### 🔁 Quick Recap of Core Stream Concepts

**✅ What is a Stream?**

* A **Stream** in Java represents a sequence of elements supporting **sequential** and **parallel aggregate operations**.
* **Does not store data**, it carries values from a source (like a Collection) through a pipeline of operations.

**✅ Types of Operations:**

* **Intermediate operations**: return a stream (e.g., filter(), map(), sorted())
* **Terminal operations**: produce a result or a side-effect (e.g., collect(), forEach(), reduce())

**✅ Characteristics:**

* **Laziness**: intermediate operations are lazy until a terminal op is invoked.
* **No mutation**: encourages functional-style programming.
* **Can be parallelized** with .parallelStream().

#### 🔷 1. Stream Creation

**✅ From Collections:**

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

**✅ From Arrays:**

String[] array = {"a", "b", "c"}; Stream<String> stream = Arrays.stream(array);

**✅ Infinite Streams:**

Stream<Integer> infinite = Stream.iterate(0, n -> n + 1).limit(10);

#### 🔷 2. Intermediate Operations

**✅**filter(Predicate)

Filters elements based on a condition.

list.stream().filter(s -> s.startsWith("a"))

#### **✅**map(Function)

Transforms each element.

list.stream().map(String::toUpperCase)

#### **✅**flatMap(Function)

Useful when each element is mapped to a stream/collection.

List<String> words = Arrays.asList("hello", "world"); List<String> chars = words.stream() .flatMap(word -> Arrays.stream(word.split(""))) .collect(Collectors.toList()); // [h, e, l, l, o, w, o, r, l, d]

#### **✅**distinct()**,**sorted()**,**limit()**,**skip()

list.stream().distinct().sorted().limit(5).skip(2)

#### 🔷 3. Terminal Operations

**✅**collect(...)

* Most powerful terminal op.
* Collect results into collections or summary statistics.

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

**Advanced**collect()**–**groupingBy**:**

Map<String, List<Employee>> groupedByDept = employees.stream().collect(Collectors.groupingBy(Employee::getDepartment));

#### **✅**forEach(...)

list.stream().forEach(System.out::println);

⚠️ Avoid mutating shared state in parallel streams with forEach.

#### **✅**reduce(...)

Used for accumulating a result (e.g., sum, product, string concat).

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

#### **✅**anyMatch**,**allMatch**,**noneMatch

boolean allPassed = students.stream().allMatch(s -> s.getMarks() > 35);

#### 🔷 4. Advanced Topics

**✅**groupingBy**+**mapping

Map<String, List<String>> deptToNames = employees.stream().collect(Collectors.groupingBy( Employee::getDepartment, Collectors.mapping(Employee::getName, Collectors.toList()) ));

#### **✅**partitioningBy

Splits stream into two groups (boolean condition).

Map<Boolean, List<Employee>> partitioned = employees.stream().collect(Collectors.partitioningBy(e -> e.getSalary() > 50000));

#### ✅ Parallel Streams

list.parallelStream()

**Caution**:

* Don’t use with shared mutable state.
* Good for CPU-bound operations with large data.
* Use ForkJoinPool.commonPool() under the hood.

#### ✅ Custom Collector (For custom data structure)

Collector<Employee, ?, Set<String>> toNameSet = Collectors.mapping(Employee::getName, Collectors.toSet());

**✅ Stream vs Collection**

| **Feature** | **Stream** | **Collection** |
| --- | --- | --- |
| Stores data? | ❌ No | ✅ Yes |
| Can be reused? | ❌ No | ✅ Yes |
| Lazy? | ✅ Yes | ❌ No |
| Parallelizable? | ✅ Yes (e.g., parallelStream()) | ❌ Not directly |

| **Method** | **Type** | **Description** |
| --- | --- | --- |
| filter() | Intermediate | Filters elements |
| map() | Intermediate | Transforms elements |
| sorted() | Intermediate | Sorts elements |
| skip(n) | Intermediate | Skips the first n elements |
| limit(n) | Intermediate | Limits the stream to n elements |
| flatMap() | Intermediate | Flattens nested structures |
| collect() | Terminal | Collects the result |
| findFirst() | Terminal | Gets the first element |
| min()/max() | Terminal | Gets min/max |
| forEach() | Terminal | Consumes each element |
| anyMatch() | Terminal | Returns true if any element matches |

## 🔹 map(): Transforming Data

**✅ What it does:**

* Transforms each element of the stream using the provided **function**.
* Returns a **Stream of transformed elements**.

**🔧 Syntax:**

<T, R> Stream<R> map(Function<? super T, ? extends R> mapper)

**🧪 Simple Example:**

List<String> names = Arrays.asList("john", "emma", "raj"); List<String> upper = names.stream() .map(String::toUpperCase) .collect(Collectors.toList()); // [JOHN, EMMA, RAJ]

**🛠️ Real-World Scenario (Mid-Level):**

You have a list of Employee objects. Return a list of employee names in uppercase.

class Employee { private String name; private int age; // constructor, getters } List<Employee> employees = Arrays.asList( new Employee("John", 30), new Employee("Emma", 28) ); List<String> upperNames = employees.stream() .map(e -> e.getName().toUpperCase()) .collect(Collectors.toList());

**🧠 Advanced Scenario:**

From a list of Order objects, extract a list of item names in all orders (without flattening).

class Order { private List<String> items; // getter } List<Order> orders = ...; List<List<String>> itemLists = orders.stream() .map(Order::getItems) // This gives you List<List<String>> .collect(Collectors.toList());

📌 Note: This still gives nested lists. Use flatMap() (next topic) to flatten.

**❓Questions for**map()

**Q1 (Easy):**

Given a List<Integer>, use map() to return a list of square of each number.

List<Integer> squares = list.stream() .map(num -> num \* num) .collect(Collectors.toList());

**Q2 (Moderate):**

You have a List<Employee> with fields name and salary. Use map() to:

* Return a list of names where salary is incremented by 10%, but original list remains unchanged.

List<Employee> updatedList = emp.stream() .map(e -> new Employee(e.getName(), e.getSalary() \* 1.10)) .collect(Collectors.toList());

**🧩 Scenario: Transforming Order → Summary DTO**

You have a list of Order objects like:

class Order { private String orderId; private double totalAmount; private Customer customer; // constructor, getters } class Customer { private String name; private String email; // constructor, getters }

You want to convert the list of orders to a list of lightweight DTOs for a report:

class OrderSummary { private String orderId; private String customerEmail; private String summary; public OrderSummary(String orderId, String customerEmail, String summary) { this.orderId = orderId; this.customerEmail = customerEmail; this.summary = summary; } @Override public String toString() { return "Order " + orderId + " for " + customerEmail + ": " + summary; } }

**🧠 Your Task:**

Using map(), return a List<OrderSummary> from a given List<Order>, where:

* summary should be like: "₹450.0 placed by Alice"
* i.e., "₹" + order.totalAmount + " placed by " + customer.name

List<OrderSummary> summaries = orders.stream() .map(or -> new OrderSummary( or.getOrderId(), or.getCustomer().getEmail(), "₹" + or.getTotalAmount() + " placed by " + or.getCustomer().getName() )) .collect(Collectors.toList());

## 🔁 function.identity

you **can** use Function.identity() — but **only if you're not modifying the object inside**map().

Let me explain both:

**✅ What is**Function.identity()**?**

Function<T, T> identity = t -> t;

It's a built-in **no-op function** — returns the same object without any change.

Used when:

* You need to "map" but don’t want to transform
* Or you're chaining intermediate ops but don’t want to touch the objects

**🔹 Example Use (Read-only passthrough):**

List<Employee> result = employees.stream() .filter(emp -> emp.getSalary() > 50000) .map(Function.identity()) // <== no transformation .collect(Collectors.toList());

Equivalent to:

.map(e -> e)

**🔴 Important: You cannot use**Function.identity()**if you need to mutate the object in**map()**.**

This won’t work:

employees.stream() .map(emp -> { emp.setSalary(emp.getSalary() \* 1.10); return Function.identity().apply(emp); // still manual })

In such cases, **you still have to perform the mutation explicitly**, then return the object.

**✅ Cleaner Mutation Example:**

If you want concise in-place mutation **and return the same object**, just do:

.map(emp -> { emp.setSalary(emp.getSalary() \* 1.10); return emp; })

Or, for readability:

.map(applyHike())

Where:

private static Function<Employee, Employee> applyHike() { return emp -> { emp.setSalary(emp.getSalary() \* 1.10); return emp; }; }

**🧠 Summary:**

| **Use Case** | **Should You Use**Function.identity()**?** |
| --- | --- |
| Just passing values through | ✅ Yes |
| Want to mutate objects | ❌ No – Use manual mutation + return |
| Collecting without transform | ✅ Yes |

## 🔹 flatMap(): Flatten + Map

**✅ What it does:**

* Takes a function that **maps an element to a Stream**
* Then **flattens** all resulting streams into **one** combined stream.

<T, R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)

**🔁**map()**vs**flatMap()**:**

| **Method** | **Output Type** | **Flattens Structure?** |
| --- | --- | --- |
| map() | Stream<Stream<T>>, Stream<List<T>>, etc. | ❌ No |
| flatMap() | Stream<T> | ✅ Yes |

**🧪 Simple Example:**

List<List<String>> data = Arrays.asList( Arrays.asList("a", "b"), Arrays.asList("c", "d") ); List<String> flat = data.stream() .flatMap(List::stream) .collect(Collectors.toList()); // Output: ["a", "b", "c", "d"]

**🧠 Real-World Scenario #1: Extract Emails from Multiple Departments**

**Classes:**

class Department { private String name; private List<Employee> employees; // getters } class Employee { private String name; private String email; // getters }

**🔹 Goal:**

Return a List<String> of **all employee emails** from all departments.

List<String> allEmails = departments.stream() .flatMap(dept -> dept.getEmployees().stream()) // Stream<Employee> .map(emp -> emp.getEmail()) // Stream<String> .collect(Collectors.toList()); // List<String>

**🔍 Step-by-step:**

1. .stream() → turns List<Department> into Stream<Department>
2. .flatMap(dept -> dept.getEmployees().stream())  
   → flattens each department’s employee list into one big Stream<Employee>
3. .map(emp -> emp.getEmail())  
   → extracts the email from each employee
4. .collect(Collectors.toList())  
   → collects all emails into a single list

**🧠 Want to try this?**

Here’s a variation:

You have List<Order> where each Order has a list of Item. Each Item has name and price. Return a flat List<String> of all item names across all orders.

Would you like to attempt this flatMap() challenge?

**You said:**

List<String> itemNames = orders.stream() .flatMap(order -> order.getItem().stream()) .map(Item::getName) .collect(Collectors.toList());

**🧠 Next-Level Challenge?**

Suppose each Item has a list of Tag objects, and you want a **distinct list of all tag names** across all orders.

Would you like to try that as your next flatMap() puzzle? Or shall we move on to filter()?

**You said:**

**🧩 Scenario: Items → Tags → Tag Names**

You have:

Order**:**

class Order { private List<Item> items; public List<Item> getItem() { return items; } }

Item**:**

class Item { private List<Tag> tags; public List<Tag> getTags() { return tags; } private String name; // getters }

Tag**:**

class Tag { private String name; public String getName() { return name; } }

**🎯 Goal:**

Return a List<String>**of all unique tag names** across all items in all orders.

✅ Use flatMap()  
✅ Use distinct() to avoid duplicates  
✅ Use method references where suitable

List<String> uniqueTagNames = orders.stream() .flatMap(o -> o.getItem().stream()) // Stream<Item> .flatMap(i -> i.getTags().stream()) // Stream<Tag> .map(Tag::getName) // Stream<String> .distinct() // remove duplicates .collect(Collectors.toList()); // List<String>

**🧩 Challenge: Split and Flatten CSV Strings**

You are given a List<String> like this:

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

**🎯 Goal:**

Use Streams and flatMap() to return a List<String> containing **all individual fruits**, without duplicates.

List<String> result = csvData.stream() .flatMap(str -> Arrays.stream(str.split(","))) // split each string and stream the array .distinct() // remove duplicates .collect(Collectors.toList()); // collect into List

**✅ Output:**

["apple", "banana", "orange", "grape"]

**🧩 Challenge: Books and Authors**

You have a List<Book> where each Book has:

class Book { private String title; private List<Author> authors; public List<Author> getAuthors() { return authors; } } class Author { private String name; public String getName() { return name; } }

**🎯 Goal:**

Return a Set<String> of all **unique author names** from the list of books, using flatMap().

Set<String> authorNames = books.stream() .flatMap(b -> b.getAuthor().stream()) .map(Author::getName) .collect(Collectors.toSet());

**🧩 Challenge: Users and Notifications (Real-world event flattening)**

You have a list of User objects:

class User { private String username; private List<Notification> notifications; public List<Notification> getNotifications() { return notifications; } } class Notification { private String type; // e.g. "EMAIL", "SMS", "PUSH" private String message; public String getType() { return type; } public String getMessage() { return message; } }

**🎯 Goal:**

Return a List<String> of **all SMS notification messages** from all users, using flatMap().

List<String> smsMessages = users.stream() .flatMap(u -> u.getNotifications().stream()) // Flatten all user notifications .filter(n -> n.getType().equalsIgnoreCase("SMS")) // Keep only SMS .map(Notification::getMessage) // Extract message .collect(Collectors.toList()); // Collect into List<String>

## 🔁 Difference between map() and flatMap()

**✅**map()**:**

* **Transforms** each element of a stream (or collection) using a function.
* The result is a **one-to-one** mapping.
* If the function returns a collection/stream, you get a **nested structure** (e.g., Stream<Stream<T>> or List<List<T>>).

**Example:**

List<String> words = Arrays.asList("apple", "banana"); List<List<String>> mapped = words.stream() .map(word -> Arrays.asList(word.split(""))) .collect(Collectors.toList()); System.out.println(mapped); // [[a, p, p, l, e], [b, a, n, a, n, a]]

**✅**flatMap()**:**

* Also transforms each element, but **flattens** the result into a **single stream**.
* Useful when the mapping function returns a stream or collection.
* Combines all inner collections into one.

**Example:**

List<String> words = Arrays.asList("apple", "banana"); List<String> flatMapped = words.stream() .flatMap(word -> Arrays.stream(word.split(""))) .collect(Collectors.toList()); System.out.println(flatMapped); // [a, p, p, l, e, b, a, n, a, n, a]

**🔁 Summary Table:**

| **Feature** | map() | flatMap() |
| --- | --- | --- |
| Transformation | One-to-one | One-to-many (with flattening) |
| Returns | Stream of objects | Flattened Stream of objects |
| Output Type Example | Stream<Stream<T>> | Stream<T> |
| Use Case | Simple transformation | Transformation + flattening nested structure |

**🧠 When to use which?**

* Use \*\*map()\*\* when the function returns a **single value** for each input.
* Use \*\*flatMap()\*\* when the function returns a **collection or stream** for each input and you want a **single flattened stream** as the output.

## 🔹 filter() in Java Streams

**✅ What it does:**

Filters out elements that don’t match a given **predicate** (a condition that returns true or false).

Stream<T> filter(Predicate<? super T> predicate)

**🔁 Simple Example:**

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5); List<Integer> evens = numbers.stream() .filter(n -> n % 2 == 0) .collect(Collectors.toList()); // [2, 4]

**🚀 Let's Start with Your First**filter()**Challenge**

**🧩 Scenario: Active Users**

You have a list of User objects:

class User { private String username; private boolean isActive; public boolean isActive() { return isActive; } public String getUsername() { return username; } }

**🎯 Goal:**

Return a List<String> of usernames where the user is **active**.

List<String> activeUsernames = users.stream() .filter(User::isActive) .map(User::getUsername) .collect(Collectors.toList());

**🧠 Ready for a Trickier One?**

You have a list of Transaction objects.  
Each transaction has:

class Transaction { private double amount; private String status; // e.g. "SUCCESS", "PENDING", "FAILED" public double getAmount() { return amount; } public String getStatus() { return status; } }

**🎯 Goal:**

Return a List<Double> of **amounts** for all **successful transactions > ₹1000**.

List<Double> successfulHighAmounts = transactions.stream() .filter(t -> t.getAmount() > 1000 && "SUCCESS".equalsIgnoreCase(t.getStatus())) .map(Transaction::getAmount) .collect(Collectors.toList());

**🧩 Challenge: Recent Orders with Valid Items**

You have a list of Order objects:

class Order { private LocalDate orderDate; private List<Item> items; public LocalDate getOrderDate() { return orderDate; } public List<Item> getItems() { return items; } } class Item { private String name; private boolean inStock; public boolean isInStock() { return inStock; } }

**🎯 Goal:**

Return a List<Order> where:

* The order is placed **in the last 30 days**
* It contains **at least one item that is in stock**

📅 You can get today’s date using:

LocalDate today = LocalDate.now();

LocalDate thirtyDaysAgo = LocalDate.now().minusDays(30); List<Order> recentOrdersWithStock = orders.stream() .filter(o -> o.getOrderDate().isAfter(thirtyDaysAgo)) // within 30 days .filter(o -> o.getItems().stream().anyMatch(Item::isInStock)) // has at least one in-stock item .collect(Collectors.toList());

**🧩 Challenge: Customers with High-Value Credit Cards**

You have a list of Customer objects:

class Customer { private String name; private List<CreditCard> creditCards; public List<CreditCard> getCreditCards() { return creditCards; } } class CreditCard { private String type; // e.g. "VISA", "MASTERCARD", "AMEX" private double creditLimit; public String getType() { return type; } public double getCreditLimit() { return creditLimit; } }

**🎯 Goal:**

Return a List<Customer> who have **at least one credit card** of type "VISA" **with credit limit > ₹1,00,000**

List<Customer> result = customers.stream() .filter(c -> c.getCreditCards().stream() .anyMatch(cc -> "VISA".equalsIgnoreCase(cc.getType()) && cc.getCreditLimit() > 100000)) .collect(Collectors.toList());

**🧩 Challenge: Students with Recent Passed Exams**

You have a list of Student objects:

class Student { private String name; private List<Exam> exams; public List<Exam> getExams() { return exams; } } class Exam { private String subject; private int score; private LocalDate examDate; public String getSubject() { return subject; } public int getScore() { return score; } public LocalDate getExamDate() { return examDate; } }

**🎯 Goal:**

Return a List<Student> who have **passed at least one exam** (score ≥ 50) in the **last 60 days**.

LocalDate sixtyDaysAgo = LocalDate.now().minusDays(60); List<Student> studentList = students.stream().filter(s->s.getExam().stream().anyMatch(e->e.getScore()>=50 && e.getExamDate().isAfter(sixtyDaysAgo))).collect(Collectors.toList());

## 🔁 **when to use**filter(...stream())**vs**flatMap()?

**🔍**filter(...stream())**vs**flatMap(...)

**✅ 1. Use**filter(...stream())

When you're checking if an *inner collection* **contains at least one element** meeting a condition, **but you still want to operate on the outer object**.

**📌 Example:**

// Get Customers who have at least one VISA card customers.stream() .filter(c -> c.getCreditCards().stream() .anyMatch(card -> "VISA".equals(card.getType()))) .collect(Collectors.toList());

* ✅ You **filter the outer object** (Customer)
* ✅ Based on a condition **inside** their List<CreditCard>

**✅ 2. Use**flatMap(...)

When you want to **flatten a nested stream** — i.e., operate on the **inner elements directly**, not the outer object.

**📌 Example:**

// Get a flat list of all credit card numbers (regardless of customer) customers.stream() .flatMap(c -> c.getCreditCards().stream()) // flatten .map(CreditCard::getNumber) .collect(Collectors.toList());

* ✅ You want to “extract and operate” on the **nested collection**
* 🔄 One List<Customer> → Stream of List<CreditCard> → flattened to Stream<CreditCard>

**🎯 Summary Table:**

| **Goal** | **Use** |
| --- | --- |
| Keep the outer object, filter based on inner list | filter(...) + .stream() inside |
| Work with inner list elements as flat stream | flatMap(...) |

**🤔 Quick Diagnostic:**

* **Do you want the**Customer**?** → Use filter(...)
* **Do you want the**CreditCard**?** → Use flatMap(...)

**🧩 Challenge Recap:**

**🔁 Sort Orders by Latest Item Delivery Date (Descending)**

**Given:**

class Order { String orderId; List<Item> items; public List<Item> getItems() { return items; } } class Item { String name; LocalDate deliveryDate; public LocalDate getDeliveryDate() { return deliveryDate; } }

**🎯 Task:**

Sort List<Order> such that orders with **most recently delivered items** appear **first**.

List<Order> sortedOrders = orders.stream() .sorted(Comparator.comparing( o -> o.getItems().stream() .map(Item::getDeliveryDate) .max(Comparator.naturalOrder()) .orElse(LocalDate.MIN), // fallback if needed Comparator.reverseOrder() )) .collect(Collectors.toList());

**🧩 Challenge 4: Sort Courses by Enrolled Student Count, then by Name**

**🎓 Given:**

You have a list of Course objects:

class Course { String name; List<Student> enrolledStudents; public String getName() { return name; } public List<Student> getEnrolledStudents() { return enrolledStudents; } }

**🎯 Task:**

Sort the list of Course objects by:

1. **Number of enrolled students** — descending
2. If equal, then **course name** — ascending

**📥 Output:**

Return a List<Course> sorted accordingly.

**Option 1: Use**Comparator.comparingInt**for student count**

List<Course> sortedCourses = courses.stream() .sorted( Comparator.comparingInt((Course c) -> c.getEnrolledStudents().size()) .reversed() .thenComparing(Course::getName) ) .collect(Collectors.toList());

**Option 2: If you really want to use a custom comparator lambda**

List<Course> sortedCourses = courses.stream() .sorted((c1, c2) -> { int cmp = Integer.compare(c2.getEnrolledStudents().size(), c1.getEnrolledStudents().size()); if (cmp == 0) { return c1.getName().compareTo(c2.getName()); } return cmp; }) .collect(Collectors.toList());

**✅ 🎯 1. Know What Comparator Wants**

| **Goal** | **Use This** |
| --- | --- |
| Compare by int (like size, quantity) | Comparator.comparingInt(obj -> value) |
| Compare by double | Comparator.comparingDouble(obj -> value) |
| Compare by Comparable (String, LocalDate, etc) | Comparator.comparing(obj -> value) |

**🔁 Add**.reversed()**if you want descending order**

Comparator.comparing(Employee::getSalary).reversed()

**➕ Chain further with**.thenComparing(...)

Comparator.comparing(Employee::getSalary).reversed() .thenComparing(Employee::getAge) .thenComparing(Employee::getName)

**🧠 🎯 2. Memory Trick: The 3-Step Formula**

Whenever you sort:

.stream()

.sorted(

Comparator.comparing(...) ← 1st condition

.thenComparing(...) ← 2nd condition

.thenComparing(...) ← 3rd condition (optional)

)

Just remember:

compare by → then compare by → then compare by

Use .reversed() **right after comparing(...)** if you want descending.

**✅ Version 1: More verbose**

Comparator.comparing(Employee::getSalary, Comparator.reverseOrder())

**✅ Version 2: Cleaner and preferred**

Comparator.comparing(Employee::getSalary).reversed()

Both achieve the same result — they **sort by salary in descending order**, and then by age and name ascending.

**🧠 Just remember:**

* .reversed() reverses the entire comparator built so far.
* It works only if .comparing(...) returns a Comparator<T> (which it does).

List<Employee> sortedEmployees = employees.stream() .sorted( Comparator.comparing(Employee::getSalary).reversed() .thenComparing(Employee::getAge) .thenComparing(Employee::getName) ) .collect(Collectors.toList());

## 🔹 noneMatch(), anyMatch(), AllMatch()

**🔍 1.**anyMatch(Predicate<T> predicate)

**✅ Definition:**

Returns true **if any element** of the stream **matches the given predicate**.

**📌 Use case:**

"Is there *at least one* even number in the list?"

**✅ Example:**

List<Integer> nums = Arrays.asList(1, 3, 5, 6); boolean hasEven = nums.stream().anyMatch(n -> n % 2 == 0); System.out.println(hasEven); // Output: true

**🔍 2.**allMatch(Predicate<T> predicate)

**✅ Definition:**

Returns true **only if all elements** of the stream **match the given predicate**.

**📌 Use case:**

"Are *all* numbers positive?"

**✅ Example:**

List<Integer> nums = Arrays.asList(10, 20, 30); boolean allPositive = nums.stream().allMatch(n -> n > 0); System.out.println(allPositive); // Output: true

**🔍 3.**noneMatch(Predicate<T> predicate)

**✅ Definition:**

Returns true **if no elements** of the stream **match the given predicate**.

**📌 Use case:**

"Are *no* numbers negative?"

**✅ Example:**

List<Integer> nums = Arrays.asList(2, 4, 6); boolean noNegatives = nums.stream().noneMatch(n -> n < 0); System.out.println(noNegatives); // Output: true

**🧠 Summary Table:**

| **Method** | **Returns**true**if...** |
| --- | --- |
| anyMatch() | **At least one** element matches |
| allMatch() | **All** elements match |
| noneMatch() | **No** elements match |

**🧩 Challenge: Product Review Quality Check**

You have a list of Product objects:

class Product { private String name; private List<Review> reviews; public List<Review> getReviews() { return reviews; } } class Review { private int rating; // 1 to 5 private String comment; public int getRating() { return rating; } public String getComment() { return comment; } }

**✅ 1. anyMatch**

Find if **any product** has **at least one review** with a **5-star rating**.

**Expected Result:** boolean — true or false

**✅ 2. allMatch**

Check if **all products** have **only 4-star or above** ratings in **all their reviews**.

**Expected Result:** boolean

**✅ 3. noneMatch**

Confirm that **no product** has **any 1-star review**.

**Expected Result:** boolean

You can assume List<Product> products is available.  
🔥 Give it a go! Let’s see your answer for all 3.

**1. 🔍 Check if any product has at least one 5-star review:**

boolean anyFiveStar = products.stream() .anyMatch(pro -> pro.getReviews().stream() .anyMatch(rev -> rev.getRating() == 5));

**2. 🔍 Check if all products have only ratings ≥ 4:**

boolean allHighRated = products.stream() .allMatch(pro -> pro.getReviews().stream() .allMatch(rev -> rev.getRating() >= 4));

**3. 🔍 Check that no product has a 1-star review:**

boolean noOneStar = products.stream() .noneMatch(pro -> pro.getReviews().stream() .anyMatch(rev -> rev.getRating() == 1));

**🧠 Tip:**

* Use filter() when you want to **extract elements**.
* Use anyMatch(), allMatch(), noneMatch() when you want to **evaluate conditions** on the entire stream.

**🧩 Challenge: Find Suspicious Customers**

You have the following data model:

class Customer { private String name; private List<Order> orders; public List<Order> getOrders() { return orders; } } class Order { private double amount; private boolean flagged; public double getAmount() { return amount; } public boolean isFlagged() { return flagged; } }

**🎯 Objective:**

✅ Check if there is **any customer** who has **at least one flagged order** with an **amount over ₹10,000**.

**✅ Expected Output:**

A single boolean — true if **such a flagged high-value order exists**, false otherwise.

boolean hasSuspicious = customers.stream() .anyMatch(c -> c.getOrders().stream() .anyMatch(o -> o.getAmount() > 10000 && o.isFlagged()));

**🧩 New Challenge: Get Names of Suspicious Customers**

**🎯 Goal:**

Return a List<String> containing the names of all customers who have **at least one flagged order with amount > ₹10,000**.

**✅ Solution:**

List<String> suspiciousCustomerNames = customers.stream() .filter(c -> c.getOrders().stream() .anyMatch(o -> o.getAmount() > 10000 && o.isFlagged())) .map(Customer::getName) .collect(Collectors.toList());

**🧾 Example Output:**

["Alice", "Rajesh", "Fatima"]

List<String> suspiciousCustomerNames = customers.stream() .filter(c -> c.getOrders().stream() .anyMatch(o -> o.getAmount() > 10000 && o.isFlagged())) .map(Customer::getName) .collect(Collectors.toList());

**🧩 Challenge: Get All Flagged High-Value Order IDs**

**💼 Scenario:**

From a list of Customer objects, extract **all order IDs** of orders that are:

* **flagged** ✅
* **amount > ₹10,000** ✅

And return them as a List<String>.

**🧾 Assumed Model (with IDs):**

class Customer { private String name; private List<Order> orders; public List<Order> getOrders() { return orders; } } class Order { private String id; private double amount; private boolean flagged; public String getId() { return id; } public double getAmount() { return amount; } public boolean isFlagged() { return flagged; } }

**✅ Goal:**

List<String> flaggedOrderIds = ...

* Should include all order IDs matching:
  + isFlagged() == true
  + getAmount() > 10000

**🔧 Solution Using**flatMap**,**filter**,**map**:**

List<String> flaggedOrderIds = customers.stream() .flatMap(c -> c.getOrders().stream()) // Stream<Order> .filter(o -> o.isFlagged() && o.getAmount() > 10000) .map(Order::getId) .collect(Collectors.toList());

**✅ Example Output:**

["ORD123", "ORD879", "ORD543"]

**🧩 Challenge: Group Flagged High-Value Orders by Customer Name**

**🎯 Goal:**

From a list of Customer objects, return a Map<String, List<String>>, where:

* Key = Customer name (String)
* Value = List of flagged, high-value order IDs (amount > ₹10,000)

**🧾 Output:**

{ "Alice": ["ORD101", "ORD109"], "Raj": ["ORD202"] }

**🧾 Data Model Recap:**

class Customer { private String name; private List<Order> orders; public String getName() { return name; } public List<Order> getOrders() { return orders; } } class Order { private String id; private double amount; private boolean flagged; public String getId() { return id; } public double getAmount() { return amount; } public boolean isFlagged() { return flagged; } }

**✅ Final Solution:**

Map<String, List<String>> flaggedOrdersByCustomer = customers.stream() .collect(Collectors.toMap( Customer::getName, customer -> customer.getOrders().stream() .filter(order -> order.isFlagged() && order.getAmount() > 10000) .map(Order::getId) .collect(Collectors.toList()) ));

**🧠 Notes:**

* Collectors.toMap() → great for creating Map<K, V> from a stream
* For each customer:
  + We stream their orders,
  + Filter the suspicious ones,
  + Map to their IDs,
  + Collect into a list.

**🚨 Optional: If a customer has no matching orders, their list will be empty.**

You can also filter them out like this:

Map<String, List<String>> flaggedOrdersByCustomer = customers.stream() .filter(c -> c.getOrders().stream() .anyMatch(o -> o.isFlagged() && o.getAmount() > 10000)) .collect(Collectors.toMap( Customer::getName, c -> c.getOrders().stream() .filter(o -> o.isFlagged() && o.getAmount() > 10000) .map(Order::getId) .collect(Collectors.toList()) ));

## 🔹 sorted()

**🔍**sorted()**in Java Streams**

**📘 Definition:**

Stream.sorted() returns a stream consisting of the elements **sorted in natural order** or according to a provided **Comparator**.

**🔧 Two Main Variants:**

1. **Natural Order**:

stream.sorted();

* Works only if elements implement Comparable (like String, Integer).

1. **Custom Comparator**:

stream.sorted(Comparator.comparing(...));

* Use this for sorting objects by specific fields.

**✅ Simple Examples**

**1️⃣ Sort integers in ascending order:**

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

**2️⃣ Sort strings alphabetically:**

List<String> namesSorted = names.stream() .sorted() .collect(Collectors.toList());

**3️⃣ Sort employees by salary (ascending):**

List<Employee> sorted = employees.stream() .sorted(Comparator.comparing(Employee::getSalary)) .collect(Collectors.toList());

**4️⃣ Sort employees by salary descending:**

.sorted(Comparator.comparing(Employee::getSalary).reversed())

**🧩 Let's Begin: Challenge 1**

You have a list of Product objects with fields:

class Product { String name; double price; int rating; }

**🎯 Task:**

Sort the list of products by:

* **price (ascending)**
* then by **rating (descending)** if prices are equal

List<Product> sortedProducts = products.stream() .sorted( Comparator.comparing(Product::getPrice) .thenComparing(Comparator.comparing(Product::getRating).reversed()) ) .collect(Collectors.toList());

**🧩 Challenge 2: Multi-Field Employee Sorting**

**👔 Given:**

You have a list of Employee objects:

class Employee { String name; double salary; int age; }

**🎯 Task:**

Sort the employees by:

1. **Salary (descending)**
2. Then by **Age (ascending)**
3. Then by **Name (alphabetically)**

Return a List<Employee> sorted in that order using Streams.

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

**🧪 Optional: Custom Comparator via Lambda**

If you wanted to use a lambda for name comparison:

.thenComparing((e1, e2) -> e1.getName().compareTo(e2.getName()))

**🧩 Challenge 3: Sort Orders by Latest Item Delivery Date**

**📦 Given:**

You have a list of Order objects.  
Each Order contains a list of Item objects, and each Item has a **delivery date**.

class Order { String orderId; List<Item> items; public List<Item> getItems() { return items; } public String getOrderId() { return orderId; } } class Item { String name; LocalDate deliveryDate; public LocalDate getDeliveryDate() { return deliveryDate; } }

**🎯 Task:**

Sort the orders **by the latest delivery date among their items** — in **descending** order (latest first).

So:

* Order with the item delivered most recently → comes first
* Orders with older deliveries → appear later

**🧾 Output:**

A List<Order> sorted by each order’s **most recent item delivery date**.

## A key difference in how Java's comparator chaining behaves based on **primitives vs. objects**.

**🔍 The Core Difference:**

Comparator.comparing(Employee::getSalary) // Uses Comparator<T>

Comparator.comparingDouble(Employee::getSalary) // Uses primitive-specific comparator

**✅ This works fine:**

Comparator.comparing(Employee::getSalary).reversed()

.thenComparing(Employee::getAge)

.thenComparing(Employee::getName)

Because:

* Comparator.comparing(...) returns a **Comparator<Employee>**
* .reversed() returns a Comparator<Employee> too
* So you can **chain thenComparing(...)** directly after reversed() ✔️

**❌ But this does NOT work:**

Comparator.comparingDouble(Employee::getSalary).reversed()

.thenComparing(Employee::getAge)

Because:

* comparingDouble(...) returns a **primitive comparator**: Comparator<Employee> backed by ToDoubleFunction<Employee>
* .reversed() on a primitive comparator returns a **different internal type**
* You **cannot chain thenComparing() directly** on the result of that .reversed()

**🔧 Solution:**

Wrap the primitive sort into a boxed comparator like:

Comparator.comparing(Employee::getSalary, Comparator.reverseOrder())

Or stick with:

Comparator.comparing(Employee::getSalary)

and reverse **manually**.

**✅ Summary:**

| **Comparator Type** | | | **Allows .reversed().thenComparing()?** |
| --- | --- | --- | --- |
| Comparator.comparing(...) | | | ✅ Yes |
| Comparator.comparingDouble(...) | | | ❌ No — reversed breaks chaining |
| **🧠 Rule of Thumb:**   | **Method Used** | **Chaining Works After .reversed()?** | **Return Type** | | --- | --- | --- | | Comparator.comparing(...) | ✅ Yes | Comparator<T> | | Comparator.comparingDouble(...) | ❌ No — chaining breaks | Comparator<T> (primitive comparator inside) | | Comparator.comparing(..., Comparator.reverseOrder()) | ✅ Yes | Comparator<T> | |  |

## 🔹 Revision Questions 1 🡪

#### 1. Map vs FlatMap

**Problem**:  
You have a list of Customer objects, and each customer has a list of Order objects.  
You need to collect names of customers who have at least one order with:

* amount greater than 10,000
* and the order is flagged as true

customers.stream().filter(cus->cus.getOrders().stream().anyMatch(o->o.getAmount()>10000 && o.isFlagged())).map(Customer::getName).collect(Collectors.toList());

#### 🔹 2. Collectors.groupingBy() Deep Dive

**Problem**:  
Given a list of transactions with the following fields:

* customerId
* amount
* transactionType (e.g. "debit", "credit")

Group all transactions **by transaction type**, and then for each group, calculate the **total transaction amount**.

transactions.stream().collect(Collectors.groupingBy(t->t.getTransactionType(), Collectors.summingDouble(t->t.getAmount())));

#### 🔹 3. Nested Object Search using Streams

**Problem**:  
You have a list of Company objects. Each Company has a list of Departments, and each Department has a list of Employees.  
Find all employee names across all companies whose:

* designation is "Manager"
* and salary is greater than 15 LPA.

comanies.stream().flatMap(c->c.getDepartment().stream()).flatMap(d->d.getEmployee().stream()).filter(e->"Manager".equalsIgnoreCase(e.getDesignation()) && e.getSalary() > 1500000).map(Employee::getName).collect(Collectors.toList())

#### 🔹 4. FlatMap Real-Life Use Case

**Problem**:  
You are given a list of Library objects. Each library contains a list of Book objects.  
Find all distinct book titles authored by “Jane Austen”.

libraries.stream().flatMap(lib->lib.getBook().stream()).filter(b->"Jane Austen".equalsIgnoreCase(b.getAuthor())).map(Book::getTitle).distinct().collect(Collectors.toList());

#### 🔹 5. Filter + Map + Collect Real-World

**Problem**:  
Given a list of students with fields:

* name, rollNo, totalMarks, isPassed

Get a list of roll numbers of students who:

* scored more than 85 marks
* and have passed

students.stream().filter(s->s.getTotalMark() > 85 && s.isPassed()).map(Student::getRollNo).collect(Collectors.toList());

#### 🔹 6. Sorting with Comparator using Streams

**Problem**:  
You have a list of Employee objects with fields: name, age, salary.  
Sort the employees:

* first by salary descending
* then by age ascending

employees.stream()

.sorted(

Comparator.comparing(Employee::getSalary, Comparator.reverseOrder())

.thenComparing(Employee::getAge)

)

.map(Employee::getName)

.collect(Collectors.toList());

#### 🔹 7. Streams with Optional + findFirst/findAny

**Problem**:  
You are given a list of products, each with a list of reviews.  
Find the **first review** with rating less than 3 for the product named “Smartphone X”.

products.stream().filter(p->"Smartphone X".equalsIgnoreCase(p.getProductName())).flatMap(p->p.getReview().stream()).filter(r->r.getRating()<3).map(Review::getComment).findFirst().get();

#### 🔹 8. Partition Employees by Pass/Fail of Appraisal

You are given a list of Employee objects:

class Employee {

String name;

double appraisalScore; // out of 10

}

**✅ Your Task:**

Use Java Streams to **partition** the employees into two groups:

* Those who scored **>= 7.0**
* Those who scored **< 7.0**

Return a Map<Boolean, List<String>>  
→ true → employee names who passed  
→ false → employee names who failed

employees.stream().collect(Collectors.partitioningBy(e->c.getAppraisalScore()>=7.0));

employees.stream()

.collect(Collectors.partitioningBy(

e -> e.getAppraisalScore() >= 7.0,

Collectors.mapping(Employee::getName, Collectors.toList())

));

#### 9. sorted()

##### 🔹 Challenge 1: Sort Integers Ascending

Given a list of integers, sort them in ascending order and return the sorted list.

list.stream().sorted().collect(Collectors.toList());

##### 🔹 Challenge 2: Sort Strings Alphabetically

Given a list of strings (names), sort them alphabetically (case-insensitive).

list.stream().sorted().collect(Collectors.toList());

or

list.stream().sorted((w1, w2) -> w1.compareTo(w2)).collect(Collectors.toList());

or

.sorted(String::compareToIgnoreCase)

##### 🔹 Challenge 3: Sort Employees by Age

Given a list of Employee objects:

class Employee {

String name;

int age;

}

Sort them by age (ascending) and return a list of employee names.

employees.stream()

.sorted(Comparator.comparing(Employee::getAge))

.map(Employee::getName)

.collect(Collectors.toList());

**🔧 Intermediate Level**

##### 🔸 Challenge 4: Sort by Salary Descending

Given a list of Employee with double salary, sort them by salary in **descending order** and return a list of names.

employees.stream()

.sorted(Comparator.comparingDouble(Employee::getSalary).reversed())

.map(Employee::getName)

.collect(Collectors.toList());

##### 🔸 Challenge 5: Sort by Salary Descending, then Age Ascending

Sort employees by:

* Salary (high to low)
* If equal, then by age (youngest first)

Return the sorted names.

employees.stream()

.sorted(Comparator.comparing(Employee::getSalary, Comparator.reverseOrder())

.thenComparing(Employee::getAge))

.map(Employee::getName)

.collect(Collectors.toList());

##### 🔸 Challenge 6: Sort Books by Title Length

Given a list of Book:

class Book {

String title;

String author;

}

Sort books by **title length** in ascending order.

books.stream().sorted(Comparator.comparing(b->b.getTitle().length())).map(Book::getTitle).collect(Collectors.toList())

**🔥 Advanced Level**

##### 🧠 Challenge 7: Sort Students by Percentage with Null-Safe Handling

Sort students by:

* percentage (descending)
* Handle cases where percentage might be null — treat them as lowest

class Student {

String name;

Double percentage;

}

students.stream()

.sorted(

Comparator.comparing(Student::getPercentage, Comparator.nullsLast(Comparator.reverseOrder()))

.thenComparing(Student::getName)

)

.map(Student::getName)

.collect(Collectors.toList());

##### 🧠 Challenge 8: Custom Sort — Last Word in Name

Sort people by the **last word in their full name** (e.g., last name), alphabetically.

"John Ronald Reuel Tolkien" → sort by "Tolkien"

##### 🧠 Challenge 9: Sort Products by Review Count (Nested List)

Given:

class Product {

String name;

List<Review> reviews;

}

Sort products by number of reviews in **descending order**, then by name (ascending).

##### 🧠 Challenge 10: Stable Sort — Maintain Original Order Where Values Equal

Sort a list of tasks by priority:

class Task {

String id;

int priority;

}

If two tasks have the same priority, **preserve the original order** (stable sort).

##### 📚 Bonus Challenge 6B: Sort Books by Title Length with Tie-Breaker

**🔹 Problem Statement:**

You still have your list of Book objects:

class Book {

private String title;

private String author;

// constructor, getters, setters

}

Now, you need to:

1. Sort books by the **length of the title** (ascending)
2. If two books have titles of the same length, sort them **alphabetically** (case-insensitive)
3. Return a List<String> containing only the sorted **titles**

books.stream()

.sorted(

Comparator.comparing(b -> b.getTitle().length()) // 1️⃣ Title length ascending

.thenComparing(Book::getTitle) // 2️⃣ Alphabetical (case-sensitive) tie-breaker

)

.map(Book::getTitle)

.collect(Collectors.toList());

## Revision Questions 2 🡪

### ✅ Topic 1: filter()

#### Q1:

Given a list of integers, return a list containing only even numbers.

integers.stream().filter(in->in%2==0).collect(Collectors.toList());

#### Q2 (filter):

Given a list of Employee objects with fields name, salary, and department, return a list of employees who belong to the "IT" department and have a salary greater than 50,000.

employees.stream().filter(emp->"IT".equalsIgnoreCase(emp.getDepartment()) && emp.getSalary()>50000).collect(Collectors.toList())

#### Q3 (filter):

Given a list of strings, return a list of strings that are **not empty** and have **more than 3 characters**.

strings.stream()

.filter(str -> str != null && !str.isEmpty() && str.length() > 3)

.collect(Collectors.toList());

#### Q4 (filter):

From a list of Student objects with fields name and percentage, filter out the students who have scored **below 40%**.

students.stream().filter(student->student.getPercentage()<40).collect(Collectors.toList())

### ✅ Topic 2: map()

#### Q1 (map):

Given a list of strings, return a list of their lengths.

str.stream().map(String::length).collect(Collectors.toList())

#### Q2 (map):

Given a list of Employee objects, return a list of their names.

emp.stream().map(Employee::getName).collect(Collectors.toList())

#### Q3 (map):

Given a list of integers, return a list where each number is **squared**.

integers.stream().map(num->num\*num).collect(Collectors.toList())

#### Q4 (map):

Given a list of Student objects, return a list of strings in the format:  
"Name: [name], Percentage: [percentage]"

students.stream().map(std->"Name : " + std.getName() + ", Percentage: " + std.getPercentage()).collect(Collectors.toList())

### ✅ Topic 3: flatMap()

#### Q1 (flatMap):

Given a list of lists of integers, flatten it into a single list of integers.

listOfLists.stream()

.flatMap(List::stream)

.collect(Collectors.toList());

#### Q2 (flatMap):

Given a list of Student objects, and each student has a list of subjects, return a list of **all subjects** of all students.

. students.stream().flatMap(std->std.getSubjects().stream()).collect(Collectors.toList())

#### Q3 (flatMap):

Given a list of sentences (strings), return a list of **all words** in those sentences.  
Assume words are separated by spaces.

sentences.stream()

.flatMap(sen -> Arrays.stream(sen.split(" ")))

.collect(Collectors.toList());

### ✅ Topic 4: sorted()

#### Q1 (sorted):

Given a list of integers, sort them in **ascending order**.

integers.stream().sorted().collect(Collectors.toList());

#### Q2 (sorted):

Given a list of Strings, sort them in **reverse alphabetical order**.

strings.stream()

.sorted(Comparator.reverseOrder())

.collect(Collectors.toList());

Alternatively (if using comparing):

strings.stream()

.sorted(Comparator.comparing(String::toString).reversed())

.collect(Collectors.toList());

#### Q3 (sorted):

Given a list of Employee objects, sort them by **salary (descending)**, then by **age (ascending)**, and then by **name (alphabetically)**.

employees.stream().sorted(Comparator.comparing(Employee::getSalary, Comparator.reverseOrder()).thenComparing(Employee::getAge).thenComparing(Employee::getName)).collect(Collectors.toList())

#### Q4 (sorted):

Given a list of Book objects with field title, sort the books first by the **length of the title**, then by the **title alphabetically**.

books.stream().sorted(Comparator.comparing(book->book.getTitle().length()).thenComparing(Book::getTitle)).collect(Collectors.toList())

### ✅ Topic 5: Combination (filter + map + sorted, etc.)

#### Q1 (Combined):

From a list of Student objects, get the names of students who have more than **75%**, sorted by percentage **descending**.

students.stream().filter(std->std.getPercentage()>75).sorted(Comparator.comparing(Student::getPercentage, Comparator.reverseOrder())).map(Student::getName).collect(Collectors.toList());

#### Q2 (Combined):

From a list of Employee objects, return the names of those in the "HR" department with salary > 60,000,  
sorted by name in ascending order.

employees.stream()

.filter(emp -> "HR".equalsIgnoreCase(emp.getDepartment()) && emp.getSalary() > 60000)

.sorted(Comparator.comparing(Employee::getName))

.map(Employee::getName)

.collect(Collectors.toList());

#### Q3 (Combined):

From a list of Order objects, where each order has a list of Product objects,  
return a **sorted list of all product names**, sorted by product **price descending**.

orders.stream().flatMap(ord->ord.getProduct().stream()).sorted(Comparator.comparing(Product::getPrice, Comparator.reverseOrder()).thenComparing(Product::getName)).map(Product::getName).collect(Collectors.toList());

## min, max, findFirst, skip, limit, anyMatch, allMach, nonMatch, dropWhile

**Terminal Operations** in Java Streams:

* min(), max()
* findFirst()
* skip(), limit()
* anyMatch(), allMatch(), noneMatch()
* dropWhile()

## ✅ Topic: min() and max()

min() expects a **Comparator**, not a method reference returning a value.

**Q1 (min):**  
Given a list of Employee objects, find the employee with the **lowest salary**.

employees.stream().min(Comparator.comparing(Employee::getSalary));

This will return an Optional<Employee> containing the employee with the lowest salary.

**Q2 (max):**

From a list of integers, find the **maximum value**.

integers.stream()

.max(Integer::compareTo);

**✅ Option 2 (using Comparator.comparing):**

integers.stream()

.max(Comparator.comparing(Integer::intValue));

Or even simpler:

integers.stream()

.max(Comparator.naturalOrder());

**✅ Topic: findFirst()**

**Q3 (findFirst):**  
From a list of strings, find the **first non-empty string**.

integers.stream()

.max(Comparator.comparingInt(Integer::intValue));

**✅ Q3 (min):**

Given a list of Student objects with fields name and percentage, find the student with the **lowest percentage**.

students.stream().min(Comparator.comparing(Student::getPercentage));

**Q4 (max):**

Given a list of Product objects with fields name and price, find the product with the **highest price**.

products.stream().max(Comparator.comparing(Product::getPrice))

**Q5 (min):**

Given a list of strings, find the string with the **minimum length**.

strings.stream().min(Comparator.comparing(String::length))

**Q6 (max with tie-breaker):**

Given a list of Employee objects, find the employee with the **highest salary**.  
If two employees have the same salary, choose the one with the **lower age**.

employees.stream()

.max(Comparator.comparing(Employee::getSalary)

.thenComparing(Employee::getAge));

**Q7 (min - composite):**

You have a list of Student objects with name, percentage, and age.

Find the student with the **lowest percentage**, and if two students have the same percentage, pick the one who is **older**.

students.stream()

.min(Comparator.comparing(Student::getPercentage)

.thenComparing(Student::getAge, Comparator.reverseOrder()));

## ✅ Topic: findFirst()

This returns an Optional<T> with the **first element** that matches the stream pipeline.

**Q1 (findFirst):**

Given a list of strings, find the **first non-empty string**.

strings.stream()

.filter(str -> str != null && !str.isEmpty())

.findFirst();

This returns an Optional<String> containing the first non-empty string.

**Q2 (findFirst):**

From a list of Student objects sorted by percentage descending, find the **topper** (i.e., the first student in the list).

**✔️ Option 1:**

students.stream()

.max(Comparator.comparing(Student::getPercentage));

* **More efficient** (O(n)) — finds the max directly without sorting.
* Preferred for just finding the topper.

**✔️ Option 2:**

students.stream()

.sorted(Comparator.comparing(Student::getPercentage, Comparator.reverseOrder()))

.findFirst();

* **Less efficient** (due to full sort).
* Useful if you already need the list sorted or plan to reuse it that way.

## ✅ Topic: skip() and limit() – Intermediate Operations

**Q1 (limit):**  
Given a list of integers, return the **first 5 even numbers**.

integers.stream()

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

.limit(5)

.collect(Collectors.toList());

**Q2 (skip):**

Given a list of strings, skip the first 3 and return the rest.

strings.stream().skip(3).collect(Collectors.toList())

**Q3 (skip + limit):**

From a list of Student objects sorted by percentage descending,  
return the **2nd and 3rd toppers' names**.

students.stream().sorted(Comparator.comparing(Student::getPercentage, Comparator.reverseOrder())).skip(1).limit(2).collect(Collectors.toList())

If the question asked only for **names**, you could add a map(Student::getName):

students.stream()

.sorted(Comparator.comparing(Student::getPercentage, Comparator.reverseOrder()))

.skip(1)

.limit(2)

.map(Student::getName)

.collect(Collectors.toList());

## ✅ anyMatch(), allMatch(), noneMatch()?

**✅ Topic: anyMatch(), allMatch(), noneMatch()**

These are **terminal operations** that return a boolean.

**Q1 (anyMatch):**

Given a list of integers, check if **any number is divisible by 7**.

integers.stream().anyMatch(num->num%7==0)

**Q2 (allMatch):**

Given a list of Student objects, check if **all students have more than 35%**.

students.stream().allMatch(std->std.getPercentage()>35)

**Q3 (noneMatch):**

Given a list of Employee objects, check if **no employee belongs to the "Fired" department**.

employees.stream()

.noneMatch(emp -> "Fired".equalsIgnoreCase(emp.getDepartment()));

## ✅ Topic: dropWhile()

Introduced in Java 9, dropWhile() is an **intermediate operation** that skips elements **until** the predicate returns false — after that, it includes **everything**.

**Q1 (dropWhile):**

Given a list of integers sorted in **ascending order**, drop all numbers **less than 50**, and collect the rest.

integers.stream().sorted(Comparator.comparing(Integer::intValue)).dropWhile(int->int<50).collect(Collectors.toList())

**Q2 (dropWhile):**

Given a list of Student objects sorted by percentage ascending,  
drop all students who have scored **less than 40%**, and collect the remaining students.

students.stream()

.dropWhile(std -> std.getPercentage() < 40)

.collect(Collectors.toList());

🧠 **Note:** dropWhile() only works as expected on a **sorted stream** based on the filtering condition. So the input list **must already be sorted by percentage ascending**.

## ✅ Topic: takeWhile()

* takeWhile() is the **opposite of dropWhile()**.
* It keeps taking elements **as long as the condition is true**.
* Once it encounters a false, it **stops entirely** — even if later elements match again.

**Q1 (takeWhile):**

Given a list of integers sorted in **ascending order**, collect all numbers **less than 100** using takeWhile().

## ✅ Revision 3:

#### ✅ Q1

From a list of Employee objects, get the names of the **top 3 earners** in the "Tech" department,  
sorted by salary descending.

employees.stream()

.filter(emp -> "Tech".equalsIgnoreCase(emp.getDepartment()))

.sorted(Comparator.comparing(Employee::getSalary, Comparator.reverseOrder()))

.limit(3)

.map(Employee::getName)

.collect(Collectors.toList());

#### ⚡ Q2:

From a list of Student objects,  
get the names of students who scored more than **60%**,  
**skip the first 2**,  
then sort them by **name**,  
and return a list of their **uppercase names**.

students.stream()

.filter(std -> std.getPercentage() > 60)

.skip(2)

.sorted(Comparator.comparing(Student::getName))

.map(std -> std.getName().toUpperCase())

.collect(Collectors.toList());

#### ⚡ Q3:

From a list of integers,  
get the **first 4 even numbers greater than 50**,  
sorted in **descending order**,

## ✅ Topic: sum() and average()

These are typically used with **primitive streams** (IntStream, DoubleStream, LongStream) via mapToInt(), mapToDouble(), etc.

**✅ Q1. Sum of even numbers (simple integers):**

**Problem:**  
Given a list of integers, calculate the **sum of even numbers**.

int sum = integers.stream()

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

.mapToInt(Integer::intValue)

.sum();

**✅ Q2. Sum of salaries of all employees:**

**Problem:**  
Given a list of Employee objects, calculate the **total salary**.

double totalSalary = employees.stream()

.mapToDouble(Employee::getSalary)

.sum();

**✅ Q3. Average percentage of students:**

**Problem:**  
Given a list of Student objects, calculate the **average percentage**.

OptionalDouble avg = students.stream()

.mapToDouble(Student::getPercentage)

.average();

Use avg.isPresent() or avg.orElse(0.0) to safely access the value.

**✅ Q4. Sum of prices of products having price > 100:**

double sum = products.stream()

.filter(p -> p.getPrice() > 100)

.mapToDouble(Product::getPrice)

.sum();

**✅ Q5. Average title length of books with more than 200 pages:**

OptionalDouble avgTitleLength = books.stream()

.filter(book -> book.getPages() > 200)

.mapToInt(book -> book.getTitle().length())

.average();

## how we can do the sum and avg for normal streams not with premitive streams

By default, sum() and average() are available only on **primitive streams** (IntStream, DoubleStream, etc.).  
If you're using **non-primitive streams** (like Stream<Employee> or Stream<Integer>), you can still calculate sum and average — but with help from Collectors.

**✅ ✅ Here’s how you do it with normal (non-primitive) streams:**

**🔸 1. Sum using Collectors.summingInt() / summingDouble()**

int total = employees.stream()

.collect(Collectors.summingInt(Employee::getAge));

or

double total = products.stream()

.collect(Collectors.summingDouble(Product::getPrice));

**🔸 2. Average using Collectors.averagingInt() / averagingDouble()**

double avg = students.stream()

.collect(Collectors.averagingDouble(Student::getPercentage));

or

double avg = books.stream()

.collect(Collectors.averagingInt(book -> book.getTitle().length()));

**✅ Example with Wrapper Types (like Integer):**

Even if you're working with a stream like Stream<Integer> (not IntStream):

int sum = integers.stream()

.collect(Collectors.summingInt(Integer::intValue));

double avg = integers.stream()

.collect(Collectors.averagingInt(Integer::intValue));

## minus

**❓ What about minus (subtraction) in streams?**

Unlike sum() or average(), there is **no built-in minus() operation** in Java Streams, because:

* Subtraction is **not associative** (order matters), so it doesn't behave well in parallel processing.
* Summing or averaging is common and meaningful in most data aggregation cases — but subtracting multiple items in a stream is often **context-specific**.

**✅ You can still perform subtraction manually using reduce():**

**🔹 Example 1: Subtract all integers from left to right:**

int result = integers.stream()

.reduce((a, b) -> a - b)

.orElse(0);

⚠️ Important:  
If integers = [100, 20, 10], this will compute 100 - 20 - 10 = 70

**🔹 Example 2: Start from an initial value (like 0):**

int result = integers.stream()

.reduce(0, (a, b) -> a - b);

This behaves like (((0 - a1) - a2) - a3)..., which may not be what you expect.  
So use initial value **only when you really mean it**.

**🔥 Bonus – Conditional subtraction (like price difference):**

double difference = products.stream()

.filter(p -> p.getCategory().equals("Electronics"))

.map(Product::getPrice)

.reduce((a, b) -> a - b)

.orElse(0.0);

So in summary:

* Use reduce() for subtraction
* Be mindful that order **matters**
* Avoid using it in parallel streams unless you're sure it's safe

Fantastic — let’s explore **which to use (sum(), Collectors.summingInt(), etc.) in which situation**, based on **readability, performance, and stream type**.

**✅ Use .sum() / .average() on primitive streams when:**

1. ✅ You already have or can easily convert to a primitive stream using .mapToInt(), .mapToDouble(), etc.
2. ✅ You care about performance — avoids boxing/unboxing.
3. ✅ You want clean, direct code for totals or averages.

**💡 Examples:**

// Best for performance

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

double avg = employees.stream().mapToDouble(Employee::getSalary).average().orElse(0.0);

**✅ Use Collectors.summingInt() / averagingDouble() when:**

1. ✅ You're already using collect() (e.g. as part of a group-by).
2. ✅ You want more **readable, object-oriented syntax**.
3. ✅ You don’t want to deal with optional handling (orElse(0.0)).

**💡 Examples:**

int totalAge = employees.stream().collect(Collectors.summingInt(Employee::getAge));

double avgScore = students.stream().collect(Collectors.averagingDouble(Student::getPercentage));

**📌 Bonus: Works great with grouping:**

Map<String, Double> avgSalaryByDept = employees.stream()

.collect(Collectors.groupingBy(

Employee::getDepartment,

Collectors.averagingDouble(Employee::getSalary)

));

**🆚 Comparison Summary:**

| **Criteria** | **.sum() / .average()** | **Collectors.summingInt() / averagingDouble()** |
| --- | --- | --- |
| ✅ Performance | ⭐⭐⭐⭐⭐ (uses primitives) | ⭐⭐ (boxing/unboxing overhead) |
| ✅ Readability | ⭐⭐⭐ (concise) | ⭐⭐⭐⭐ (fluent with collect) |
| ✅ Grouping support | ❌ | ✅ Works well with groupingBy() |
| ✅ Optional handling needed | ✅ (you may need .orElse()) | ❌ (returns plain value) |
| ✅ Use in combined collectors | ❌ | ✅ Fully supported |

**✅ TL;DR — Which to use?**

| **Scenario** | **Use this** |
| --- | --- |
| You just want the sum/avg from a stream | .sum() / .average() (primitive) |
| You’re working inside collect(...) or groupingBy() | Collectors.summingInt() / averagingDouble() |
| You want best performance for large streams | .sum() with mapToInt() or mapToDouble() |
| You need code that's easier to read in teams | Collectors.summingInt() style |

## ✅ Topic: Collectors Methods

We'll cover these key collectors, one by one with examples and then quiz:

## 🔹 1. toList() / toSet()

Collect stream elements into a List or Set.

List<String> names = employees.stream()

.map(Employee::getName)

.collect(Collectors.toList());

Set<String> depts = employees.stream()

.map(Employee::getDepartment)

.collect(Collectors.toSet());

## 🔹 2. joining()

Concatenate strings from a stream into one string, optionally with delimiters, prefix, and suffix.

String names = employees.stream()

.map(Employee::getName)

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

// Output: [Alice, Bob, Charlie]

## 🔹 3. counting()

Counts the number of elements.

long count = students.stream()

.filter(std -> std.getPercentage() > 90)

.collect(Collectors.counting());

## 🔹 4. summingInt(), summingDouble()

We already covered this — collects and sums values.

int totalAge = employees.stream()

.collect(Collectors.summingInt(Employee::getAge));

## 🔹 5. averagingInt(), averagingDouble()

Collects and averages values.

double avgMarks = students.stream()

.collect(Collectors.averagingDouble(Student::getPercentage));

## 🔹 6. groupingBy()

Groups stream elements by a classifier function into a Map<K, List<T>>.

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

.collect(Collectors.groupingBy(Employee::getDepartment));

With downstream collector (e.g., count employees per department):

Map<String, Long> countPerDept = employees.stream()

.collect(Collectors.groupingBy(Employee::getDepartment, Collectors.counting()));

## 🔹 7. partitioningBy()

Splits elements into two groups (true and false).

Map<Boolean, List<Student>> partitioned = students.stream()

.collect(Collectors.partitioningBy(std -> std.getPercentage() >= 50));

## 🔹 8. mapping() (used with groupingBy)

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

.collect(Collectors.groupingBy(

Employee::getDepartment,

Collectors.mapping(Employee::getName, Collectors.toList())

));

## ✅ Collectors.groupingBy()

**📘 What it does:**

Groups stream elements into a Map<K, List<T>>, or with downstream collectors into other types like Map<K, Long>, Map<K, Double>, etc.

**🔹 Basic form:**

Map<String, List<Employee>> empByDept =

employees.stream().collect(Collectors.groupingBy(Employee::getDepartment));

**🔹 With downstream collector (e.g., counting):**

Map<String, Long> empCountByDept =

employees.stream().collect(Collectors.groupingBy(Employee::getDepartment, Collectors.counting()));

**🔹 With mapping inside grouping:**

Map<String, List<String>> deptToNames =

employees.stream().collect(Collectors.groupingBy(

Employee::getDepartment,

Collectors.mapping(Employee::getName, Collectors.toList())

));

**🔹 Advanced: grouping + averaging:**

Map<String, Double> avgSalaryByDept =

employees.stream().collect(Collectors.groupingBy(

Employee::getDepartment,

Collectors.averagingDouble(Employee::getSalary)

));

**✅ Ready to practice?**

**🔹 Q1:**

From a list of Student objects, group students by their **grade** (e.g., "A", "B", "C") into a Map<String, List<Student>>.

students.stream()

.collect(Collectors.groupingBy(Student::getGrade));

This will return a Map<String, List<Student>>.

**🔹 Q2:**

From a list of Employee objects, get a map of **department to count of employees**.

employees.stream().collect(Collectors.groupingBy(Employee::getDepartment, Collectors.counting()))

**🔹 Q3:**

From a list of Student objects, create a map of **grade to list of student names**  
(i.e., Map<String, List<String>>).

students.stream()

.collect(Collectors.groupingBy(

Student::getGrade,

Collectors.mapping(Student::getName, Collectors.toList())

));

**✅ Q4: groupingBy + averagingDouble**

**Problem:**  
From a list of Employee objects, create a map of  
**department name → average salary**.

You’ll need to use groupingBy() with Collectors.averagingDouble().

employees.stream()

.collect(Collectors.groupingBy(

Employee::getDepartment,

Collectors.averagingDouble(Employee::getSalary)

));

**🔥 Q5: Nested groupingBy**

**Problem:**  
From a list of Student objects, group them by **grade**,  
and within each grade, group them by **pass/fail** based on whether their percentage is ≥ 40.

That is:  
📌 Map<String, Map<Boolean, List<Student>>>  
(grade → [pass/fail → students])

Map<String, Map<Boolean, List<Student>>> result =

students.stream()

.collect(Collectors.groupingBy(

Student::getGrade, // Outer grouping by grade

Collectors.groupingBy(

std -> std.getPercentage() >= 40, // Inner grouping by pass/fail

Collectors.toList()

)

));

**department → gender → list of employee names** (i.e., triple collector)?

Students.stream().collect(Collectors.groupingBy(Student::getGrade, Collectors.groupingBy(std->std.getPercentage()>=40, Collectors.toList())));

**🔥 Q6: Triple Collector Challenge**

From a list of Employee objects, build:

📌 Map<String, Map<String, List<String>>>  
→ Group by **department**, then by **gender**, and collect **employee names**.

**You said:**

employees.stream()

.collect(Collectors.groupingBy(

Employee::getDepartment,

Collectors.groupingBy(

Employee::getGender,

Collectors.mapping(Employee::getName, Collectors.toList())

)

));

## ✅ Step 3: Collectors.reducing()

**📘 What it does:**

Performs a **reduction** like reduce() but within the collect() framework — especially useful with groupingBy().

**🔹 Basic usage:**

int total = numbers.stream()

.collect(Collectors.reducing(0, Integer::intValue, Integer::sum));

* 0 → identity
* Integer::intValue → mapper
* Integer::sum → accumulator

**🔹 In groupingBy:**

**Q1:** From a list of Employee objects, create a map of department → total salary.

You’ll use groupingBy() with Collectors.reducing(...).

Map<String, Double> totalSalaryByDept =

employees.stream()

.collect(Collectors.groupingBy(

Employee::getDepartment,

Collectors.reducing(

0.0, // identity

Employee::getSalary, // mapper

Double::sum // accumulator

)

));

**Q2:**  
Using Collectors.reducing(), compute the **longest book title** from a list of Book objects.

Expected output: Optional<String> or just String  
(Hint: you’ll need Collectors.mapping() + reducing())

## what is identity, mapper, accumulator

Great question! Understanding identity, mapper, and accumulator is key to mastering Collectors.reducing() (and also Stream.reduce()).

Let’s break them down in a very clear way:

**🔹 1. Identity**

A **default value** used when the stream is empty (also the starting point for reduction).

* For summing: 0
* For multiplying: 1
* For string concatenation: "" (empty string)
* For lists: new ArrayList<>()
* For doubles: 0.0

💡 It **guarantees a result**, even if the stream has no elements.

**🔹 2. Mapper**

A function that extracts the value you want to reduce **from each object**.

* Example: If you want to reduce Employee::getSalary, the mapper is:

emp -> emp.getSalary()

**🔹 3. Accumulator**

A **binary operator** that combines two values.

* For summing: Integer::sum, Double::sum, (a, b) -> a + b
* For finding max: Math::max
* For joining: (a, b) -> a + ", " + b

**🔁 How it works together:**

.collect(Collectors.reducing(

0.0, // identity

Employee::getSalary, // mapper: from Employee to Double

Double::sum // accumulator: sum all the doubles

));

It's equivalent to:

double total = 0.0;

for (Employee e : employees) {

total = total + e.getSalary();

}

**✅ Summary Table:**

| **Term** | **Meaning** | **Example** |
| --- | --- | --- |
| Identity | Initial value / default | 0, 0.0, "" |
| Mapper | How to extract the value from object | Employee::getSalary |
| Accumulator | How to combine two values | Double::sum |

## why we need to use Integer::intValue even if we are streaming over int stream

**❓ Why use Integer::intValue even if we're working with int values?**

Actually... **you only need Integer::intValue** if you're working with a **stream of boxed Integer objects**, not an IntStream.

Let’s break it down:

**✅ 1. When you're using Stream<Integer>**

Java streams over wrapper objects like Stream<Integer> do **not** expose .sum() or .average() directly.

If you're reducing them using Collectors.reducing(), you need to extract the primitive int value from the wrapper. That's where Integer::intValue comes in.

List<Integer> nums = List.of(1, 2, 3);

int sum = nums.stream()

.collect(Collectors.reducing(0, Integer::intValue, Integer::sum));

* Integer::intValue is the **mapper**: it converts Integer to int.
* Integer::sum is the **accumulator**.

**✅ 2. When you're using IntStream**

If you're using IntStream, like this:

IntStream.of(1, 2, 3).sum();

Then you **don’t need** Integer::intValue — because you're already working with primitives (int), not Integer.

Same for:

int sum = numbers.stream()

.mapToInt(Integer::intValue)

.sum();

Here, mapToInt(Integer::intValue) converts the stream from Stream<Integer> → IntStream,  
and then .sum() directly gives you the result.

**✅ So in short:**

| **Scenario** | **Need Integer::intValue?** |
| --- | --- |
| Stream<Integer> + collect(reducing(...)) | ✅ Yes |
| Stream<Integer> + mapToInt(...).sum() | ❌ No (already mapped) |
| IntStream directly | ❌ No |

You use Integer::intValue to **unwrap the boxed Integer to a primitive int** when the stream doesn’t already do that for you.

## ✅ Collectors.toMap()

to collect stream results into a **Map**.

**🔹 Basic syntax:**

Collectors.toMap(

keyMapper, // how to derive the key

valueMapper // how to derive the value

)

**🔸 Example 1: Create a Map<id, name> from a list of Employee**

Map<Integer, String> empMap = employees.stream()

.collect(Collectors.toMap(

Employee::getId, // key = ID

Employee::getName // value = Name

));

**🔸 Example 2: Word → its length**

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

.collect(Collectors.toMap(

word -> word, // key = word

word -> word.length() // value = length

));

**⚠️ What if keys collide?**

If multiple elements produce the **same key**, toMap() will throw IllegalStateException.

To handle it, use the **3-arg version** of toMap():

Collectors.toMap(

keyMapper,

valueMapper,

(existing, replacement) -> existing // or replacement, or combine both

)

**Example:**

java

CopyEdit

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

.collect(Collectors.toMap(

Item::getCategory,

Item::getName,

(name1, name2) -> name1 + ", " + name2 // merge function

));

**🔹 Also: You can specify the map type (like TreeMap):**

java

CopyEdit

Collectors.toMap(

keyMapper,

valueMapper,

mergeFunction,

TreeMap::new

)

**✅ Summary:**

| **Purpose** | **Collector** |
| --- | --- |
| To list | Collectors.toList() |
| To set | Collectors.toSet() |
| To map | Collectors.toMap(k, v) |
| To map (with merge) | Collectors.toMap(k, v, merge) |
| To TreeMap, etc. | Collectors.toMap(k, v, merge, mapFactory) |

### how the **third argument** in Collectors.toMap(...) handles **collisions** (i.e., when multiple elements produce the same key).

**✅ Syntax:**

Collectors.toMap(

keyMapper,

valueMapper,

mergeFunction // handles key collisions

)

**🔹 Why do collisions happen?**

Because the **keyMapper** function might produce **duplicate keys**.

Example:

List<String> names = List.of("John", "Jack", "Jane", "Joe");

If you use .toMap(name -> name.length(), name -> name)  
Then:

* "John", "Jack", "Jane", and "Joe" all have length 4 or 3
* So you may get a collision on the key 4

**🔥 The merge function is used to resolve the conflict:**

(existingValue, newValue) -> ...

It’s a **BiFunction** that takes:

* the **existing value** for the key
* the **new value** that caused the collision
* returns the value to keep

**🧠 Example:**

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

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

.collect(Collectors.toMap(

name -> name.charAt(0), // key: first letter

name -> name, // value: full name

(existing, replacement) -> existing + ", " + replacement // merge on collision

));

📌 Output:

{

A=Alice, Alex,

B=Bob,

C=Charlie

}

**🔄 Alternate merge functions:**

* Keep the **first** value: (e, r) -> e
* Keep the **last** value: (e, r) -> r
* Combine them (e.g., list of values)

**⚠️ Without merge function:**

names.stream().collect(Collectors.toMap(

name -> name.charAt(0),

name -> name

));

🚫 This will throw IllegalStateException if two names start with the same letter.

## when to use IntStream, Arrays.stream, Strema.of() I am very confused and what it returns?

**🔷 1. IntStream**

IntStream is a **primitive stream of int values**, part of the java.util.stream package.

**🔹 When to use:**

* When you're working with **primitive int[]** or need numeric operations like sum(), average(), etc.
* To avoid boxing/unboxing overhead of Stream<Integer>

**🔹 How to get it:**

IntStream.range(1, 5); // 1, 2, 3, 4

IntStream.of(1, 2, 3); // 1, 2, 3

Arrays.stream(new int[]{1, 2}); // 1, 2

**🔹 What it returns:**

An IntStream — supports .sum(), .average(), .map(), etc., **but not .collect(Collectors.toList()) unless you call .boxed()**.

**🔷 2. Arrays.stream()**

This method creates a stream **from an array**. Its behavior **depends on the array type**:

| **Input** | **Resulting Stream Type** |
| --- | --- |
| int[] | IntStream |
| String[] | Stream<String> |
| Integer[] | Stream<Integer> |

**🔹 Examples:**

Arrays.stream(new int[]{1, 2, 3}) // returns IntStream

Arrays.stream(new String[]{"a", "b"}) // returns Stream<String>

**🔷 3. Stream.of()**

Used to create a stream from **varargs or single objects**.

**🔹 When to use:**

* You have specific values or want to wrap a few items into a stream
* You want to stream from an array and don't need primitive specialization

**🔹 Examples:**

Stream.of(1, 2, 3); // Stream<Integer>

Stream.of(new String[]{"a", "b"}) // Stream<String[]>

Stream.of("a", "b", "c") // Stream<String>

Stream.of(array).flatMap(Arrays::stream) // Flattened

⚠️ If you do Stream.of(new int[]{1,2,3}), it creates a **Stream<int[]>**, not IntStream.