Stream API introduced in 👉 Java 8
Main package 👉 java.util.stream

📘 1️⃣ What is Stream API?

A **Stream** is a sequence of elements that supports functional-style operations to process data.

👉 It does **NOT store data**
👉 It processes data from collections (List, Set, Map, etc.)

Think like:

Collection (data) → Stream (processing pipeline) → Result

📘 2️⃣ Why Stream API?

Before Java 8:

```
List<Integer> list = Arrays.asList(10,20,30,40);

for(Integer i : list){
   if(i % 2 == 0){
      System.out.println(i);
   }
}
```

After Stream:

```
list.stream()
   .filter(i -> i % 2 == 0)
   .forEach(System.out::println);
```

✓ Less code
✓ Readable
✓ Functional style
✓ Parallel processing support

📘 3️⃣ Stream Characteristics

- Does NOT modify original collection
- Lazy evaluation
- Can be used only once
- Supports parallel processing

📘 4️⃣ How to Create Stream?

1️⃣ From Collection

list.stream();

2️⃣ From Array

Arrays.stream(array);

3️⃣ Using Stream.of()

Stream.of(1,2,3,4);

4️⃣ Infinite Stream

Stream.iterate(1, n -> n+1);
Stream.generate(Math::random);

📘 5️⃣ Stream Pipeline Structure

Every stream has 3 parts:

1. Source
2. Intermediate Operations
3. Terminal Operation

Example:

```
list.stream()          // Source
   .filter(x -> x>10)   // Intermediate
   .sorted()          // Intermediate
   .collect(Collectors.toList());  // Terminal
```

📘 6️⃣ Intermediate Operations

These return Stream again.

🔹 filter()

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

Keeps only matching elements.

🔹 map()

Transforms elements.

.map(x -> x * 2)

Example:

[1,2,3] → [2,4,6]

🔹 **flatMap()**

Used for nested collections.

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

Example:

[[1,2],[3,4]] → [1,2,3,4]

🔹 **sorted()**

Default → ascending

.sorted()

Custom:

.sorted((a,b) -> b-a) // descending

🔹 **distinct()**

Removes duplicates.

🔹 **limit() / skip()**

.limit(3)
.skip(2)

📘 7️⃣ Terminal Operations

Ends stream.

🔹 **forEach()**

.forEach(System.out::println);

🔹 **collect()**

Most powerful.

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

Collectors class → java.util.stream.Collectors

🔹 **reduce()**

Used for aggregation.

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

Example:

[1,2,3,4]
Step1: 0+1=1
Step2: 1+2=3
Step3: 3+3=6
Step4: 6+4=10

🔹 **count()**

.count();

🔹 **findFirst() / findAny()**

Returns Optional.

🔹 **allMatch / anyMatch / noneMatch**

.allMatch(x -> x>0)
.anyMatch(x -> x<0)
.noneMatch(x -> x<0)

📘 8️⃣ **Comparator in Stream**

Default sorting → natural order.

Custom:

.sorted((a,b) -> a.compareTo(b))

Descending:

.sorted(Comparator.reverseOrder())

📘 9️⃣ **Grouping & Partitioning**

🔹 **groupingBy()**

Returns Map

.collect(Collectors.groupingBy(x -> x));

Example:

["apple","apple","banana"]
→ {apple=2, banana=1}

🔹 **partitioningBy()**

Returns Map<Boolean, List>

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

📘 🔟 **Optional Class**

Stream returns Optional sometimes.

Class → java.util.Optional

Used to avoid NullPointerException.

```
Optional<Integer> opt = list.stream().findFirst();
opt.get();
opt.orElse(0);
```

📘 1️⃣1️⃣ **Parallel Stream**

list.parallelStream()

Used for multi-core processing.

⚠ Not good for small data.

📘 1️⃣2️⃣ **Difference: Collection vs Stream**

Collection	**Stream**
Stores data	Processes data
Eager	Lazy
Can reuse	One time use

📘 1️⃣3️⃣ **Common Interview Questions**

✓ Difference between map and flatMap
✓ Difference between reduce and collect
✓ Why stream is lazy?
✓ How parallel stream works?
✓ Why Optional used?

📘 1️⃣4️⃣ Internal Working (Concept)

Stream works like pipeline:

Data → filter → map → sorted → collect

Lazy evaluation:
Operations execute only when terminal operation runs.

📘 1️⃣5️⃣ Real-World Example (Employee)

```
employees.stream()
        .filter(e -> e.getSalary() > 50000)
        .sorted(Comparator.comparing(Employee::getSalary))
        .collect(Collectors.toList());
```

📘 1️⃣6️⃣ Performance Notes

✓ Avoid using stream in simple loops
✓ Prefer parallel only for large data
✓ Avoid modifying shared variables

📘 1️⃣7️⃣ Summary Cheat Sheet

Most Used:

filter
map
flatMap
sorted
distinct
limit
skip
collect
reduce
groupingBy
partitioningBy