8. 스트림 API2 - 기능

#1.인강/0.자바/7.자바-고급3편

스트림 생성

스트림(Stream)은 자바 8부터 추가된 기능으로, 데이터 처리에 있어서 **간결하고 효율적인 코드 작성**을 가능하게 해준다. 스트림을 이용하면 컬렉션(List, Set 등)이나 배열에 저장된 요소들을 **반복문 없이도** 간단하게 필터링(filter), 변환(map), 정렬(sorted) 등의 작업을 적용할 수 있다.
특히 스트림은 **중간 연산**과 **최종 연산**을 구분하며, **지연 연산(lazy evaluation)**을 통해 불필요한 연산을 최소화한다.
자바 스트림은 내부적으로 **파이프라인** 형태를 만들어 데이터를 단계별로 처리하고, 결과를 효율적으로 반환한다.

스트림이 제공하는 다양한 **스트림 생성**, **중간 연산**, **최종 연산**을 자세히 알아보자.
먼저 스트림을 생성하는 다양한 방법부터 알아보자.

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 (초기값 + 함수)

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

스트림을 생성하는 대표적인 방법들을 코드로 알아보자.

```
package stream.operation;

import java.util.Arrays;
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");
        Stream<String> stream1 = list.stream();
        stream1.forEach(System.out::println);

        System.out.println("2. 배열로부터 생성");
        String[] arr = {"a", "b", "c"};
        Stream<String> stream2 = Arrays.stream(arr);
        stream2.forEach(System.out::println);

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

        System.out.println("4. 무한 스트림 생성 - iterate()");
        // iterate: 초기값과 다음 값을 만드는 함수를 지정
        Stream<Integer> infiniteStream = Stream.iterate(0, n -> n + 2); // 짝수
무한 스트림
        infiniteStream.limit(3)
                .forEach(System.out::println);

        System.out.println("5. 무한 스트림 생성 - generate()");
        // generate: Supplier를 사용하여 무한하게 생성
        Stream<Double> randomStream = Stream.generate(Math::random);
        randomStream.limit(3)
              .forEach(System.out::println);
```

```
    }
}
```

실행 결과

```
1. 컬렉션으로부터 생성
a
b
c
2. 배열로부터 생성
a
b
c
3. Stream.of() 사용
a
b
c
4. 무한 스트림 생성 - iterate()
0
2
4
5. 무한 스트림 생성 - generate()
0.5330114888402112
0.4593127342014255
0.6281913838258091
```

정리

- **컬렉션, 배열, Stream.of** 은 기본적으로 **유한한** 데이터 소스로부터 스트림을 생성한다.
- **iterate**, **generate**는 별도의 종료 조건이 없으면 **무한히 데이터를 만들어내는** 스트림을 생성한다.
  - 따라서 **필요한 만큼만(limit) 사용**해야 한다. 그렇지 않으면 무한 루프처럼 계속 스트림을 뽑아내므로 주의해야 한다.
- 스트림은 일반적으로 한 번 사용하면 재사용할 수 없다(소진되면 끝). 따라서, `stream()`으로 얻은 스트림을 여러 번 순회하려면, 다시 스트림을 생성해야 한다.

중간 연산

중간 연산(Intermediate Operation)이란, 스트림 파이프라인에서 **데이터를 변환, 필터링, 정렬 등**을 하는 단계이다.

- 여러 중간 연산을 **연결**하여 원하는 형태로 데이터를 가공할 수 있다.
- **결과가 즉시 생성되지 않고**, 최종 연산이 호출될 때 **한꺼번에** 처리된다는 특징이 있다(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)`

중간 연산을 코드로 알아보자.

```
package stream.operation;
```

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

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(n -> System.out.print(n + " "));
        System.out.println("\n");

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

        // 3. distinct
        System.out.println("3. distinct - 중복 제거");
        numbers.stream()
                .distinct()
                .forEach(n -> System.out.print(n + " "));
        System.out.println("\n");

        // 4. sorted (기본 정렬)
        System.out.println("4. sorted - 기본 정렬");
        Stream.of(3, 1, 4, 1, 5, 9, 2, 6, 5)
                .sorted()
                .forEach(n -> System.out.print(n + " "));
        System.out.println("\n");

        // 5. sorted (커스텀 정렬)
        System.out.println("5. sorted with Comparator - 내림차순 정렬");
        Stream.of(3, 1, 4, 1, 5, 9, 2, 6, 5)
                .sorted(Comparator.reverseOrder())
                .forEach(n -> System.out.print(n + " "));
        System.out.println("\n");
```

```
        // 6. peek
        System.out.println("6. 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("최종값: " + n));
        System.out.println();

        // 7. limit
        System.out.println("7. limit - 처음 5개 요소만");
        numbers.stream()
                .limit(5)
                .forEach(n -> System.out.print(n + " "));
        System.out.println("\n");

        // 8. skip
        System.out.println("8. skip - 처음 5개 요소를 건너뛰기");
        numbers.stream()
                .skip(5)
                .forEach(n -> System.out.print(n + " "));
        System.out.println("\n");

        List<Integer> numbers2 = List.of(1, 2, 3, 4, 5, 1, 2, 3);
        // 9. takeWhile (Java 9+)
        System.out.println("9. takeWhile - 5보다 작은 동안만 선택");
        numbers2.stream()
                .takeWhile(n -> n < 5)
                .forEach(n -> System.out.print(n + " "));
        System.out.println("\n");

        // 10. dropWhile (Java 9+)
        System.out.println("10. dropWhile - 5보다 작은 동안 건너뛰기");
        numbers2.stream()
                .dropWhile(n -> n < 5)
                .forEach(n -> System.out.print(n + " "));
    }
}
```

실행 결과

```
1. filter - 짝수만 선택
2 2 4 6 8 10

2. map - 각 숫자를 제곱
1 4 4 9 16 25 25 36 49 64 81 100

3. distinct - 중복 제거
1 2 3 4 5 6 7 8 9 10

4. sorted - 기본 정렬
1 1 2 3 4 5 5 6 9

5. sorted with Comparator - 내림차순 정렬
9 6 5 5 4 3 2 1 1

6. peek - 동작 확인용
before: 1, after: 1, 최종값: 1
before: 2, after: 4, 최종값: 4
before: 2, after: 4, 최종값: 4
before: 3, after: 9, 최종값: 9
before: 4, after: 16, 최종값: 16

7. limit - 처음 5개 요소만
1 2 2 3 4

8. skip - 처음 5개 요소를 건너뛰기
5 5 6 7 8 9 10

9. takeWhile - 5보다 작은 동안만 선택
1 2 3 4

10. dropWhile - 5보다 작은 동안 건너뛰기
5 1 2 3
```

1. filter

- 조건에 맞는 요소만 추려낸다.
- 예: `n -> n % 2 == 0`은 짝수만 필터링.

2. map

- 요소를 **다른 형태로 변환**한다.
- 예: `n -> n * n` 은 각 숫자를 제곱.

3. distinct

- 중복 요소를 제거한다.

4, 5. sorted

- 스트림의 요소를 정렬한다. (기본 정렬 혹은 커스텀 Comparator)

6. peek

- **중간 단계**에서 요소를 **엿보는(peek)** 용도로 사용한다.
- 참고로 **peek은 데이터를 변경하지 않는다.** 주로 디버깅이나 로깅 용도로 사용된다.

7. limit

- **앞에서부터 N개의 요소**만 추출한다.

8. skip

- **앞에서부터 N개의 요소**를 건너뛰고, 나머지 요소로 스트림을 구성한다.

9. takeWhile (Java 9+)

- **조건을 만족하는 동안** 요소를 가져온다. 조건이 처음으로 거짓이 되는 지점에서 스트림을 멈춘다.
  - 스트림이 중간에 멈추기 때문에 원하는 목적을 빨리 달성하면 성능을 최적화 할 수 있다.

10. dropWhile (Java 9+)

- **조건을 만족하는 동안** 요소를 버린다. 조건이 처음으로 거짓이 되는 지점부터 스트림을 구성한다.

중간 연산 정리

- 중간 연산은 **파이프라인** 형태로 연결할 수 있으며, **스트림을 변경**하지만 원본 데이터 자체를 바꾸지 않는다.
- 중간 연산은 **lazy**(지연, 게으르게)하게 동작하므로, **최종 연산**이 실행될 때까지 실제 처리는 일어나지 않는다.
- `peek`은 **디버깅 목적**으로 자주 사용하며, 실제 스트림의 요소값을 변경하거나 연산 결과를 반환하지는 않는다.
- `takeWhile`, `dropWhile`는 자바 9부터 추가된 기능으로, **정렬된 스트림**에서 사용할 때 유용하다.
  - 정렬되지 않은 스트림에서 쓰면 예측하기 어렵다.

FlatMap

중간 연산의 하나인 `FlatMap`에 대해서 알아보자.

`map`은 각 요소를 하나의 값으로 변환하지만, `flatMap`은 각 요소를 스트림(또는 여러 요소)으로 변환한 뒤, 그 결과를 하나의 스트림으로 평탄화(flatten)해준다.

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]
```

코드와 그림을 통해 자세히 알아보자.

```
package stream.operation;

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

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 = " + outerList);

        // for
        List<Integer> forResult = new ArrayList<>();
        for (List<Integer> list : outerList) {
            for (Integer i : list) {
                forResult.add(i);
            }
        }
```

```
        System.out.println("forResult = " + forResult);

        // map
        List<Stream<Integer>> mapResult = outerList.stream()
                .map(list -> list.stream())
                .toList();
        System.out.println("mapResult = " + mapResult);

        // flatMap
        List<Integer> flatMapResult = outerList.stream()
                .flatMap(list -> list.stream())
                .toList();
        System.out.println("flatMapResult = " + flatMapResult);
    }
}
```

- for 문으로 평탄화를 처리하는 경우 이중 for 문을 사용해야 한다.

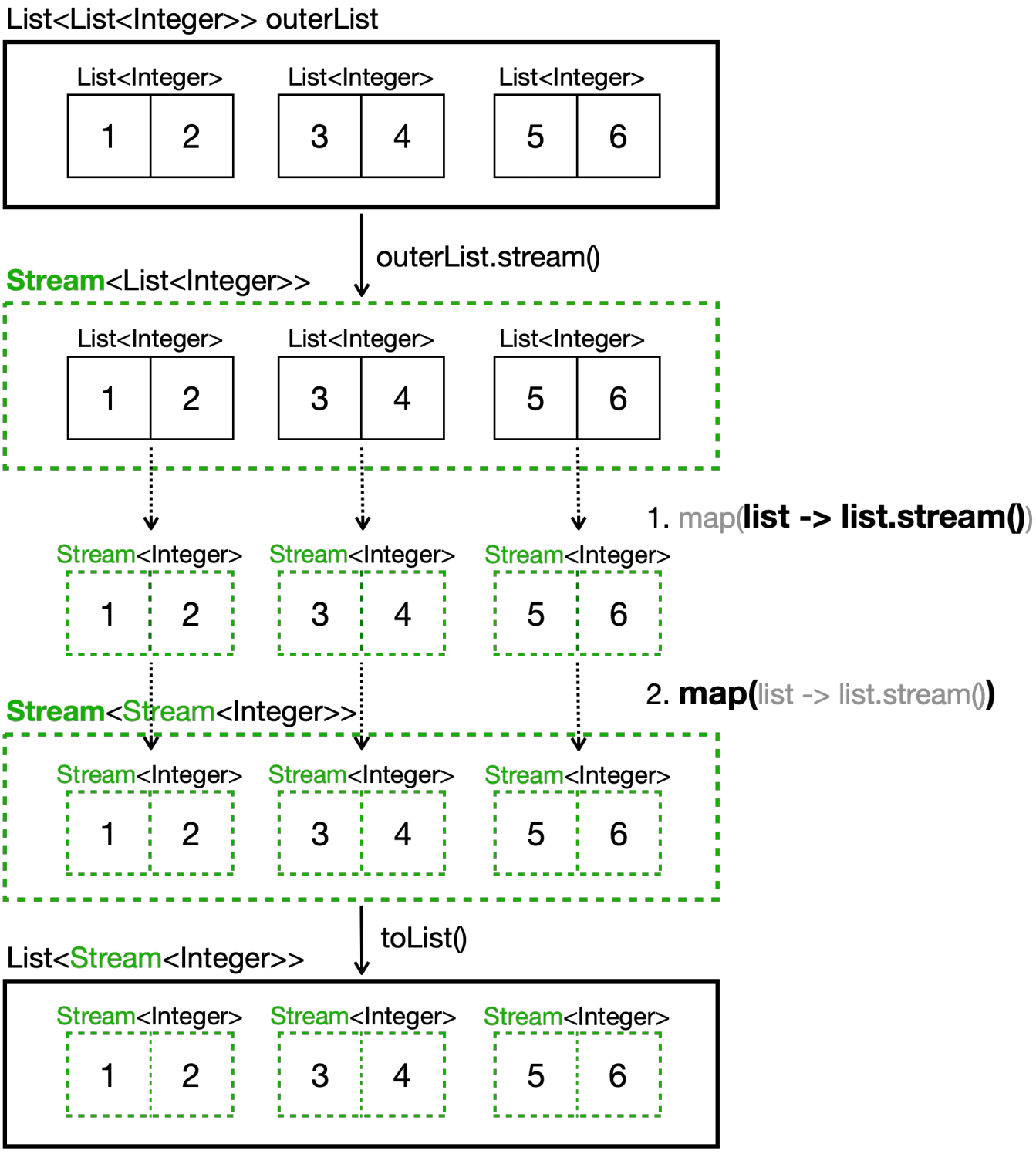
실행 결과

```
outerList = [[1, 2], [3, 4], [5, 6]]
forResult = [1, 2, 3, 4, 5, 6]
mapResult = [java.util.stream.ReferencePipeline$Head@3498ed,
java.util.stream.ReferencePipeline$Head@1a407d53,
java.util.stream.ReferencePipeline$Head@3d8c7aca]
flatMapResult = [1, 2, 3, 4, 5, 6]
```

- `map`을 쓰면 **이중 구조**가 그대로 유지된다. 즉, 각 요소가 `Stream` 형태가 되므로 결과가 `List<Stream<Integer>>`가 된다.
- `mapResult`는 Stream 객체 참조값을 출력하므로 `[java.util.stream.ReferencePipeline$Head@...]` 형태로 보인다.
- `flatMap`을 쓰면 내부의 `Stream`들을 **하나로 합쳐** `List<Integer>`를 얻을 수 있다.

그림으로 자세히 알아보자.

Map() 그림



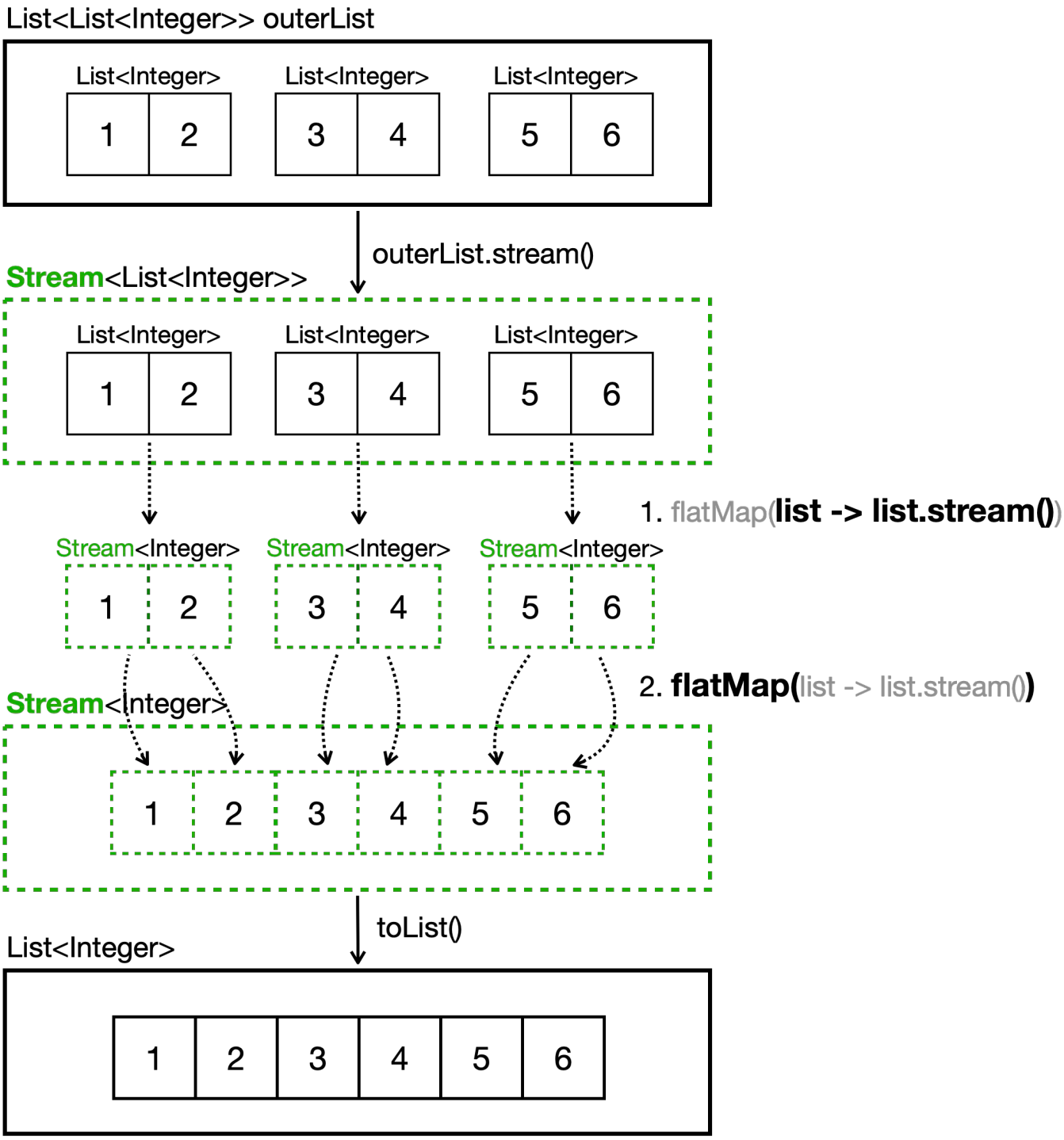
- `outerList.stream()`
  - `List<List<Integer>>` ➜ `Stream<List<Integer>>`
  - `stream()` 을 호출하면 `List<List<Integer>>` 에서 밖에 있는 `List` 가 `Stream` 으로 변한다.
  - 스트림 내부에는 3개의 `List<Integer>` 요소가 존재한다.
- `map(list -> list.stream())`
  - `Stream<List<Integer>>` ➜ `Stream<Stream<Integer>>`
  - `map()` 을 호출하면 `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.
중첩 컬렉션을 다룰 때는 `map` 대신에 `FlatMap`을 사용하면 중첩 컬렉션을 편리하게 하나의 컬렉션으로 변환할 수 있다.

FlatMap() 그림



- `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.

정리

- `flatMap`은 **중첩 구조**(컬렉션 안의 컬렉션, 배열 안의 배열 등)를 **일차원으로 펼치는 데** 사용된다.
- 예를 들어, 문자열 리스트들이 들어있는 리스트를 평탄화하면, 하나의 연속된 문자열 리스트로 만들 수 있다.

Optional 간단 설명

최종 연산을 시작하기 전에 잠깐 자바가 제공하는 `Optional`(옵셔널) 클래스를 간단히 알아보자.

참고: 옵셔널은 뒤에서 매우 자세히 다룬다. 여기서는 스트림의 최종 연산을 이해하는데 필요한 최소한의 옵셔널만 기능만 알아본다.

자바 Optional 클래스

```
package java.util;

public final class Optional<T> {

    private final T value;
    ...

    public boolean isPresent() {
        return value != null;
    }

    public T get() {
        if (value == null) {
            throw new NoSuchElementException("No value present");
        }
        return value;
```

```
    }
}
```

- `Optional`은 내부에 하나의 값(`value`)을 가진다.
- `isPresent()`를 통해 그 값(`value`)이 있는지 없는지 확인할 수 있다.
- `get()`을 통해 내부의 값을 꺼낼 수 있다. 만약 값이 없다면 예외가 발생한다.
- `Optional`은 이름 그대로 필수가 아니라 옵션이라는 뜻이다.
  - 이 말은 옵셔널 내부에 값(`value`)이 있을 수도 있고 없을 수도 있다는 뜻이다.

옵셔널을 어떻게 사용하는지 코드로 알아보자.

```
package stream.operation;

import java.util.Optional;

public class OptionalSimpleMain {

    public static void main(String[] args) {
        Optional<Integer> optional1 = Optional.of(10);
        System.out.println("optional1 = " + optional1);
        if (optional1.isPresent()) { // 값이 있는지 확인할 수 있는 안전한 메서드 제공
            Integer i = optional1.get(); // Optional 안에 있는 값을 획득
            System.out.println("i = " + i);
        }

        Optional<Object> optional2 = Optional.ofNullable(null);
        System.out.println("optional2 = " + optional2);
        if (optional2.isPresent()) {
            Object o = optional2.get();
            System.out.println("o = " + o);
        }

        // 값이 없는 Optional에서 get()을 호출하면 NoSuchElementException이 발생한다."
        Object o2 = optional2.get();
        System.out.println("o2 = " + o2);
    }
}
```

실행 결과

```
```

```
optional1 = Optional[10]
i = 10
optional2 = Optional.empty
Exception in thread "main" java.util.NoSuchElementException: No value present
    at java.base/java.util.Optional.get(Optional.java:143)
    at stream.operation.OptionalSimpleMain.main(OptionalSimpleMain.java:24)
...
```

- `Optional`은 내부에 값을 담아두고, 그 값이 `null`인지 아닌지를 체크할 수 있는 `isPresent()`와 같은 안전한 체크 메서드를 제공한다. 따라서 안전한 체크 메서드를 통해 체크하고 난 다음에 값이 있을 때만 `get()`으로 값을 꺼내는 방식으로 사용할 수 있다.
- `Optional`은 `null` 값으로 인한 오류(`NullPointerException`)를 방지하고, 코드에서 "값이 없을 수도 있다"는 상황을 명시적으로 표현하기 위해 사용된다. 간단히 말해, `null`을 직접 다루는 대신 `Optional`을 사용하면 값의 유무를 안전하게 처리할 수 있어 코드가 더 명확하고 안정적으로 작성할 수 있다.

`Optional`은 매우 다양한 기능을 제공한다. 아마도 지금 이 기능만 가지고는 `Optional`의 진가를 알기는 어려울 것이다. 앞서 이야기 한 것 처럼 옵셔널은 뒤에서 매우 자세히 다룬다. 여기서는 스트림의 최종 연산을 이해하는데 필요한 최소한의 옵셔널 기능만 알아보았다.

최종 연산

최종 연산(Terminal Operation)은 스트림 파이프라인의 **끝**에 호출되어 **실제 연산을 수행**하고 결과를 만들어낸다. 최종 연산이 실행된 후에 스트림은 **소모**되어 더 이상 사용할 수 없다.

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 초기값이 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)`

예제 코드로 알아보자.

```
package stream.operation;

import java.util.Arrays;
import java.util.List;
import java.util.Optional;
```

```java
import java.util.stream.Collectors;

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

        // Collectors는 뒤에서 더 자세히 (복잡한 수집이 필요할 때 사용)
        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);
        System.out.println();

        System.out.println("2. toList() (Java 16+)");
        List<Integer> evenNumbers2 = numbers.stream()
                .filter(n -> n % 2 == 0)
                .toList(); // 수정 불가능 리스트
        System.out.println("짝수 리스트: " + evenNumbers2);
        System.out.println();

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

        System.out.println("4. forEach - 각 요소 처리");
        numbers.stream()
                .limit(5)
                .forEach(n -> System.out.print(n + " "));
        System.out.println("\n");

        System.out.println("5. count - 요소 개수");
        long count = numbers.stream()
                .filter(n -> n > 5)
                .count();
        System.out.println("5보다 큰 숫자 개수: " + count);
        System.out.println();

        System.out.println("6. reduce - 요소들의 합");
        System.out.println("초기값이 없는 reduce");
```

```
        Optional<Integer> sum1 = numbers.stream()
                .reduce((a, b) -> a + b);
        System.out.println("합계(초기값 없음): " + sum1.get());

        System.out.println("초기값이 있는 reduce");
        int sum2 = numbers.stream()
                .reduce(100, (a, b) -> a + b);
        System.out.println("합계(초기값 100): " + sum2);
        System.out.println();

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

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

        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);
        System.out.println();

        System.out.println("10. findAny - 아무 요소나 하나 찾기");
        Integer any = numbers.stream()
                .filter(n -> n > 5)
                .findAny().get();
        System.out.println("5보다 큰 아무 숫자: " + any);
        System.out.println();

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

        System.out.println("12. allMatch - 모든 요소가 조건을 만족하는지");
```

```
        boolean allPositive = numbers.stream()
                .allMatch(n -> n > 0);
        System.out.println("모든 숫자가 양수인가? " + allPositive);
        System.out.println();

        System.out.println("13. noneMatch - 조건을 만족하는 요소가 없는지");
        boolean noNegative = numbers.stream()
                .noneMatch(n -> n < 0);
        System.out.println("음수가 없나? " + noNegative);
    }
}
```

실행 결과

```
1. collect - List 수집
짝수 리스트: [2, 2, 4, 6, 8, 10]

2. toList() (Java 16+)
짝수 리스트: [2, 2, 4, 6, 8, 10]

3. toArray - 배열로 변환
짝수 배열: [2, 2, 4, 6, 8, 10]

4. forEach - 각 요소 출력
1 2 2 3 4

5. count - 요소 개수
5보다 큰 숫자 개수: 5

6. reduce - 요소들의 합
초기값이 없는 reduce
합계(초기값 없음): 62
초기값이 있는 reduce
합계(초기값 0): 162

7. min - 최솟값
최솟값: 1

8. max - 최댓값
최댓값: 10

9. findFirst - 첫 번째 요소
```

```
5보다 큰 첫 번째 숫자: 6

10. findAny - 아무 요소나 찾기
5보다 큰 아무 숫자: 6

11. anyMatch - 조건을 만족하는 요소 존재 여부
짝수가 있나? true

12. allMatch - 모든 요소가 조건을 만족하는지
모든 숫자가 양수인가? true

13. noneMatch - 조건을 만족하는 요소가 없는지
음수가 없나? true
```

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)
- 초기값을 주는 형태, 주지 않는 형태 두 가지가 있다. 초기값이 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 만든다.
- 최종 연산을 한 번 수행하면, **스트림은 재사용할 수 없다.**
- `reduce`를 사용할 때 초기값을 지정하면, 스트림이 비어 있어도 **초기값**이 결과가 된다. 초기값이 없으면 `Optional`을 반환한다.
  - 초기값이 없는데, 스트림이 비어 있을 경우 빈 `Optional`(`Optional.empty()`)을 반환한다.
- `findFirst()`, `findAny()`도 결과가 없을 수 있으므로 **Optional**을 통해 값 유무를 확인해야 한다.

지금까지 **스트림 생성**, **중간 연산**, **최종 연산**에 대해 알아보았다.
스트림은 컬렉션이나 배열을 사용하는 데 있어 코드를 단순화하고, 다양한 데이터 처리 연산을 간결하게 표현할 수 있게 해준다. 상황에 맞는 **중간 연산**과 **최종 연산**을 적절히 조합해서, 가독성과 유지보수성이 높은 코드를 작성하자.

기본형 특화 스트림

스트림 API에는 **기본형(primitive) 특화 스트림**이 존재한다.
자바에서는 `IntStream`, `LongStream`, `DoubleStream` 세 가지 형태를 제공하여 **기본 자료형(int, long, double)에 특화된 기능**을 사용할 수 있게 한다.

- 예를 들어, `IntStream`은 **합계 계산**, **평균**, **최솟값**, **최댓값** 등 정수와 관련된 연산을 좀 더 편리하게 제공하고, **오토박싱/언박싱** 비용을 줄여 성능도 향상시킬 수 있다.

기본형 특화 스트림의 종류

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

기본형 특화 스트림은 **합계**, **평균** 등 자주 사용하는 연산을 **편리한 메서드**로 제공한다. 또한, **타입 변환**과 **박싱/언박싱** 메서드도 제공하여 다른 스트림과 연계해 작업하기 수월하다.

참고로 기본형 특화 스트림은 `int`, `long`, `double` 같은 숫자를 사용한다. 따라서 숫자 연산에 특화된 메서드를 제공할 수 있다.

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();

코드로 알아보자.

```
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 가능)
        IntStream range1 = IntStream.range(1, 6);       // [1,2,3,4,5]
        IntStream range2 = IntStream.rangeClosed(1, 5); // [1,2,3,4,5]

        // 1. 통계 관련 메서드 (sum, average, max, min ,count)
        // sum(): 합계 계산
        int sum = IntStream.range(1, 6).sum(); // 15
        System.out.println("sum = " + sum);

        // average(): 평균값 계산
```

```
        double avg = IntStream.range(1, 6)
                .average()
                .getAsDouble(); // 3.0
        System.out.println("avg = " + avg);

        // summaryStatistics(): 모든 통계 정보
        IntSummaryStatistics stats = IntStream.range(1,
6).summaryStatistics();
        System.out.println("합계: " + stats.getSum());         // 15
        System.out.println("평균: " + stats.getAverage());     // 3.0
        System.out.println("최댓값: " + stats.getMax());       // 5
        System.out.println("최솟값: " + stats.getMin());       // 1
        System.out.println("개수: " + stats.getCount());       // 5

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

        // 3. 기본형 특화 매핑
        // int -> long 변환 매핑
        LongStream mappedLong = IntStream.range(1, 5)
                .mapToLong(i -> i * 10L);

        // int -> double 변환 매핑
        DoubleStream mappedDouble = IntStream.range(1, 5)
                .mapToDouble(i -> i * 1.5);

        // int -> 객체 변환 매핑
        Stream<String> mappedObj = IntStream.range(1, 5)
                .mapToObj(i -> "Number: " + i);

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

        // 5. 객체 스트림 -> 기본형 특화 스트림으로 매핑 활용
        int result = Stream.of(1, 2, 3, 4, 5)
                .mapToInt(i -> i)
                .sum();
```

```
        System.out.println("result = " + result);
    }
}
```

실행 결과

```
1 2 3 4 5
sum = 15
avg = 3.0
합계: 15
평균: 3.0
최댓값: 5
최솟값: 1
개수: 5
result = 15
```

- **기본형 특화 스트림(IntStream, LongStream, DoubleStream)** 을 이용하면 숫자 계산(합계, 평균, 최대·최소 등)을 간편하게 처리하고, 박싱/언박싱 오버헤드를 줄여 **성능상의 이점**도 얻을 수 있다.
- **range(), rangeClosed()** 같은 메서드를 사용하면 범위를 쉽게 다룰 수 있어 **반복문 대신**에 자주 쓰인다.
- `mapToXxx`, `boxed()` 등의 메서드를 잘 활용하면 **객체 스트림**과 **기본형 특화 스트림**을 자유롭게 오가며 다양한 작업을 할 수 있다.
- `summaryStatistics()`를 이용하면 합계, 평균, 최솟값, 최댓값 등 통계 정보를 **한 번에** 구할 수 있어 편리합니다.

기본형 특화 스트림을 잘 이용하면 **가독성**, **성능** 모두 잡을 수 있다. 따라서 숫자 중심의 연산에서는 적극 활용하는 것을 고려하자.

성능 - 전통적인 for문 vs 스트림 vs 기본형 특화 스트림

실제로 어느정도 차이가 날지는 데이터 양, 연산 종류, JVM 최적화 등에 따라 달라지기 때문에 다음 내용은 참고만 하자.

- 전통적인 for문이 보통 가장 빠르다.
- 스트림보다 전통적인 for 문이 1.5배 ~ 2배정도 빠르다.
    - 여기서 말하는 스트림은 `Integer` 같은 객체를 다루는 `Stream`을 말한다.
    - 박싱/언박싱 오버헤드가 발생한다.

- 기본형 특화 스트림(`IntStream` 등)은 전통적인 for문에 가까운 성능을 보여준다.
  - 전통적인 for문과 거의 비슷하거나 전통적인 for문이 10% ~ 30% 정도 더 빠르다.
  - 박싱/언박싱 오버헤드를 피할 수 있다.
  - 내부적으로 최적화된 연산을 수행할 수 있다.

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.