1. Why Stream API Exists

Before Java 8, most collection processing was written using:

- `for` / `for-each` loops
- temporary lists
- manual counters / sums
- lots of boilerplate code

That approach works, but it often becomes:

- repetitive (same patterns again and again)
- error-prone (forgetting edge cases, wrong accumulators)
- harder to read (logic scattered across loops)

Stream API provides a better style:

You write operations like a **data query**:

> "From these students, keep only those scoring ≥ 70, sort them, take top 3, collect them."

That reads like a pipeline, not like step-by-step mechanics.

---

2. Example Data Model — Student

```
class Student implements Comparable<Student> {

    private String name;
    private int score;

    public Student(String name, int score) {
        this.name = name;
        this.score = score;
    }
```

```
    public String getName() { return name; }
    public int getScore() { return score; }

    @Override
    public String toString() {
        return this.name + " - " + this.score;
    }

    @Override
    public int compareTo(Student another) {
        return another.getScore() - this.score; // descending order
    }
}
```

Explanation

- `Comparable<Student>` means: objects know how to compare themselves for sorting.
- `compareTo` returns descending order: higher scores come first.
- `toString()` controls how objects print in console.

---

3. Sample Student Data

```
List<Student> listStudents = new ArrayList<>();

listStudents.add(new Student("Alice", 82));
listStudents.add(new Student("Bob", 90));
listStudents.add(new Student("Carol", 67));
listStudents.add(new Student("David", 80));
listStudents.add(new Student("Eric", 55));
listStudents.add(new Student("Frank", 49));
listStudents.add(new Student("Gary", 88));
listStudents.add(new Student("Henry", 98));
listStudents.add(new Student("Ivan", 66));
listStudents.add(new Student("John", 52));
```

---

4. Task 1 — Filter students with score ≥ 70

✅ Traditional Non-Stream Approach

```
List<Student> listGoodStudents = new ArrayList<>();

for (Student student : listStudents) {
    if (student.getScore() >= 70) {
        listGoodStudents.add(student);
    }
}

for (Student student : listGoodStudents) {
    System.out.println(student);
}
```

What happens here (step-by-step)

1. You create a new empty list.
2. You manually loop through each student.
3. You check the condition.
4. You add matching students to the list.
5. You loop again to print them.

This works, but it takes multiple steps and extra code.

✅ Stream Approach

```
List<Student> listGoodStudents = listStudents.stream()
        .filter(s -> s.getScore() >= 70)
        .collect(Collectors.toList());

listGoodStudents.forEach(System.out::println);
```

Explanation of each part

- `listStudents.stream()`
  Creates a stream (a processing pipeline) from the list.

- `.filter(s -> s.getScore() >= 70)`
  Keeps only elements where the predicate is `true`.
  Predicate = function returning boolean.

- `.collect(Collectors.toList())`
  Terminal operation: converts the stream result into a `List`.

- `forEach(System.out::println)`
  Prints each element.

Key teaching point

Streams are about *describing transformations*, not controlling the loop.

---

5. Task 2 — Average Score of all Students

✅ Traditional Approach

```
double sum = 0;

for (Student student : listStudents) {
    sum += student.getScore();
}

double average = sum / listStudents.size();
System.out.println("Average score: " + average);
```

What can go wrong here?

- Division by zero if list is empty.

- Manual sum logic is repetitive.

✅ Stream Approach

```
double average = listStudents.stream()
```

```
        .mapToInt(Student::getScore)
        .average()
        .getAsDouble();

System.out.println("Average score: " + average);
```

Explanation

- `mapToInt(Student::getScore)`
  Converts `Stream<Student>` into `IntStream` (primitive stream of int).

Why?
Primitive streams (`IntStream`, `LongStream`, `DoubleStream`) support numeric operations like `sum()`, `average()`, `min()`, `max()` efficiently.

- `.average()` returns `OptionalDouble`
  Because average may not exist if stream is empty.

- `.getAsDouble()` extracts value
  ⚠ Unsafe if stream is empty.

Safe version (recommended for teaching)

```
double average = listStudents.stream()
        .mapToInt(Student::getScore)
        .average()
        .orElse(0.0);
```

---

6. What is a Stream?

A Stream is:

✅ A sequence of elements from a source
✅ Supports aggregate operations
✅ Can be processed sequentially or in parallel
✅ Lazy evaluation (does not execute until terminal operation)

A Stream is NOT:
❌ A data structure (it doesn't store data)

❌ A collection (cannot add/remove elements)
❌ Reusable (once consumed, it's finished)

---

7. Creating Streams

From a Collection

```
Stream<Student> stream = listStudents.stream();
```

From an Array

```
int[] numbers = {1, 8, 2, 3};
IntStream s = Arrays.stream(numbers);
```

From a File

```
BufferedReader br = new BufferedReader(new
FileReader("students.txt"));
Stream<String> lines = br.lines();
```

---

8. Stream Pipeline Structure

Every stream pipeline has:

1. **Source** (collection/array/file/etc.)
2. **Intermediate operations** (0 or more)
3. **Terminal operation** (exactly 1)

Example:

```
List<Student> top3Students = listStudents.stream()
        .filter(s -> s.getScore() >= 70)
        .sorted()
        .limit(3)
        .collect(Collectors.toList());
```

How to read it like English:

“From students → keep those ≥ 70 → sort them → take first 3 → store into list.”

---

9. Intermediate Operations (Detailed)

Intermediate operations:

- return a NEW stream
- don’t execute immediately
- are lazy until terminal operation is called

Common Intermediate Ops

9.1 filter(Predicate)

Keeps elements matching condition.

```
stream.filter(x -> condition)
```

9.2 map(Function)

Transforms each element into something else (1-to-1 transform).

Example: Student → Name

```
listStudents.stream()
    .map(Student::getName)
    .forEach(System.out::println);
```

9.3 mapToInt / mapToLong / mapToDouble

Used when you want numeric computations.

Example: Student → score (int)

```
listStudents.stream()
    .mapToInt(Student::getScore)
    .sum();
```

9.4 flatMap(Function)

Used when each element produces multiple values (flattening).

Example: List<List<String>> → Stream<String>

```
nested.stream()
    .flatMap(List::stream)
    .forEach(System.out::println);
```

9.5 distinct()

Removes duplicates based on `equals()` and `hashCode()`.

9.6 sorted()

- `sorted()` uses natural ordering (`Comparable`)
- `sorted(Comparator)` uses custom rules

9.7 limit(n)

Takes first `n` elements.

9.8 skip(n)

Skips first `n` elements.

9.9 peek(Consumer)

Used mainly for debugging.

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

⚠ Don't use `peek()` for business logic (it's not meant for side effects).

---

10. Terminal Operations (Detailed)

Terminal operations:

- trigger execution
- produce final result or side effect
- end the stream (cannot reuse)

Common Terminal Ops

10.1 forEach(Consumer)

Performs action (side effect).

10.2 collect(Collector)

Collect results into a container.

10.3 count()

Counts elements.

10.4 min / max

Return Optional result.

10.5 reduce()

General purpose “combine into one result” operation.

10.6 anyMatch / allMatch / noneMatch

Return boolean and short-circuit.

10.7 findFirst / findAny

Return Optional.

---

11. Optional in Streams (Important Concept)

Many terminal ops return Optional because result may not exist.

Example: min score from empty list is undefined.

Safe optional usage

```
Optional<Student> best = listStudents.stream()
        .max(Comparator.comparingInt(Student::getScore));

best.ifPresent(System.out::println);
```

---

12. Parallel Streams (Detailed)

```
listStudents.parallelStream()
    .filter(s -> s.getScore() >= 70)
    .forEach(System.out::println);
```

What happens internally?

- Stream is split into chunks
- Each chunk processed by different threads (ForkJoinPool)
- Results combined

When parallel helps

✅ very large data sets
✅ CPU-heavy computations
✅ stateless operations

When parallel harms

❌ small lists
❌ lots of side effects (`forEach` printing)
❌ IO operations
❌ shared mutable state

---

13. Streams vs Collections (Clear Difference)

Collections	Streams
store elements	process elements
eager	lazy
can mutate	does not mutate source
reusable	single-use

---

14. Collectors Deep Dive (Very Useful for Teaching)

14.1 toList(), toSet()

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

14.2 toMap()

```
Map<String, Student> map = listStudents.stream()
    .collect(Collectors.toMap(Student::getName, s -> s));
```

⚠ duplicate key requires merge function.

14.3 groupingBy()

Group by category (returns map).

```
Map<Integer, List<Student>> byScore =
    listStudents.stream()
        .collect(Collectors.groupingBy(Student::getScore));
```

14.4 partitioningBy()

Always creates 2 buckets (true/false).

```
Map<Boolean, List<Student>> partition =
    listStudents.stream()
        .collect(Collectors.partitioningBy(s -> s.getScore() >=
70));
```

14.5 counting()

```
long count = listStudents.stream()
    .collect(Collectors.counting());
```

14.6 mapping() (downstream collector)

```
Map<Boolean, List<String>> names =
    listStudents.stream()
        .collect(Collectors.partitioningBy(
            s -> s.getScore() >= 70,
            Collectors.mapping(Student::getName,
Collectors.toList())
        ));
```

14.7 joining()

```
String names = listStudents.stream()
    .map(Student::getName)
    .collect(Collectors.joining(", "));
```

---

15. Teaching Pitfalls to Highlight

✅ 1. Streams cannot be reused

```
Stream<Student> s = listStudents.stream();
s.count();
s.forEach(System.out::println); // ERROR (stream already consumed)
```

✅ 2. Optional must be handled

Avoid `.get()` blindly.

✅ 3. Side effects reduce stream quality

Avoid modifying external variables inside streams.

Bad:

```
List<Student> out = new ArrayList<>();
listStudents.stream().forEach(out::add);
```

Better:

```
List<Student> out =
listStudents.stream().collect(Collectors.toList());
```

---

16. Quick Summary for Students

- Stream = pipeline to process data
- Intermediate operations are lazy and return streams
- Terminal operation triggers execution and produces result
- Streams do not change the source collection
- Collectors provide powerful grouping + mapping + partitioning operations