17. Stream API

`Stream` 생성 및 가공

✅ 1. Stream 생성 방법

Java에서 Stream을 생성하는 방법은 매우 다양하며, 대표적으로 다음과 같다:

🔹 컬렉션 기반 생성

```
List<String> list = List.of("A", "B", "C");
Stream<String> stream = list.stream();
```

🔹 배열 기반 생성

```
String[] array = {"A", "B", "C"};
Stream<String> stream = Arrays.stream(array);
```

🔹 Stream.of()

```
Stream<Integer> stream = Stream.of(1, 2, 3, 4);
```

🔹 Stream.builder()

```
Stream<String> stream = Stream.<String>builder()
    .add("A").add("B").add("C")
    .build();
```

🔹 무한 스트림 (iterate, generate)

```
Stream<Integer> infinite = Stream.iterate(0, n -> n + 2); // 짝수 무한
Stream<Double> randoms = Stream.generate(Math::random);   // 무작위 무한
```

> 무한 스트림은 `.limit(n)`으로 제한해서 사용해야 함

✅ 2. 중간 연산 (Intermediate Operations)

중간 연산은 **데이터를 가공하거나 필터링하거나 정렬하거나 변경**하는 연산이다.
💡 *Stream 자체를 반환하므로 체이닝이 가능하다.*

📌 filter(Predicate)

```
stream.filter(s -> s.startsWith("A"))
```

조건을 만족하는 요소만 통과

📌 map(Function)

```
stream.map(String::toLowerCase)
```

값을 변환 → `"A"` → `"a"` 등

📌 flatMap(Function)

```
Stream<String> lines = Stream.of("a b", "c d");
Stream<String> words = lines.flatMap(line -> Arrays.stream(line.split(" ")));
```

`Stream<Stream<T>>` → `Stream<T>` 평탄화

📌 distinct()

```
stream.distinct()
```

중복 제거

📌 sorted(), sorted(Comparator)

```
stream.sorted()
stream.sorted(Comparator.reverseOrder())
```

기본 정렬 or 사용자 정의 정렬

📌 peek(Consumer)

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

중간 확인용 디버깅 (side-effect만 가능)

📌 limit(n), skip(n)

```
stream.limit(5) // 앞에서 5개만
stream.skip(3)  // 앞에서 3개 건너뜀
```

✅ 3. 최종 연산 (Terminal Operations)

최종 연산은 Stream 처리를 종료하고 결과를 반환한다.

연산	설명
`forEach(Consumer)`	요소 순회
`collect(Collectors)`	List, Set, Map 등으로 수집
`reduce(BinaryOperator)`	누적 연산
`count()`	개수 반환
`min()`, `max()`	최소/최대값
`anyMatch()`, `allMatch()`, `noneMatch()`	조건 만족 여부
`findFirst()`, `findAny()`	요소 반환 (Optional)

✅ 4. 실전 예시

```
List<String> names = List.of("Kim", "Lee", "Park", "Kang", "Kim");

List<String> result = names.stream()
    .filter(n -> !n.equals("Kim"))
    .map(String::toUpperCase)
    .distinct()
    .sorted()
    .collect(Collectors.toList());

System.out.println(result); // [KANG, LEE, PARK]
```

🧠 요약 정리표

목적	메서드	예시
Stream 생성	`stream()`, `of()`, `Arrays.stream()`	`List.of(1,2,3).stream()`
필터링	`filter(Predicate)`	`.filter(x -> x > 0)`
변환	`map(Function)`	`.map(String::length)`
평탄화	`flatMap(Function)`	`.flatMap(...)`
정렬	`sorted()`	`.sorted(Comparator.reverseOrder())`
중복 제거	`distinct()`	`.distinct()`
일부만 추출	`limit(n)`, `skip(n)`	`.limit(5)`
최종 처리	`forEach`, `collect`, `reduce`	`.collect(Collectors.toList())`

중간 연산: `filter`, `map`, `distinct`, `sorted`

🟦 1. `filter(Predicate<T>)`

설명: 조건에 맞는 요소만 골라냄 (boolean 반환하는 람다 필요)

📌 기본 형태:

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

📌 예시:

```
List<String> names = List.of("Alice", "Bob", "Angela", "David");

names.stream()
     .filter(name -> name.startsWith("A"))
     .forEach(System.out::println);
// 출력: Alice, Angela
```

🔍 내부적으로

`filter`는 내부적으로 `boolean test(T t)` 메서드를 가지는 `Predicate` 인터페이스를 이용해서 조건을 평가한다.
`true`면 stream에 남고, `false`면 제외됨.

🟧 2. `map(Function<T, R>)`

설명: 각 요소를 **다른 값으로 변환**함. (예: 대문자로, 길이로, 다른 타입으로)

📌 기본 형태:

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

📌 예시:

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

names.stream()
     .map(String::toUpperCase)
     .forEach(System.out::println);
// 출력: ALICE, BOB
```

또는

```
List<String> names = List.of("Apple", "Banana", "Carrot");

names.stream()
     .map(String::length)
     .forEach(System.out::println);
// 출력: 5, 6, 6
```

🔍 내부적으로

`map`은 `Function<T, R>`을 이용해서 각 요소에 변환 함수를 적용한 다음, 새로운 Stream을 만들어 반환한다.

🟩 3. `distinct()`

설명: 중복을 제거한 Stream 반환 (equals/hashCode 기준)

📌 기본 형태:

```
Stream<T> distinct()
```

📌 예시:

```
List<Integer> nums = List.of(1, 2, 2, 3, 3, 3, 4);

nums.stream()
    .distinct()
    .forEach(System.out::println);
// 출력: 1, 2, 3, 4
```

🔍 내부적으로

`distinct()`는 내부적으로 `HashSet`을 사용하여 `equals()`와 `hashCode()` 기준으로 중복을 걸러냄. 따라서 성능은 O(n) 정도지만, 객체의 `equals`, `hashCode` 구현에 따라 동작이 달라짐.

🟥 4. `sorted()` / `sorted(Comparator)`

설명: 요소를 정렬하여 새로운 Stream을 생성

- 기본 정렬: `Comparable` 인터페이스 기준
- 사용자 정렬: `Comparator<T>` 전달

📌 기본 형태:

```
Stream<T> sorted() // 자연 순서
Stream<T> sorted(Comparator<? super T> comparator)
```

📌 예시 1: 기본 정렬

```
List<Integer> nums = List.of(5, 3, 9, 1);

nums.stream()
    .sorted()
    .forEach(System.out::println);
// 출력: 1, 3, 5, 9
```

📌 예시 2: 역순 정렬

```
nums.stream()
    .sorted(Comparator.reverseOrder())
    .forEach(System.out::println);
// 출력: 9, 5, 3, 1
```

📌 예시 3: 객체 정렬

```
List<String> names = List.of("Kim", "Lee", "Park");

names.stream()
     .sorted((a, b) -> b.length() - a.length())
     .forEach(System.out::println);
// 출력: "Park", "Kim", "Lee"
```

🧠 요약 비교표

연산	인터페이스	주요 용도
`filter`	`Predicate<T>`	조건 필터링
`map`	`Function<T, R>`	값 변환
`distinct`	내부 equals/hashCode 사용	중복 제거
`sorted`	`Comparable` or `Comparator`	정렬

💡 실전 예제: 4개 조합

```
List<String> words = List.of("apple", "banana", "apple", "cherry", "banana", "date");

words.stream()
     .filter(w -> w.length() > 5)           // 길이 6 이상
     .map(String::toUpperCase)              // 대문자 변환
     .distinct()                            // 중복 제거
     .sorted()                              // 알파벳 정렬
     .forEach(System.out::println);

// 출력:
// BANANA
// CHERRY
```

최종 연산: `forEach`, `collect`, `reduce`, `count`

✅ 1. `forEach(Consumer<T>)`

설명: 스트림의 각 요소에 대해 어떤 작업을 수행함 (출력, 저장 등 부수 효과(side effect)용)

📌 기본 구조

```
void forEach(Consumer<? super T> action)
```

📌 예제

```
List<String> list = List.of("A", "B", "C");
list.stream().forEach(System.out::println);
// 출력: A B C
```

⚠️ 주의

- 순서를 보장하지 않음 → 병렬 스트림(parallelStream)에서는 특히 주의
- 부작용(side-effect)에만 사용하고 Stream 내부 로직에는 영향 없음

✅ 2. `collect(Collector<T, A, R>)`

설명: 스트림의 요소를 **컬렉션(List, Set, Map)** 으로 모으거나, 통계를 내거나, 문자열로 합치거나 등 다양한 결과로 수집

📌 가장 자주 쓰는 Collectors

```
Collectors.toList()
Collectors.toSet()
Collectors.toMap()
Collectors.joining()
Collectors.counting()
Collectors.averagingInt()
Collectors.groupingBy()
Collectors.partitioningBy()
```

📌 예제 1: List로 수집

```
List<String> upper = List.of("a", "b", "c")
    .stream()
    .map(String::toUpperCase)
    .collect(Collectors.toList());
```

📌 예제 2: Map으로 수집

```
Map<String, Integer> map = List.of("a", "bb", "ccc")
    .stream()
    .collect(Collectors.toMap(s -> s, String::length));
// 결과: {"a":1, "bb":2, "ccc":3}
```

✅ 3. `reduce()` — 누산 연산

설명: 모든 요소를 하나의 결과값으로 줄여나가는 연산 (합계, 곱, 최댓값 등)

📌 기본 구조

```
Optional<T> reduce(BinaryOperator<T> accumulator)
T reduce(T identity, BinaryOperator<T> accumulator)
<U> U reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U>
combiner) // 병렬
```

📌 예제 1: 합계 구하기

```
int sum = List.of(1, 2, 3, 4)
    .stream()
    .reduce(0, (a, b) -> a + b);
// 결과: 10
```

📌 예제 2: 가장 긴 문자열

```
Optional<String> longest = List.of("a", "bbb", "cc")
    .stream()
    .reduce((s1, s2) -> s1.length() > s2.length() ? s1 : s2);
// 결과: Optional["bbb"]
```

✅ 4. `count()`

설명: 스트림의 요소 개수 반환 (`long`)

📌 예제

```
long cnt = List.of(1, 2, 3, 4, 5)
    .stream()
    .filter(n -> n % 2 == 0)
    .count();
// 결과: 2 (2와 4)
```

🧠 요약 정리표

연산	설명	반환 타입	예시
`forEach`	요소마다 작업 수행	void	출력, side-effect
`collect`	리스트, 맵, 문자열 등으로 수집	List, Map 등	`.collect(Collectors.toList())`
`reduce`	누적 계산	Optional or T	합계, 최대, 곱
`count`	개수 계산	long	`.count()`

🧪 실전 예제: 조합

```
List<String> names = List.of("Kim", "Lee", "Park", "Kim", "Choi");

long uniqueK = names.stream()
    .filter(n -> n.startsWith("K"))
    .distinct()
    .count();

System.out.println("K로 시작하는 고유 이름 개수: " + uniqueK);
// 출력: K로 시작하는 고유 이름 개수: 1
```

병렬 스트림 (`parallelStream()`)

✅ 병렬 스트림 (`parallelStream()`) 이란?

- 일반 스트림(`stream()`)은 **단일 스레드**로 순차적으로 처리됨
- `parallelStream()`은 내부적으로 **ForkJoinPool의 공용 스레드풀**을 활용해서 병렬 처리함
- 연산의 병렬성을 **개발자가 직접 구현하지 않아도** 자동으로 병렬화됨

🔹 사용법 비교

```
List<Integer> list = IntStream.rangeClosed(1, 10_000_000)
    .boxed()
    .collect(Collectors.toList());

// 순차 처리
long time1 = System.currentTimeMillis();
int sum1 = list.stream().mapToInt(i -> i).sum();
System.out.println("순차 처리: " + (System.currentTimeMillis() - time1) + "ms");

// 병렬 처리
long time2 = System.currentTimeMillis();
int sum2 = list.parallelStream().mapToInt(i -> i).sum();
System.out.println("병렬 처리: " + (System.currentTimeMillis() - time2) + "ms");
```

✅ 특징 및 내부 동작

항목	설명
내부 스레드풀	`ForkJoinPool.commonPool()` 사용 (기본적으로 CPU 코어 수 활용)
스레드 관리	개발자가 직접 스레드를 다루지 않음
동작 방식	데이터를 분할 → 병렬 처리 → 결과 병합
단점	스레드간 컨텍스트 스위칭 비용 발생, 결과 순서 보장 안 됨
주의	병렬화에 오히려 시간이 더 걸릴 수도 있음 (작은 데이터, 사이드 이펙트 있을 때)

⚠️ 언제 `parallelStream()`을 쓰면 안 되는가?

- **입출력(IO)** 을 포함하는 연산 (디스크, 네트워크 등): 병렬성이 효과 없음
- **사이드 이펙트**가 있는 연산: `forEach`에서 파일 저장, 콘솔 출력 등
- **순서가 중요한 연산**: `.forEachOrdered()` 대신 `.forEach()`는 순서 보장 X
- **작은 데이터셋**: 병렬 처리 오버헤드가 더 클 수 있음

✅ 예제: 성능 차이 체감

```
List<Integer> big = IntStream.rangeClosed(1, 100_000_000)
    .boxed()
    .collect(Collectors.toList());

long seq = System.currentTimeMillis();
long count1 = big.stream().filter(n -> n % 2 == 0).count();
System.out.println("순차: " + (System.currentTimeMillis() - seq) + "ms");

long par = System.currentTimeMillis();
long count2 = big.parallelStream().filter(n -> n % 2 == 0).count();
System.out.println("병렬: " + (System.currentTimeMillis() - par) + "ms");
```

> 💡 실행 환경에 따라 **병렬 스트림이 훨씬 빠르거나 오히려 느릴 수 있음** → 실험 필요

✅ `.parallel()` vs `.parallelStream()`

```
stream().parallel();        // 기존 스트림을 병렬화
parallelStream();           // 처음부터 병렬 스트림 생성
```

```
list.stream().parallel()
    .map(...)
    .forEach(...);
```

→ **동일한 효과**, 단 `stream()`으로 시작할 경우 병렬 여부를 나중에 제어할 수 있음

🧠 요약 정리표

메서드	설명
`parallelStream()`	컬렉션 기반 병렬 스트림 시작
`stream().parallel()`	기존 순차 스트림을 병렬로 전환
`forEach()` vs `forEachOrdered()`	병렬에서는 `forEachOrdered()`만 순서 보장
병렬처리 대상	CPU 바운드 작업(계산 중심)에 효과적
비추천 사례	I/O 작업, 작은 컬렉션, 순서 민감 연산, side-effect 포함

🚧 고급 활용: 병렬 스트림 커스터마이징

```
ForkJoinPool customPool = new ForkJoinPool(4);
customPool.submit(() -> {
    list.parallelStream().forEach(System.out::println);
}).get();
```

→ `ForkJoinPool.commonPool()` 대신 **사용자 정의 스레드풀**을 적용할 수 있음 (주의: 동기화 문제 발생 가능)