STREAM API — ULTIMATE SHORT-NOTES (DETAILED YET CRISP)

---

1) What is Stream API?

- Stream = *flow of data* on which operations are applied.
- NOT a data structure, NOT storage.
- Provides functional-style processing of collections.
- Supports **pipeline operations** (map → filter → sort → collect).
- Introduced in Java 8.

---

2) Stream Pipeline Structure

```
Source → Intermediate Ops (Lazy) → Terminal Op (Execution)
```

- Source = list.stream(), Arrays.stream(), Stream.of()
- Intermediate = return Stream (filter, map, sorted…)
- Terminal = returns result & **closes stream** (collect, reduce…)

---

3) Why Streams? (Real Purpose)

- Less boilerplate code
- Declarative, functional programming
- Easy transformations
- Built-in filtering, mapping, sorting
- Natural parallelism support
- Lazy evaluation (performance)

---

4) Stream Creation

```
List → list.stream()
Array → Arrays.stream(arr)
Values → Stream.of(1,2,3)
Infinite → Stream.generate(), Stream.iterate()
```

---

5) Intermediate Operations (Lazy)

1) filter(predicate)

Keep only elements that match condition.

```
.filter(n -> n > 10)
```

2) map(function)

Transform each element → new form.

```
.map(String::toUpperCase)
```

3) flatMap(function)

Flatten nested structure + transform.

```
.flatMap(list -> list.stream())
```

4) distinct()

Remove duplicates using equals().

5) sorted()

Sort data (default natural order).

```
.sorted()
.sorted(Comparator.comparingInt(Student::getAge))
```

6) limit(n) / skip(n)

Take first n elements / skip first n.

7) peek()

Debug inside stream pipeline.

---

6) Terminal Operations

1) collect()

Convert stream → List/Set/Map

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

2) forEach()

Perform action for each element.

3) reduce()

Aggregate many values → one result.

```
.reduce(0, (a,b) -> a+b)
```

4) count()

Number of elements.

5) findFirst() / findAny()

Short-circuit operations.

6) max()/min()

Find largest/smallest.

---

7) map() vs flatMap() (Super Important)

- **map:** one → one
- **flatMap:** one → many + flatten
- map returns Stream<R>
- flatMap returns Stream<T> from nested values

---

8) Lazy Evaluation

- Intermediate operations don’t run immediately.
- Execution starts only when terminal operation is called.
- Helps performance + optimization.

---

9) Short-Circuit Operations

Stop processing early:

- findFirst

- anyMatch
- noneMatch
- limit

These increase performance.

---

10) reduce() (Aggregation)

Used to calculate:

- sum
- product
- max/min
- concatenation

Structure:

```
reduce(identity, accumulator)
```

Example:

```
nums.stream().reduce(0, (a,b) -> a+b);
```

---

11) Collectors — MOST POWERFUL PART

✔ toList(), toSet(), toMap()

Convert stream → collection.

✔ groupingBy()

Group elements by key.

```
groupingBy(Student::getDept)
```

✔ partitioningBy()

Boolean-based grouping (True/False).

✔ counting()

Count elements in each group.

✔ summingInt()

Sum values in group.

✔ maxBy(), minBy()

Find max/min element inside group.

✔ joining()

Join strings with separator.

---

12) Grouping Use Cases

Group by Dept:

```
groupingBy(Employee::getDept)
```

Count per group:

```
groupingBy(..., counting())
```

Highest salary per group:

```
groupingBy(..., maxBy(comparator))
```

Total marks per group:

```
groupingBy(..., summingInt(Student::getMarks))
```

---

13) Stream Limitations

- Stream cannot be reused → “stream has already been operated upon or closed”
- Cannot modify original collection (no mutation)
- Not good for index-based operations
- Parallel stream unsafe for mutable shared data

---

15) Common Interview Questions (One-Liners)

✔ Why streams cannot be reused?

Because terminal operation consumes its internal Spliterator.

✔ Why flatMap needed?

map → produces nested streams; flatMap removes nesting.

✔ difference between map & filter?

map → transform
filter → keep/remove

✔ orElse vs orElseGet?

orElse runs always; orElseGet runs lazily.

✔ What is lazy evaluation?

Operations run only when terminal operator is called.

✔ Intermediate vs Terminal?

Intermediate returns Stream; terminal returns final result.

16) Full Real-World Stream Pipeline

```
employees.stream()
         .filter(e -> e.getSalary() > 50000)
         .sorted(Comparator.comparing(Employee::getAge))
         .map(Employee::getName)
         .distinct()
         .limit(5)
         .collect(Collectors.toList());
```

17) Stream API Golden Rules

- Streams are ONE-TIME use
- Always prefer pure functions (no side-effects)
- Keep pipelines readable
- Use method references when possible
- Avoid parallel stream unless necessary

Concept	In One Line
filter	keep some elements
map	convert elements
flatMap	flatten nested structures
sorted	sort elements
distinct	remove duplicates
reduce	combine to single result
collect	produce final collection
groupingBy	group into categories
partitioningBy	split into true/false
findFirst	first element
findAny	random element (parallel fast)
max/min	biggest/smallest
limit/skip	slicing
any/all/noneMatch	boolean checks
parallelStream	multi-thread processing

STREAM API — SHORT USE-CASES FOR EVERY CONCEPT

---

1) stream()

Use case: Convert list → stream for processing

```
list.stream();
```

---

2) filter()

Use case: Get students with marks > 60

```
list.stream().filter(s -> s.getMarks() > 60);
```

---

3) map()

Use case: Extract list of names

```
list.stream().map(Emp::getName);
```

---

4) flatMap()

Use case: Flatten List<List<Integer>> → List<Integer>

```
data.stream().flatMap(l -> l.stream());
```

---

5) sorted()

Use case: Sort employees by salary

```
list.stream().sorted(Comparator.comparingInt(Emp::getSalary));
```

---

6) distinct()

Use case: Remove duplicate elements

```
list.stream().distinct();
```

---

7) limit()

Use case: Get first 5 records

```
list.stream().limit(5);
```

---

8) skip()

Use case: Skip first 3 elements

```
list.stream().skip(3);
```

---

9) peek()

Use case: Debug values inside pipeline

```
list.stream().peek(System.out::println);
```

---

10) collect()

Use case: Convert stream → List

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

---

11) reduce()

Use case: Sum of numbers

```
nums.stream().reduce(0, (a,b) -> a+b);
```

---

12) findFirst()

Use case: Get first matching user

```
list.stream().findFirst();
```

---

13) findAny()

Use case: Faster fetch in parallelStream

```
list.parallelStream().findAny();
```

---

14) max()

Use case: Find employee with highest salary

```
list.stream().max(Comparator.comparingInt(Emp::getSalary));
```

---

15) min()

Use case: Find lowest marks student

```
list.stream().min(Comparator.comparingInt(Stud::getMarks));
```

---

16) anyMatch()

Use case: Does any student score > 90?

```
list.stream().anyMatch(s -> s.getMarks() > 90);
```

---

17) allMatch()

Use case: Check all employees are active

```
list.stream().allMatch(Emp::isActive);
```

---

18) noneMatch()

Use case: No product out of stock?

```
list.stream().noneMatch(p -> p.getQty() == 0);
```

Collectors — Short Use Cases

---

19) groupingBy()

Use case: Group employees by department

```
groupingBy(Emp::getDept)
```

---

20) partitioningBy()

Use case: Split into pass/fail

```
partitioningBy(s -> s.getMarks() > 60)
```

---

21) counting()

Use case: Count students per category

```
groupingBy(..., counting())
```

---

22) summingInt()

Use case: Total salary per department

```
groupingBy(..., summingInt(Emp::getSalary))
```

---

23) maxBy()

Use case: Top scorer in each group

```
groupingBy(..., maxBy(comparingInt(S::getMarks)))
```

---

24) joining()

Use case: Convert List<String> → CSV

```
joining(","
```