스트림 API - 기능

소유자	종수 김
태그	

스트림 생성

스트림은 자바 8부터 추가된 기능, 데이터 처리에 있어서 간결하고 효율적인 코드 작성을 가능케함.

중간 연산, 최종 연산을 구분하며, 지연 연산을 통해 불필요한 연산을 최소화.

내부적으로 파이프라인 형태를 만들어 데이터를 단계별로 처리하고, 결과를 반환.

스트림 생성 정리표

생성 방법	코드 예시	특징
컬렉션	`list.stream()`	`List`, `Set` 등 컬렉션에서 스트림 생성
배열	`Arrays.stream(arr)`	배열에서 스트림 생성
Stream.of(...)	`Stream.of("a", "b", "c")`	직접 요소를 입력해 스트림 생성
무한 스트림(iterate)	`Stream.iterate(0, n -> n + 2)`	무한 스트림 생성 (초깃값 + 함수)

무한 스트림(generate)	`Stream.generate(Math::random)`	무한 스트림 생성 (Supplier 사용)

```
package stream.operation;

import java.util.List;
import java.util.stream.Stream;

public class CreateStreamMain {
    public static void main(String[] args) {
        System.out.println("1. 컬렉션으로부터 생성");
        List<String> list = List.of("a", "b", "c", "d", "e", "f", "g");
        Stream<String> stream = list.stream();
        stream.forEach(System.out::println);

        System.out.println("2. 배열로부터 생성");
        String[] arr = {"a", "b", "c", "d", "e", "f", "g"};
```

```
        Stream<String> stream2 = Stream.of(arr);
        stream2.forEach(System.out::println);

        System.out.println("3. Stream.of() 사용");
        Stream<String> stream3 = Stream.of("a", "b", "c", "d", "e", "f", "g");
        stream3.forEach(System.out::println);

        System.out.println("4. 무한 스트림 생성 - iterate()");
        // iterate : 초기 값과 다음 값을 만드는 함수를 지정
        // seed : 초기 값 UnaryOperator : 인자와, 리턴 값이 같은 람다
        Stream<Integer> infiniteStream = Stream.iterate(0, n -> n + 2);
//        infiniteStream.forEach(System.out::println); // 무한대로 계속 출력
        infiniteStream.limit(5).forEach(System.out::println);

        System.out.println("5. 무한 스트림 생성 - generate");
        // generate : Supplier 사용하여 무한하게 생성
        Stream<Double> randomStream = Stream.generate(Math::random);
        randomStream.limit(5).forEach(System.out::println);
    }
}
```

정리

- 컬렉션, 배열, Stream.of = 기본적으로 유한한 데이터 소스로부터 스트림을 생성
- iterate, generate = 별도의 종료 조건이 없으면 무한히 데이터를 만들어내는 스트림 생성
    - limit을 통해 필요한 만큼만 사용.
- 스트림은 일반적으로 한 번 사용하면 재사용 할 수 없음.

중간 연산

중간 연산(Intermediate Operation)이란, 스트림 파이프라인에서 데이터를 변환, 필터링, 정렬 등을 하는 단계

- 여러 중간 연산을 연결하여 원하는 형태로 데이터로 가공
- 결과가 즉시 생성되지 않고, 최종 연산이 호출될 때 한꺼번에 처리 (= 지연 연산)

중간 연산 정리표

연산	설명	예시
filter	조건에 맞는 요소만 남김	`stream.filter(n -> n > 5)`
map	요소를 다른 형태로 변환	`stream.map(n -> n * 2)`
flatMap	중첩 구조 스트림을 일차원으로 평탄화	`stream.flatMap(list -> list.stream())`
distinct	중복 요소 제거	`stream.distinct()`
sorted	요소 정렬	`stream.sorted()` / `stream.sorted(Comparator.reverseOrder())`
peek	중간 처리 (로그, 디버깅)	`stream.peek(System.out::println)`
limit	앞에서 N개의 요소만 추출	`stream.limit(5)`
skip	앞에서 N개의 요소를 건너뛰고 이후 요소만 추출	`stream.skip(5)`
takeWhile	조건을 만족하는 동안 요소 추출 (Java 9+)	`stream.takeWhile(n -> n < 5)`
dropWhile	조건을 만족하는 동안 요소를 버리고 이후 요소 추출 (Java 9+)	`stream.dropWhile(n -> n < 5)`

```
package stream.operation;

import java.util.Collections;
import java.util.List;

public class IntermediateOperationsMain {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(1, 2, 2, 3, 4, 5, 5, 6, 7, 8, 9, 10);

        // 1. filter
        System.out.println("1. filter - 짝수만 선택");
        numbers.stream()
```

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

// 2. map
System.out.println("2. map - 각 숫자를 제곱");
numbers.stream()
        .map(n -> n * n)
        .forEach(System.out::println);

// 3. distinct
System.out.println("3. distinct - 중복 제거");
numbers.stream()
        .distinct()
        .forEach(System.out::println);

// 4. sorted (기본 정렬)
System.out.println("4. 정렬");
numbers.stream()
        .sorted((o1, o2) -> o2 - o1)
        .forEach(System.out::println);

// 5. peek - 중간 연산 사이에 peek을 넣음으로써 값의 변화를 확인이 가능
System.out.println("5. peek - 동작 확인용");
numbers.stream()
        .peek(n -> System.out.print("before: " + n + ", "))
        .map(n -> n * n)
        .peek(n -> System.out.print("after: " + n + ", "))
        .limit(5)
        .forEach(n -> System.out.println("최종 값: " + n + ", "));
System.out.println();

// 6. limit
System.out.println("6. limit - 처음 몇 개 요소만");
numbers.stream()
        .limit(5)
        .forEach(System.out::println);

// 7. skip
```

```
        System.out.println("7. skip - 처음 몇 개 요소를 건너뜀");
        numbers.stream()
                .skip(5)
                .forEach(System.out::println);

        List<Integer> numbers2 = List.of(1, 2, 3, 4, 5, 1, 2, 3, 6);

        //8. takeWhile (Java 9+)
        System.out.println("8. takeWhile - 5보다 작은 동안만 선택 - 처음 해당 값을 만
        numbers2.stream()
                .takeWhile(n → n < 5)
                .forEach(System.out::println);

        //9. dropWhile (Java 9+)
        System.out.println("9. dropWhile - 5보다 작은 동안 건너뛰기 - 그 뒤 수가 무한
        numbers2.stream()
                .dropWhile(n → n < 5)
                .forEach(System.out::println);
    }
}
```

- 중간 연산은 파이프라인 형태로 연결할 수 있으며, 스트림을 변경하지만, 원본 데이터 자체를 바꾸지 않음.
- 중간 연산은 lazy하게 동작하므로, 최종 연산이 실행될 때 까지 실제 처리는 일어나지 않음
- peek은 디버깅 목적으로 주로 사용. 실제 스트림의 요소값을 변경하거나, 연산 결과를 반환하지 않음.
- takeWhile, dropWhile은 자바 9부터 추가된 기능, 정렬된 스트림에 유용

FlatMap

중간 연산중 하나, map은 각 요소를 하나의 값으로 변환하지만, flatMap은 각 요소를 스트림(또는 여러 요소)으로 변환한 뒤, 그 결과를 하나의 스트림으로 평탄화(flattern)

이렇게 리스트 안에 리스트가 있다고 가정하자.

```
[
  [1, 2],
  [3, 4],
  [5, 6]
]
```

`FlatMap`을 사용하면 이 데이터를 다음과 같이 쉽게 평탄화(flatten)할 수 있다.

```
[1, 2, 3, 4, 5, 6]
```

```
package stream.operation;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

public class MapVsFlatMapMain {
    public static void main(String[] args) {
        List<List<Integer>> outerList = List.of(
                List.of(1, 2),
                List.of(3, 4),
                List.of(5, 6)
        );
        System.out.println(outerList);

        // 일반적인 평탄화
        List<Integer> forResult = new ArrayList<>();
        for (List<Integer> list : outerList) {
            for (int x : list) {
                forResult.add(x);
            }
        }
        System.out.println(forResult);

        // map으로 가능?
```

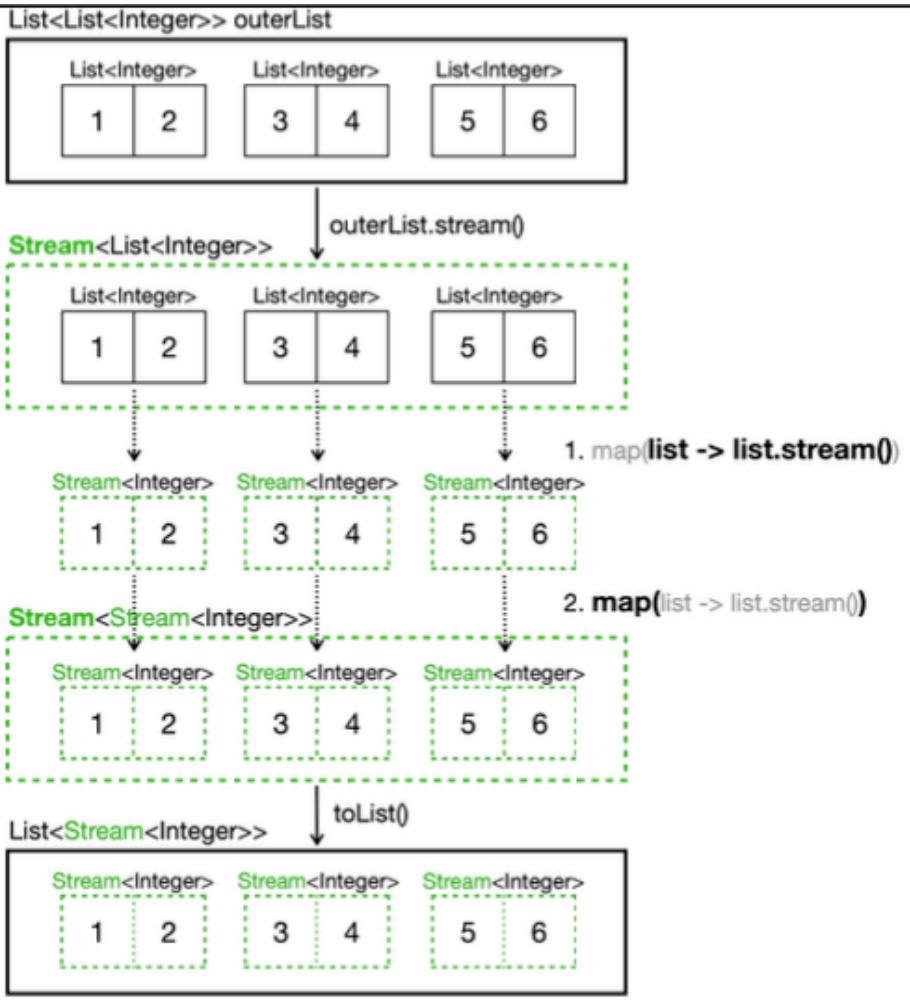
```
        List<List<Integer>> list1 = outerList.stream()
                .map(list -> list.stream()
                        .map(i -> i * 2)
                        .toList())
                .toList();
        System.out.println(list1);

        // flatMap
        List<Integer> list2 = outerList.stream()
                .flatMap(list -> list.stream()
                        .map(i -> i * 2))
                .toList();
        System.out.println(list2);
    }
}


[[1, 2], [3, 4], [5, 6]]
[1, 2, 3, 4, 5, 6]
[[2, 4], [6, 8], [10, 12]]
[2, 4, 6, 8, 10, 12]
```

- map을 쓰면, 이중 구조가 그대로 유지. 즉, 각 요소가 Stream 형태가 되므로 결과가 List<Stream<Integer>>
- mapResult는 Stream 객체 참조값을 출력하므로 [java.util.stream.ReferencePipeline] 형태로 보임.
- flatMap을 쓰면 내부의 Stream들을 하나로 합쳐 List<Integer>를 얻을 수 있음.

Map



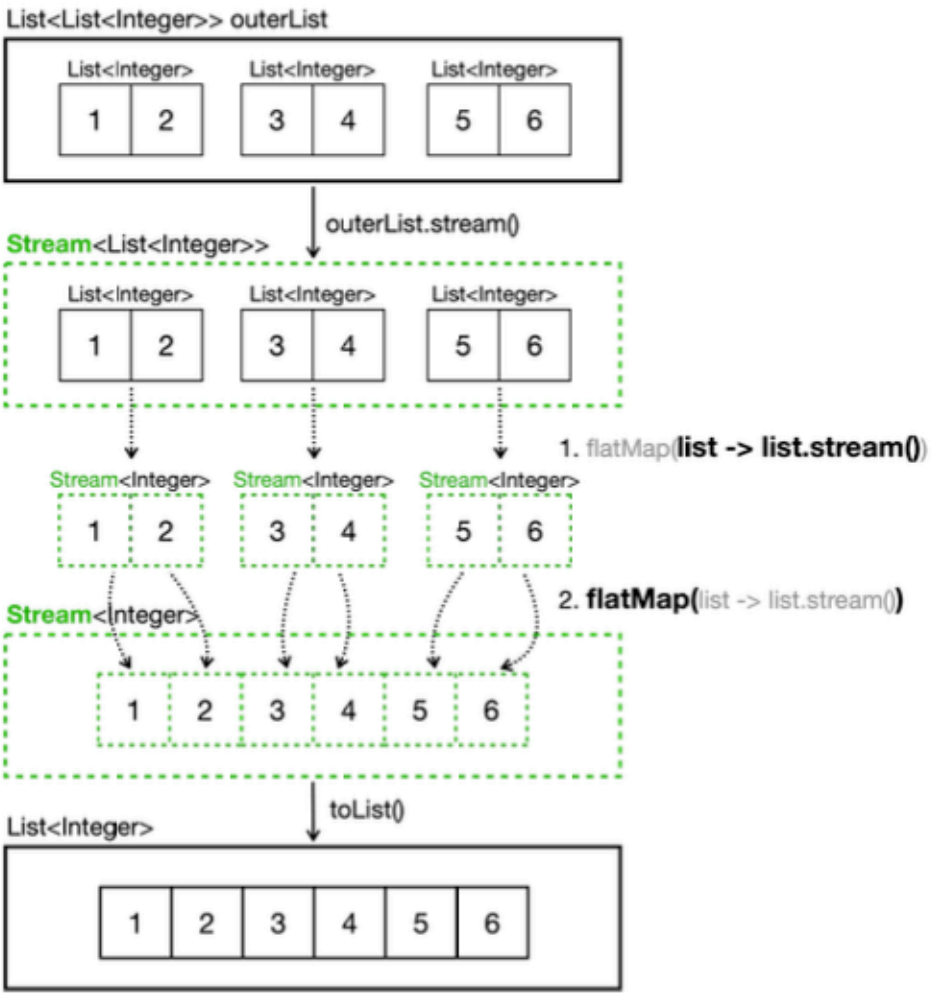
- `outerList.stream()`
  - `List<List<Integer>>` ➜ `Stream<List<Integer>>`
  - `stream()` 을 호출하면 `List<List<Integer>>` 에서 밖에 있는 `List` 가 `Stream` 으로 변한다.
  - 스트림 내부에는 3개의 `List<Integer>` 요소가 존재한다.
- `map(list -> list.stream())`
  - `Stream<List<Integer>>` ➜ `Stream<Stream<Integer>>`
  - `map()` 을 호출하면 `list -> list.stream()` 이 호출되면서 내부에 있는 3개의 `List<Integer>` 를 `Stream<Integer>` 로 변환한다.
  - 변환된 3개의 `Stream<Integer>` 가 외부 `Stream` 에 포함된다. 따라서 `Stream<Stream<Integer>>` 가 된다.
- `toList()`
  - `Stream<Stream<Integer>>` ➜ `List<Stream<Integer>>`
  - `toList()` 는 스트림을 리스트로 변환하는 기능이다.
  - `Stream<Stream<Integer>>` 는 내부 요소로 `Stream<Integer>` 를 3개 가진다.
  - `toList()` 는 외부 스트림에 대해서 실행한 것이다.
  - 따라서 내부 요소로 `Stream<Integer>` 를 3개 가지는 `List<Stream<Integer>>` 로 변환된다.

결과적으로 `List<List<Integer>>` ➜ `List<Stream<Integer>>` 가 되었다. 이것은 우리가 기대한 결과가 아니다.

- map()을 호출하면 list.stream()이 호출됨.

flamtMap

중첩 컬렉션을 다룰 때는 map 대신에, flatMap을 사용하면, 중첩 컬렉션을 편리하게 하나의 컬렉션으로 변환할 수 있음



- `outerList.stream()`
  - `List<List<Integer>>` ➜ `Stream<List<Integer>>`
  - `stream()` 을 호출하면 `List<List<Integer>>` 에서 밖에 있는 `List` 가 `Stream` 으로 변한다.
  - 스트림 내부에는 3개의 `List<Integer>` 요소가 존재한다.
- `flatMap(list -> list.stream())`
  - `Stream<List<Integer>>` ➜ `Stream<Integer>`
  - `flatMap()` 을 호출하면 `list -> list.stream()` 이 호출되면서 내부에 있는 3개의 `List<Integer>` 를 `Stream<Integer>` 로 변환한다.
  - `flatMap()` 은 `Stream<Integer>` **내부의 값을 꺼내서** 외부 `Stream` 에 포함한다. 여기서는 1,2,3,4,5,6의 값을 꺼낸다.
  - 이렇게 꺼낸 1,2,3,4,5,6 값 각각이 외부 `Stream` 에 포함된다. 따라서 `Stream<Integer>` 가 된다.
- `toList()`
  - `Stream<Integer>` ➜ `List<Integer>`
  - `toList()` 는 스트림을 리스트로 변환하는 기능이다.
  - `Stream<Integer>` 는 내부 요소로 `Integer` 를 6개 가진다.
  - 따라서 `Stream<Integer>` 는 `List<Integer>` 로 변환된다.

- flatMap(list → list.stream())
  - Stream<List<Integer>> → Stream<Integer>
  - flatMap()을 호출하면 list → list.stream()이 호출되면서 내부에 있는 3개의 List<Integer>를 Stream<Integer>로 변환.
  - flatMap()은 Stream<Integer> 내부의 값을 꺼내서, 외부 Stream에 포함.
- flatMap은 중첩 구조(컬렉션 안의 컬렉션, 배열 안의 배열 등)를 일차원으로 펼치는데 사용.
- Ex) 문자열 리스트들이 들어있는 리스트를 평탄화하면, 하나의 연속된 문자열 리스트로 만들 수 있음.

중첩 리스트 구조에서 평탄화

```
List<List<List<String>>> company = List.of(
            // 회사 A
            List.of(
                  List.of("Alice", "Bob"),      // 부서 A-1
                  List.of("Charlie", "Diana")    // 부서 A-2
            ),
            // 회사 B
            List.of(
                  List.of("Eve", "Frank"),      // 부서 B-1
                  List.of("Jerry", "Tom")       // 부서 B-2
            )
      );
      // 회사 A의 부서 A-2의 2번째 직원 Diana
      System.out.println(company.get(0).get(1).get(1));

      //flatMap 으로 표현
      // 부서를 없애고, 회사로 표현
      List<List<String>> company2 = company.stream()
            .flatMap(list → list.stream())
            .toList();
      System.out.println(company2);

            // 두 번 flatMap으로 계층을 단계적으로 평탄화
            // 3중 구조를 2중 → 1중으로 "하나씩" 풀어냄
```

```
        // 각각의 flatMap이 스트림을 반환하므로 올바르게 작동
    List<String> company3 = company.stream()
        .flatMap(list -> list.stream())
        .flatMap(dept -> dept.stream())
        .map(String::toUpperCase)
        .toList();
    System.out.println(company3);

        // 내부에서 2단계 평탄화를 먼저 하고, 그 결과 스트림을 외부 flatMap에서 평
        // 즉, 내부에서 바로 Stream<String>을 만든 뒤 한 번의 flatMap으로 처리
    List<String> company4 = company.stream()
        .flatMap(list -> list.stream()
            .flatMap(dept -> dept.stream()))
        .filter(name -> name.startsWith("A"))
        .toList();
    System.out.println(company4);
```

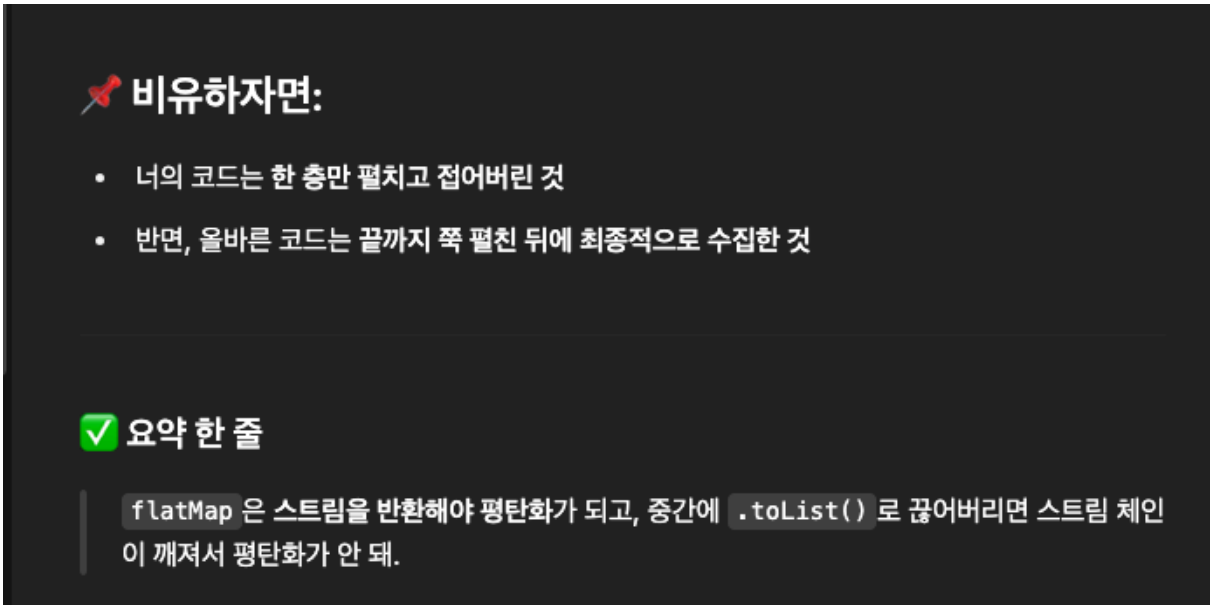
✅ 둘의 차이

방식	단계적 처리 (company3)	내부 일괄 처리 (company4)
flatMap 횟수	2회	1회 (중첩됨)
처리 순서	외부 → 중간 → 내부	내부 먼저 처리 후 외부
결과	동일 (List<String>)	동일 (List<String>)
성능	차이 없음 (스트림 체인이 같음)	차이 없음

```
# 컴파일 에러
List<String> company3 = company.stream()
        .flatMap(list -> list.stream()
            .flatMap(listN -> listN.stream())
            .toList())
        .toList();
```

스트림 내부에서 평탄화된 스트림을 toList()로 수집해버려서, 그 안에서 스트림을 "종료"

flatMap(...)은 Stream<T> → Stream<R> 을 기대함
그런데 너의 코드에서는 Stream<T> → List<R> 가 되어버림
결과적으로 flatMap이 아닌 **map + toList()**처럼 작동하고, 평탄화 실패함



최종 연산

최종 연산(Terminal Operation)은 스트림 파이프라인의 끝에 호출되어 실제 연산을 수행하고 결과를 만들어냄.
최종 연산이 실행 된 후 스트림은 소모되어 더 이상 사용 불가능

▼ 최종 연산 예제

최종 연산 정리표

연산	설명	예시
collect	Collector를 사용하여 결과 수집 (다양한 형태로 변환 가능)	`stream.collect(Collectors.toList())`
toList (Java16+)	스트림을 불변 리스트로 수집	`stream.toList()`

---

toArray	스트림을 배열로 변환	`stream.toArray(Integer[]::new)`
forEach	각 요소에 대해 동작 수행 (반환값 없음)	stream.forEach (System.out::println)
count	요소 개수 반환	`long count = stream.count();`
reduce	누적 함수를 사용해 모든 요소를 단일 결과로 합침 초깃값이 없으면 Optional로 반환	`int sum = stream.reduce (0, Integer::sum);`
min / max	최솟값, 최댓값을 Optional로 반환	`stream.min(Integer::compareTo)`, `stream.max(Integer::compareTo)`
findFirst	조건에 맞는 첫 번째 요소 (Optional 반환)	`stream.findFirst()`
findAny	조건에 맞는 아무 요소나 (Optional 반환)	`stream.findAny()`
anyMatch	하나라도 조건을 만족하는지 (boolean)	`stream.anyMatch(n -> n > 5)`
allMatch	모두 조건을 만족하는지 (boolean)	`stream.allMatch(n -> n > 0)`
noneMatch	하나도 조건을 만족하지 않는지 (boolean)	`stream.noneMatch(n -> n < 0)`

```
package stream.operation;

import java.util.Arrays;
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;

public class TerminalOperationMain {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(1, 2, 2, 3, 4, 5, 5, 6, 7, 8, 9, 10);

        // Collectors
        System.out.println("1. collect - List 수집");
        List<Integer> evenNumbers1 = numbers.stream()
                .filter(n -> n % 2 == 0)
                .collect(Collectors.toList());
        System.out.println("짝수 리스트 : " + evenNumbers1);

        // Collectors - 편의 기능
        System.out.println("2. collect - List 수집 - toList()");
        List<Integer> evenNumbers2 = numbers.stream()
                .filter(n -> n % 2 == 0)
                .toList();
        System.out.println("짝수 리스트 : " + evenNumbers2);

        System.out.println("3. toArray - 배열로 변환");
        Integer[] evenNumber3 = numbers.stream()
                .filter(n -> n % 2 == 0)
                .toArray(Integer[]::new);
        System.out.println("짝수 배열 : " + Arrays.toString(evenNumber3));

        System.out.println("4. forEach - 각 요소 처리");
        numbers.stream()
                .limit(5)
                .forEach(System.out::print);
        System.out.println();

        System.out.println("5. count - 요소 개수");
```

```
long count = numbers.stream()
        .filter(n → n > 5)
        .count();
System.out.println(count);

// 빈 numbers 라면 값이 없을 수도 있기에, Optional 로 제공
System.out.println("6. reduce - 요소들의 합");
System.out.println("초기 값이 없는 reduce");
Optional<Integer> sum1 = numbers.stream()
        .reduce((a, b) → a + b);
System.out.println("합계(초기 값 없음)  " + sum1.get());

// 초기 값이 존재한다는건 numbers가 비어있더라도 최소 값이 보장 때문에 int
System.out.println("6. reduce - 요소들의 합");
System.out.println("초기 값이 있는 reduce");
int sum2 = numbers.stream()
        .reduce(100, (a, b) → a + b);
System.out.println("합계(초기 값 있음)  " + sum2);

System.out.println("7. min - 최솟값");
Optional<Integer> min = numbers.stream()
        .min(Integer::compareTo);
System.out.println("최솟값 : " + min.get());

System.out.println("8. max - 최댓값");
Optional<Integer> max = numbers.stream()
        .max(Integer::compareTo);
System.out.println("최댓값 : " + max.get());

System.out.println("9. findFirst - 첫 번째 요소");
Integer first = numbers.stream()
        .filter(n → n > 5)
        .findFirst()
        .get();
System.out.println("5보다 큰 첫 번째 숫자 : " + first);

// 멀티 쓰레드 상황에서, 주로 사용 (어떤 쓰레드던 하나 걸리면 가져옴, 그 외 상
System.out.println("10. findAny - 아무 요소나 하나");
```

```
        Integer any = numbers.stream()
                .filter(n -> n > 5)
                .findAny()
                .get();
        System.out.println("5보다 큰 아무 요소 중 하나 : " +any);

        System.out.println("11. anyMatch - 조건을 만족하는 요소 존재 여부");
        boolean hasEven = numbers.stream()
                .anyMatch(n -> n % 2 == 0);
        System.out.println("짝수가 있는가 : " + hasEven);

        System.out.println("12. allMatch - 모든 요소가 조건을 만족하는가");
        boolean allPositive = numbers.stream()
                .allMatch(n -> n > 0);
        System.out.println("모든 숫자가 양수인가 " + allPositive);

        System.out.println("13. noneMatch - 하나도 조건을 만족하지 않는지");
        boolean noNegative = numbers.stream()
                .noneMatch(n -> n < 0);
        System.out.println("음수가 없는가 : " + noNegative);
    }
}
```

1. collect

- Collectors를 사용해 다양한 형태로 결과를 **수집(collect)** 한다.
- 예: `Collectors.toList()`, `Collectors.toSet()`, `Collectors.joining()` 등 다양한 `Collector`를 제공
- 이 부분은 뒤에서 자세히 다룬다.

2. toList (Java 16+)

- `Collectors.toList()` 대신, 바로 `stream.toList()`를 써서 간단하게 **List**로 변환한다.

3. toArray

- 스트림을 **배열**로 변환한다.

4. forEach

- 각 요소에 대해 **순차적으로** 작업을 수행한다. (반환값 없음)

5. count

- 스트림의 요소 개수를 반환한다.

6. reduce

- 요소들을 **하나의 값**으로 누적한다. (합계, 곱, 최솟값, 최댓값 등)
- 초깃값을 주는 형태, 주지 않는 형태 두 가지가 있다. 초깃값이 없는 경우 `Optional`을 반환한다.

7. min, 8. max

- 최솟값, 최댓값을 구한다.

- 결과는 `Optional`로 감싸져 있으므로, `get()` 메서드나 `ifPresent()` 등을 사용해 값을 가져온다.

9. findFirst, 10. findAny

- findFirst: 조건에 맞는 첫 번째 요소, findAny: 조건에 맞는 아무 요소나 반환(순서와 관계 없음)
- 병렬 스트림인 경우 `findAny`는 더욱 효율적으로 동작할 수 있다.

11. anyMatch, 12. allMatch, 13. noneMatch

- 스트림 요소 중 조건을 **하나라도 만족**하는지(anyMatch),
- 스트림 **모두가 만족**하는지(allMatch),
- **아무도 만족하지 않는지**(noneMatch)를 **boolean**으로 반환한다.

정리

- 최종 연산이 호출되면, 그 동안 정의된 모든 중간 연산이 한 번에 적용되어 결과를 만듦.
- 최종 연산을 한번 수행 시, 스트림은 재사용 불가능

- Stream의 요소가 없는 경우를 대비해 Optional 메서드로 제공하는 함수가 있음.

스트림 생성, 중간 연산, 최종 연산을 통해 데이터 처리 연산을 간결하게 표현 가능.

기본형 특화 스트림

기본형(primitive) 특화 스트림이 존재.

IntStream, LongStream, DoubleStream 세 가지 형태를 제공하여 기본 자료형(int, long, double)에 특화된 기능을 사용.

- IntStream을 통해 정수와 관련된 연산을 좀 더 편리하게 제공.
- 오토 박싱 / 언박싱 비용을 줄여 성능 향상

스트림 타입	대상 원시 타입	생성 예시
IntStream	int	`IntStream.of(1, 2, 3)`, `IntStream.range(1, 10)`, `mapToInt(...)`
LongStream	long	`LongStream.of(10L, 20L)`, `LongStream.range(1, 10)`, `mapToLong(...)`
DoubleStream	double	`DoubleStream.of(3.14, 2.78)`, `DoubleStream.generate(Math::random)`, `mapToDouble(...)`

기본형 특화 스트림의 숫자 범위 생성 기능

- `range(int startInclusive, int endExclusive)` : 시작값 이상, 끝값 미만
  - `IntStream.range(1, 5)` → `[1, 2, 3, 4]`
- `rangeClosed(int startInclusive, int endInclusive)` : 시작값 이상, 끝값 **포함**
  - `IntStream.rangeClosed(1, 5)` → `[1, 2, 3, 4, 5]`

주요 기능 및 메서드

합계, 평균 등 자주 사용하는 연산을 편리한 메서드로 제공.

타입 변환과 박싱 / 언박싱 메서드도 제공하여 다른 스트림과 연계해 작업

int, long, double 같은 숫자를 사용

▼ 예제

메서드 / 기능	설명	예시
sum()	모든 요소의 **합계**를 구한다.	int total = IntStream.of(1, 2, 3).sum();
average()	모든 요소의 **평균**을 구한다. OptionalDouble을 반환.	double avg = IntStream .range(1, 5).average().getAsDouble();
summary Statistics()	최솟값, 최댓값, 합계, 개수, 평균 등이 담긴 IntSummaryStatistics (또는 Long/Double) 객체 반환	IntSummaryStatistics stats = IntStream.range(1,5).summaryStatistics();

---

mapToLong() mapToDouble()	**타입 변환** : IntStream → LongStream, DoubleStream ...	LongStream ls = IntStream.of(1,2).mapToLong(i ➔ i * 10L);
mapToObj()	**객체 스트림**으로 변환 : 기본형 → 참조형	Stream<String> s = IntStream.range(1,5) .mapToObj(i ➔ "No: "+i);
boxed()	기본형 특화 스트림을 **박싱(Wrapper)된 객체 스트림**으로 변환	Stream<Integer> si = IntStream.range(1,5).boxed();
sum(), min(), max(), count()	합계, 최솟값, 최댓값, 개수를 반환 (타입별로 int/long/double 반환)	long cnt = LongStream.of(1,2,3).count();

```
package stream.operation;

import java.util.IntSummaryStatistics;
import java.util.stream.DoubleStream;
import java.util.stream.IntStream;
import java.util.stream.LongStream;
import java.util.stream.Stream;

public class PrimitiveStreamMain {
    public static void main(String[] args) {
        // 기본형 특화 스트림 생성(IntStream, LongStream, DoubleStream)
        IntStream stream = IntStream.of(1, 2, 3, 4, 5);
```

```
        stream.forEach(i -> System.out.print(i + " "));
        System.out.println();

        // 범위 생성 메서드 (IntStream, LongStream)
        IntStream range1 = IntStream.range(1, 6); // [1,2,3,4,5]
        range1.forEach(i -> System.out.print(i + " "));
        System.out.println();
        IntStream range2 = IntStream.rangeClosed(1, 6); // [1,2,3,4,5,6]
        range2.forEach(i -> System.out.print(i + " "));
        System.out.println();

        // 1. 통계 관련 메서드 (sum, average, max, min, count)
        int sum = IntStream.range(1, 6).sum();
        System.out.println("sum = " + sum);

        // average : 평균
        double average = IntStream.range(1, 6).average().getAsDouble();
        System.out.println("average = " + average);

        // summaryStatistics() : 모든 통계 정보
        IntSummaryStatistics stats = IntStream.range(1, 6).summaryStatistics
        System.out.println("합계 " + stats.getSum());
        System.out.println("평균 " + stats.getAverage());
        System.out.println("최대값 " + stats.getMin());
        System.out.println("최소값 " + stats.getMin());
        System.out.println("개수 " + stats.getCount());

        // 2. 타입 변환 메서드
        // IntStream -> LongStream
        LongStream longStream = IntStream.range(1, 5).asLongStream();
        DoubleStream doubleStream = IntStream.range(1, 5).asDoubleStream
        // IntStream -> Stream<Integer>
        Stream<Integer> boxed = IntStream.range(1, 5).boxed();

        // 3. 기본형 특화 매핑
        // int -> long
        LongStream longStream1 = IntStream.range(1, 5)
                .mapToLong(i -> i);
```

```
        // int → double
        DoubleStream doubleStream1 = IntStream.range(1, 5)
                .mapToDouble(i → i);
        // int → 객체 변환 매핑
        Stream<String> stringStream = IntStream.range(1, 5)
                .mapToObj(String::valueOf);

        // 4. 객체 스트림 -> 기본형 특화 스트림 매핑
        Stream<Integer> integerStream = Stream.of(1, 2, 3, 4, 5);
        // 기본형 특화 스트림의 함수를 사용하기 위해
        IntStream intStream = integerStream.mapToInt(i → i);

        // 5. 객체 스트림 -> 기본형 특화 스트림으로 매핑 활용
        int sum1 = Stream.of(1, 2, 3, 4, 5)
                .mapToInt(i → i)
                .sum();
        System.out.println("sum1 = " + sum1);

    }
}
```

- 기본형 특화 스트림을 이용하면 숫자 계산을 간편하게 처리하고, 박싱 / 언박싱 오버헤드를 줄여 성능상의 이점도 얻을 수 있음.
- range() , rangeClosed() 같은 메서드를 사용하면 범위를 쉽게 다룰 수 있어, 반복문 대신 자주 쓰임.
- mapToXxx, boxed() 등의 메서드를 잘 활용하여 객체 스트림과 기본형 특화 스트림을 자유롭게 사용 가능.
- summaryStatistics()를 이용하여 통계 정보를 한번에 구할 수 있음.

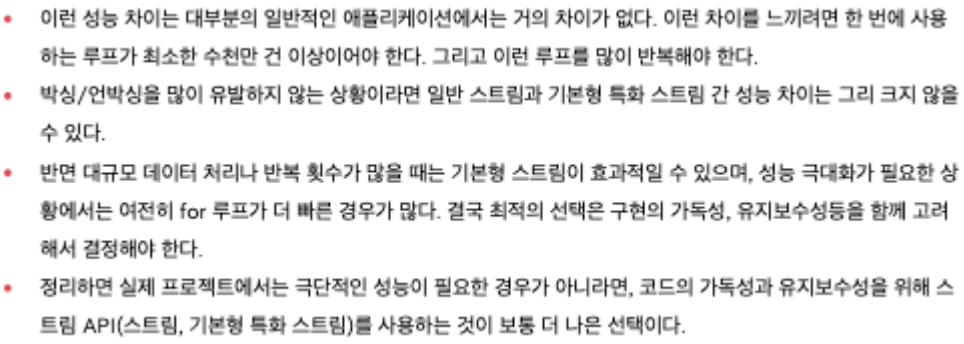
성능 - 전통적인 for문 vs 스트림 vs 기본형 특화 스트림

실제로 어느정도 차이가 날지는 데이터 양, 연산 종류, JVM 최적화 등에 따라 달라지니 참고만.

- 전통적인 for 문이 가장 빠름
- 스트림(Stream<Integer>)보다 전통적인 for 문이 1.5 ~ 2 배 빠름.
    - 박싱 / 언박싱 오버헤드가 발생
- 기본형 특화 스트림(IntStream)은 전통적인 for문에 가까운 성능

- 전통적인 for문과 거의 비슷하거나 차이가 크지 않음
- 박싱 / 언박싱 오버헤드를 피할 수 있음
- 내부적으로 최적화 된 연산을 수행

실무 선택



- 이런 성능 차이는 대부분의 일반적인 애플리케이션에서는 거의 차이가 없다. 이런 차이를 느끼려면 한 번에 사용하는 루프가 최소한 수천만 건 이상이어야 한다. 그리고 이런 루프를 많이 반복해야 한다.
- 박싱/언박싱을 많이 유발하지 않는 상황이라면 일반 스트림과 기본형 특화 스트림 간 성능 차이는 그리 크지 않을 수 있다.
- 반면 대규모 데이터 처리나 반복 횟수가 많을 때는 기본형 스트림이 효과적일 수 있으며, 성능 극대화가 필요한 상황에서는 여전히 for 루프가 더 빠른 경우가 많다. 결국 최적의 선택은 구현의 가독성, 유지보수성등을 함께 고려해서 결정해야 한다.
- 정리하면 실제 프로젝트에서는 극단적인 성능이 필요한 경우가 아니라면, 코드의 가독성과 유지보수성을 위해 스트림 API(스트림, 기본형 특화 스트림)를 사용하는 것이 보통 더 나은 선택이다.

- 실무에서 극단적인 성능이 필요한 경우가 아니라면, 코드의 가독성과 유지보수성을 위해 스트림을 사용하는게 보통 더 나음.
- 저 정도로 극단적인 성능이 필요한 경우는 많이 없음.