.sort(Comparator.nullsFirst(Comparator.naturalOrder()));

// Output: [null, Alice, Bob]

### 🔹 reversed()

Returns a comparator with the **reverse ordering** of the current comparator.

Comparator<Person> byAge = Comparator.comparing(p -> p.age);

Comparator<Person> byAgeDesc = byAge.reversed();

## 🧠 Summary Table

| **Method** | **Description** |
| --- | --- |
| naturalOrder() | Sorts in natural ascending order |
| reverseOrder() | Sorts in descending order |
| comparing(...) | Compare using key extractor |
| thenComparing(...) | Chain multiple comparisons |
| nullsFirst(...) / nullsLast(...) | Handle null elements in sort |
| reversed() | Reverse any comparator |

Would you like a **practical example using Stream's sorted() with Comparator**?

### ✅ 1. Separate Odd and Even Numbers

java

CopyEdit

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

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

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

System.out.println("Even: " + result.get(true)); // [2, 4, 6]

System.out.println("Odd: " + result.get(false)); // [1, 3, 5]

* **partitioningBy()** returns a Map<Boolean, List<T>>.
* true → even numbers, false → odd numbers.

## Objective:

Ye code list ke numbers ko **2 groups (true/false)** mein **partition** karta hai — based on whether they are **even (even → true)** ya **odd (odd → false)**.

## 🔧 Breakdown:

### 1. Arrays.asList(...)

Creates a list of integers:  
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

### 2. .stream()

Stream bana deta hai list ke elements ka — ab hum functional operations apply kar sakte hain.

### 3. .collect(Collectors.partitioningBy(...))

Ye ek **terminal operation** hai jo stream ke elements ko **do groups** mein divide karta hai:

* true group: jinke liye condition true ho
* false group: jinke liye condition false ho

### 4. i -> i % 2 == 0

Yeh ek **lambda expression** hai:

* Har i ko check karega: agar i % 2 == 0 hai to **even number** (true)
* Nahi to **odd number** (false)

### 🔚 Result:

Code ka output hoga:

java

CopyEdit

{

false=[1, 3, 5, 7, 9], // odd numbers

true=[2, 4, 6, 8, 10] // even numbers

}

i.e. collect.get(true) → even numbers  
and collect.get(false) → odd numbers

## ✅ Summary:

* partitioningBy() hamesha Map<Boolean, List<T>> deta hai
* True → condition ke match hone wale elements
* False → baaki elements

### 👨‍💻 Bonus Tip:

Agar aap condition ko change kar do (jaise i > 5), to partitioning bhi waisa hi ho jaayega!

Chaho to uska example bhi de sakta hoon.

### ✅ 2. Remove Duplicate Elements from List

java

CopyEdit

List<String> names = Arrays.asList("Java", "Python", "Java", "C++");

List<String> unique = names.stream()

.distinct()

.collect(Collectors.toList());

System.out.println(unique); // [Java, Python, C++]

* **distinct()** removes duplicates.

### 3. Frequency of Each Character in String

java

CopyEdit

String input = "banana";

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

.mapToObj(c -> (char) c)

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

System.out.println(freq); // {a=3, b=1, n=2}

* **chars()** → int stream of char values.
* Group by character → count.

**Step-by-Step Explanation**

java

CopyEdit

input.chars()

* input.chars() returns an **IntStream** of Unicode values of characters in the string "banana".
  + Example: 'b' → 98, 'a' → 97, 'n' → 110

java

CopyEdit

.mapToObj(c -> (char) c)

* Converts each int (Unicode value) to a Character.
* **Why?** Because .chars() returns IntStream, not Stream<Character>, and we want to work with characters.

👉 Example:

java

CopyEdit

// Before mapToObj

IntStream: [98, 97, 110, 97, 110, 97]

// After mapToObj

Stream<Character>: ['b', 'a', 'n', 'a', 'n', 'a']

java

CopyEdit

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

This is a **terminal operation** that collects the stream into a Map:

* **groupingBy(Function.identity())**:
  + Groups the elements by themselves — here, each character becomes a key in the Map.
* **Collectors.counting()**:
  + Counts how many times each character appears.

**✅ Final Output**

For "banana":

* 'a' appears 3 times
* 'b' appears 1 time
* 'n' appears 2 times

So the output is:

java

CopyEdit

{a=3, b=1, n=2}

**🧠 Summary**

| **Part** | **Meaning** |
| --- | --- |
| .chars() | Converts string to stream of Unicode ints |
| .mapToObj(c -> (char) c) | Convert ints to characters |
| .collect(Collectors.groupingBy(...)) | Group and count each character |

### Option 1: ****Using**** for ****loop and**** Map ****(Simple & Readable)****

java

CopyEdit

import java.util.\*;

public class CharFrequency {

public static void main(String[] args) {

String input = "banana";

Map<Character, Integer> freq = new HashMap<>();

for (char ch : input.toCharArray()) {

freq.put(ch, freq.getOrDefault(ch, 0) + 1);

}

System.out.println(freq); // Output: {a=3, b=1, n=2}

}

}

#### 🔍 Explanation:

* toCharArray() gives an array of characters.
* getOrDefault(ch, 0) + 1 increments the count.
* Simple logic and **no need to understand Streams or lambdas**.

### ✅ Option 2: ****Using**** TreeMap ****if you want sorted keys****

java

CopyEdit

Map<Character, Integer> freq = new TreeMap<>();

for (char ch : input.toCharArray()) {

freq.put(ch, freq.getOrDefault(ch, 0) + 1);

}

System.out.println(freq); // {a=3, b=1, n=2}

### 4. Frequency of Each Element in an Array

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

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

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

System.out.println(freq); // {orange=1, banana=1, apple=3}

### 5. Sort List in Reverse Order

java

CopyEdit

List<Integer> list = Arrays.asList(5, 2, 9, 1);

list.stream()

.sorted(Comparator.reverseOrder())

.forEach(System.out::println);

* Sorts in descending order.

### 6. Join Strings with Prefix, Suffix, and Delimiter

java

CopyEdit

List<String> list = Arrays.asList("Java", "Python", "Go");

String result = list.stream()

.collect(Collectors.joining(", ", "Start: ", " :End"));

System.out.println(result); // Start: Java, Python, Go :End

### 7. Print Multiples of 5

java

CopyEdit

List<Integer> list = Arrays.asList(10, 15, 22, 33, 50);

list.stream()

.filter(i -> i % 5 == 0)

.forEach(System.out::println);

### 8. Max and Min in a List

java

CopyEdit

List<Integer> list = Arrays.asList(3, 8, 1, 9);

int max = list.stream().max(Integer::compare).get();

int min = list.stream().min(Integer::compare).get();

System.out.println("Max: " + max); // 9

System.out.println("Min: " + min); // 1

For Maximum number :

Integer collect = Arrays.*asList*(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)  
 .stream()  
 .max((s1,s2)-> s1.compareTo(s2)).get();

System.*out*.println("Even numbers: " + collect);

For minimum number:

Integer collect = Arrays.*asList*(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)  
 .stream()  
 .min((s1,s2)-> s1.compareTo(s2)).get();  
  
 System.*out*.println("Even numbers: " + collect);

### 9. Merge Two Arrays (Sorted)

java

CopyEdit

Integer[] a = {1, 5, 3};

Integer[] b = {4, 2, 6};

Integer[] merged = Stream.concat(Arrays.stream(a), Arrays.stream(b))

.sorted()

.toArray(Integer[]::new);

System.out.println(Arrays.toString(merged)); // [1, 2, 3, 4, 5, 6]

### Code:

java

CopyEdit

Integer[] a = {1, 5, 3};

Integer[] b = {4, 2, 6};

✅ **Explanation**:  
You define two arrays of Integer type:

* a = [1, 5, 3]
* b = [4, 2, 6]

These arrays will be **merged and sorted**.

### 🔸 Code:

java

CopyEdit

Integer[] merged = Stream.concat(Arrays.stream(a), Arrays.stream(b))

.sorted()

.toArray(Integer[]::new);

✅ Let's break this entire statement line by line:

#### 1️⃣ Arrays.stream(a)

* Converts array a into a **Stream**:

arduino

CopyEdit

Stream: [1, 5, 3]

#### 2️⃣ Arrays.stream(b)

* Converts array b into another **Stream**:

arduino

CopyEdit

Stream: [4, 2, 6]

#### 3️⃣ Stream.concat(..., ...)

* Combines (merges) the two streams a and b into one stream:

arduino

CopyEdit

Stream: [1, 5, 3, 4, 2, 6]

#### 4️⃣ .sorted()

* Sorts the merged stream in **natural order (ascending)**:

arduino

CopyEdit

Stream: [1, 2, 3, 4, 5, 6]

#### 5️⃣ .toArray(Integer[]::new)

* Converts the sorted stream into a **new Integer array** using a method reference.
* Equivalent to:

java

CopyEdit

.toArray(size -> new Integer[size])

✅ So now merged is:

java

CopyEdit

[1, 2, 3, 4, 5, 6]

### 🔸 Code:

java

CopyEdit

System.out.println(Arrays.toString(merged));

✅ Prints the array in readable format:

java

CopyEdit

[1, 2, 3, 4, 5, 6]

### 10. Merge Arrays Without Duplicates

java

CopyEdit

Integer[] merged = Stream.concat(Arrays.stream(a), Arrays.stream(b))

.distinct()

.toArray(Integer[]::new);

### 11. Three Max and Min Numbers

java

CopyEdit

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

List<Integer> top3 = list.stream()

.sorted(Comparator.reverseOrder())

.limit(3)

.collect(Collectors.toList());

System.out.println(top3); // [9, 5, 3]

### 12. Sort List by Length

java

CopyEdit

List<String> list = Arrays.asList("Java", "C", "Python");

list.stream()

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

.forEach(System.out::println); // C Java Python

### 13. Sum & Average of Array Elements

java

CopyEdit

int[] arr = {1, 2, 3, 4, 5};

int sum = Arrays.stream(arr).sum();

double avg = Arrays.stream(arr).average().orElse(0.0);

System.out.println("Sum: " + sum); // 15

System.out.println("Avg: " + avg); // 3.0

### 16. Last Element of List

java

CopyEdit

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

String last = list.stream().skip(list.size() - 1).findFirst().orElse(null);

System.out.println(last); // c

### 18. Sum of All Digits

java

CopyEdit

int num = 1234;

int sum = String.valueOf(num).chars()

.map(Character::getNumericValue)

.sum();

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

Absolutely! Let's break this Java program down line by line — it's a concise and modern way to calculate the **sum of all digits** in an integer using **Java 8 streams**.

### ✅ Program:

int num = 1234;

int sum = String.valueOf(num).chars()

.map(Character::getNumericValue)

.sum();

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

## 🔍 Step-by-Step Explanation:

### 🔹 int num = 1234;

You declare an integer variable num and assign the value 1234 to it.

### 🔹 String.valueOf(num)

This converts the number 1234 into the **String** "1234".

👉 Why?  
Because the chars() method works on **strings**, not on integers.

### 🔹 .chars()

* This returns an IntStream of **Unicode values (int)** of each character in the string "1234".

So:

"1234".chars() → [49, 50, 51, 52]

These numbers are the **ASCII (Unicode) codes** for '1', '2', '3', '4'.

### 🔹 .map(Character::getNumericValue)

This maps (converts) each character **code** into its corresponding digit value.

So the stream becomes:

[1, 2, 3, 4]

Explanation:

* '1' → 1
* '2' → 2
* '3' → 3
* '4' → 4

Character.getNumericValue(int ch) handles this mapping.

### 🔹 .sum()

Adds up all values in the stream:

1 + 2 + 3 + 4 = 10

So, sum = 10.

### 🔹 System.out.println(sum);

Prints the final sum:

10

## ✅ Final Output:

10

## 🔁 Equivalent Traditional Way (for comparison):

int num = 1234;

int sum = 0;

while (num != 0) {

sum += num % 10; // get last digit

num /= 10; // remove last digit

}

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

### 19. Second Highest Number

java

CopyEdit

List<Integer> list = Arrays.asList(5, 9, 1, 3);

int secondMax = list.stream()

.sorted(Comparator.reverseOrder())

.skip(1)

.findFirst()

.get();

System.out.println(secondMax); // 5

### 20. Reverse Each Word in a String

java

CopyEdit

String str = "Java Stream API";

String result = Arrays.stream(str.split(" "))

.map(w -> new StringBuilder(w).reverse().toString())

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

System.out.println(result); // avaJ maertS IPA

Great! Let’s fully break down this program that **reverses each word in a string** using **Java 8 Stream API**.

### ✅ Code:

String str = "Java Stream API";

String result = Arrays.stream(str.split(" "))

.map(w -> new StringBuilder(w).reverse().toString())

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

System.out.println(result); // avaJ maertS IPA

## 🔍 Objective:

Take a sentence → "Java Stream API"  
Reverse each word individually → "avaJ maertS IPA"  
But **do not reverse the word order**.

## 🔎 Step-by-Step Explanation

### 🔹 String str = "Java Stream API";

We define the input string.

### 🔹 str.split(" ")

* Splits the sentence into an array of words by spaces:

["Java", "Stream", "API"]

### 🔹 Arrays.stream(...)

Converts the array of words into a **Stream**, so we can process each word using the Stream API.

### 🔹 .map(w -> new StringBuilder(w).reverse().toString())

This step is the core logic. It takes each word w and:

1. Creates a StringBuilder from the word → new StringBuilder(w)
2. Calls .reverse() → reverses the characters
3. Calls .toString() → converts it back to a normal String

### 🧠 Example:

| **Input Word** | **Reversed Word** |
| --- | --- |
| "Java" | "avaJ" |
| "Stream" | "maertS" |
| "API" | "IPA" |

### 🔹 .collect(Collectors.joining(" "))

* Joins all the reversed words back together into a single string
* Uses " " (space) as the separator between words

Resulting in:

"avaJ maertS IPA"

### 🔹 System.out.println(result);

Prints the final output:

avaJ maertS IPA

## 🧪 Output:

avaJ maertS IPA

## 🔁 Traditional Way (For Comparison):

String str = "Java Stream API";

String[] words = str.split(" ");

StringBuilder result = new StringBuilder();

for (String word : words) {

result.append(new StringBuilder(word).reverse()).append(" ");

}

System.out.println(result.toString().trim()); // avaJ maertS IPA

### 21. Sum of First 10 Numbers

java

CopyEdit

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

System.out.println(sum); // 55

| **Method** | **Includes End?** | **Output** |
| --- | --- | --- |
| range(1, 5) | ❌ No (exclusive) | 1 2 3 4 |
| rangeClosed(1, 5) | ✅ Yes (inclusive) | 1 2 3 4 5 |

Sure! Let's break down the Java program you shared:

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

System.out.println(sum); // 55

### 🔍 Explanation:

This program calculates the **sum of the first 10 natural numbers** (i.e., from 1 to 10), using **Java 8's Stream API**.

### ✅ Step-by-step Breakdown:

#### 1. IntStream.rangeClosed(1, 10)

* This is a static method from the java.util.stream.IntStream class.
* It creates a stream of **int values** starting from 1 to 10 **inclusive**.
  + So it produces: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

⚠️ Note: If you use IntStream.range(1, 10) instead, it will go from 1 to **9**, excluding 10.

#### 2. .sum()

* This is a **terminal operation**.
* It calculates the **sum of all elements in the stream**.
* So it adds up: 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = 55

#### 3. System.out.println(sum);

* It prints the result 55 to the console.

### 🧠 Concept Used:

* This is a **functional programming approach** using Java Streams.
* It's cleaner and more expressive than using a traditional loop.

### 💡 Traditional Equivalent:

If you wrote this without streams, it would look like:

int sum = 0;

for (int i = 1; i <= 10; i++) {

sum += i;

}

System.out.println(sum); // 55

Let me know if you want a version with debugging/printing inside the stream for learning purposes!

✅ 22. Strings Starting with Number

```java

List<String> list = Arrays.asList("1Java", "Python", "2Go");

list.stream()

.filter(s -> Character.isDigit(s.charAt(0)))

.forEach(System.out::println);

✅ 23. Find Duplicates in List

```java

List<String> names = Arrays.asList("Java", "Python", "Java");

Set<String> set = new HashSet<>();

names.stream()

.filter(e -> !set.add(e))

.forEach(System.out::println); // Java

### ✅ What is RestTemplate in Spring Boot?

RestTemplate is a **synchronous HTTP client** provided by Spring to perform **HTTP requests (GET, POST, PUT, DELETE, etc.)** from your Java application to **consume RESTful web services**.

Think of it as a way for your Spring Boot app to **call another API/server**.

## 🔍 Common Use Cases of RestTemplate:

| **Purpose** | **HTTP Method** | **Example Method** |
| --- | --- | --- |
| Get data from API | GET | getForObject(), getForEntity() |
| Send data to API | POST | postForObject(), postForEntity() |
| Update data on API | PUT | put() |
| Delete data | DELETE | delete() |

## ✅ Example 1: GET Request

@RestController

public class MyController {

@Autowired

private RestTemplate restTemplate;

@GetMapping("/get-user")

public String getUser() {

String url = "https://jsonplaceholder.typicode.com/users/1";

String result = restTemplate.getForObject(url, String.class);

return result;

}

}

## ✅ Example 2: POST Request

@PostMapping("/add-user")

public String createUser() {

String url = "https://example.com/api/users";

User user = new User("John", "john@example.com");

ResponseEntity<String> response = restTemplate.postForEntity(url, user, String.class);

return response.getBody();

}

## 🔧 How to Use It in Spring Boot

### 1. Add RestTemplate Bean

@Configuration

public class AppConfig {

@Bean

public RestTemplate restTemplate() {

return new RestTemplate();

}

}

Or use @SpringBootApplication class.

## 🔄 Common Methods of RestTemplate:

| **Method** | **Description** |
| --- | --- |
| getForObject() | Retrieves and returns response body directly |
| getForEntity() | Retrieves full response (body + headers + status) |
| postForObject() | Sends POST and returns response body |
| postForEntity() | Sends POST and returns full response |
| put() | Sends PUT request (no return) |
| delete() | Sends DELETE request |
| exchange() | Custom HTTP method with headers/body etc. |

## ⚠️ Note:

* RestTemplate is **synchronous** and **blocking**.
* For new projects, Spring recommends using **WebClient** (asynchronous, reactive) from **Spring WebFlux**, but RestTemplate is still widely used and stable.

## 🧠 Summary:

| **Feature** | **Description** |
| --- | --- |
| RestTemplate | Used to consume REST APIs from Java |
| HTTP methods | Supports GET, POST, PUT, DELETE, etc. |
| Usage | Call external/internal services over HTTP |
| Alternative | Use WebClient for non-blocking I/O |

Let me know if you want to call an API from your Spring Boot app using RestTemplate with headers or custom body.

Absolutely! Here's a **comparison table** between **RestTemplate (synchronous)** and **WebClient (asynchronous)** in Spring:

**✅ RestTemplate vs WebClient**

| **Feature / Criteria** | **RestTemplate** | **WebClient** |
| --- | --- | --- |
| 🔄 **Type** | Synchronous (blocking) | Asynchronous (non-blocking) |
| 🚀 **Introduced In** | Spring 3 | Spring 5 (with WebFlux) |
| 🔁 **Reactive Programming** | ❌ Not supported | ✅ Fully supported |
| ⚙️ **Thread Usage** | Blocks thread until response returns | Does **not block** threads |
| 🔧 **Request Execution** | Simple, blocking HTTP calls | Reactive, event-driven HTTP calls |
| 📈 **Performance Under Load** | Slower for concurrent calls | Fast and efficient for many requests |
| ✅ **Ease of Use** | Easier for beginners | Slightly complex (uses Mono, Flux) |
| 📦 **Part of** | spring-web | spring-webflux |
| 📤 **Streaming Support** | ❌ Limited | ✅ Full streaming & backpressure |
| ⚠️ **Status** | Deprecated (not officially, but discouraged) | ✅ Recommended for new apps |
| 🧪 **Testing Support** | JUnit / MockRestServiceServer | WebTestClient / StepVerifier |
| 🌐 **Used in** | Traditional MVC-based apps | Reactive + Microservices apps |

**🧠 Summary:**

* ✅ **Use RestTemplate**: For simple, blocking HTTP calls in small apps or legacy systems.
* 🚀 **Use WebClient**: For **modern**, **scalable**, **non-blocking** apps (especially microservices or reactive apps).