스트림 API

소유자	종수 김
태그	

기본

스트림 API 시작

MyStreamV3를 통해, 필터와 맵 등을 수행하는 클래스를 직접 작성했었음.

```
List<String> result = MyStreamV3.of(students)
        .filter(s -> s.getScore() >= 80)```
    .map(s -> s.getName())
    .toList();
```

- 개발자가 작업을 '어떻게(How)' 수행해야하는지 보다, '무엇(What)'을 수행해야 하는지, 즉 원하는 결과에 집중할 수 있음.
- 이러한 방식을 선언적 프로그래밍 방식이라고 함.
- 자바도 Stream API라는 이름으로 스트림 관련 기능들을 제공.
    - 데이터들이 흘러가면서 필터되고 매핑되며 데이터가 물 흐르듯 흘러가는 형태

Stream 사용

```
package stream.start;

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

public class StreamStartMain {
    public static void main(String[] args) {
        List<String> names = List.of("Apple", "Banana", "Berry", "Tomato");

        //B로 시작하는 이름만 필터 후 리스트 수집
```

```
        Stream<String> stream = names.stream();
        List<String> result = stream
                .filter(name -> name.startsWith("B"))
                .map(String::toUpperCase)
                .toList();

        // 외부 반복
        for (String name : result) {
            System.out.println(name);
        }

        // 내부 반복 (foreach)
        names.stream()
                .filter(name -> name.startsWith("B"))
                .map(String::toUpperCase)
                .forEach(System.out::println);
    }
}
```

1. Stream 생성

   ```
   List<String> names = List.of("Apple", "Banana", "Berry", "Tomato");
   Stream<String> stream = names.stream();
   ```

   - List의 stream() 메서드를 통해, 자바가 제공하는 스트림을 생성할 수 있음.

2. 중간 연산(Intermediate Operations) - filter, map

   ```
   filter(name -> name.startsWith("B"))
     .map(s -> s.toUpperCase())
   ```

   - 중간 연산은 스트림에서 요소를 걸러내거나(필터링), 다른 형태로 변환(매핑)하는 기능

3. 최종 연산(Terminal Operation)

   ```
   List<String> result = stream
           .filter(name -> name.startsWith("B"))
   ```

```
        .map(s -> s.toUpperCase())
        .toList();
```

  - toList() 중간 연산에서 정의한 연산을 기반으로 최종 결과를 List로 만들어 반환.

- 외부 반복

```
System.out.println("=== 외부 반복 ==="); for (String s : result) {
    System.out.println(s);
 }
```

  - 사용자가 직접 반복 구문을 작성

- 내부 반복

```
System.out.println("=== forEach, 내부 반복 ==="); names.stream()
    .filter(name -> name.startsWith("B"))
      .map(s -> s.toUpperCase())
    .forEach(s -> System.out.println(s));
```

  - 내부 반복을 사용하면 스트림이 알아서 반복문을 수행. 개발자가 로직을 직접 작성할 필요 없음

정리

- **중간 연산**(`filter`, `map` 등)은 데이터를 걸러내거나 형태를 변환하며, **최종 연산**(`toList()`, `forEach` 등)을 통해 최종 결과를 모으거나 실행할 수 있다.
- 스트림의 내부 반복을 통해, "어떻게 반복할지(for 루프, while 루프 등) 직접 신경 쓰기보다는, 결과가 어떻게 변환되어야 하는지"에만 집중할 수 있다. 이런 특징을 **선언형 프로그래밍**(Declarative Programming) 스타일이라 한다.
- 메서드 참조는 람다식을 더 간결하게 표현하며, 가독성을 높여준다.

- 다양한 중간 연산과 최종 연산을 통해, 복잡한 데이터 처리 로직도 간단하고 선언적으로 구현할 수 있음.

Stream API란 ?

- 정의
  - 스트림(Stream)은 자바 8부터 추가된 기능으로 **데이터의 흐름을 추상화**해서 다루는 도구

- 컬렉션(Collection) 또는 배열 등의 요소들을 연산 파이프라인을 통해 연속적인 형태로 처리할 수 있게 해줌.
  - 연산 파이프라인 : 여러 연산(중간 연산, 최종 연산)을 체이닝해서 데이터를 변환, 필터링, 계산하는 구조.

> **용어: 파이프라인**
> 스트림이 여러 단계를 거쳐 변환되고 처리되는 모습이 **마치 물이 여러 파이프(관)를 타고 이동하면서 정수 시설이나 필터를 거치는 과정**과 유사하다. 각 파이프 구간마다(=중간 연산) 데이터를 가공하고, 마지막 종착지(=종료 연산)까지 흐른다는 개념이 비슷하기 때문에 '파이프라인'이라는 용어를 사용한다.

스트림의 특징

1. 데이터 소스를 변경하지 않음(Immutable)
   a. 스트림에서 제공하는 연산들은 원본 컬렉션(List, Set 등)을 변경하지 않고 **결과만 새로 생성**

```
package stream.basic;

import java.util.List;

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

        List<Integer> filterdList = originList.stream()
                .filter(n -> n % 2 == 0)
                .toList();
        System.out.println(filterdList);
    }
}

  originList = [1, 2, 3, 4, 5]
  filteredList = [2, 4]
  originList = [1, 2, 3, 4, 5]
```

- 원본 리스트는 변하지 않음

2. 일회성 소비
    a. 한 번 사용(소비)된 스트림은 다시 사용할 수 없으며, 필요하다면 **새로 스트림을 생성**해야 함.

```
package stream.basic;

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

public class DuplicateExecutionMain {
    public static void main(String[] args) {
        // 스트림 중복 실행 확인
        Stream<Integer> stream = Stream.of(1, 2, 3);
        stream.forEach(System.out::println); // 1. 최초 실행

        // 오류 메세지 : java.lang.IllegalStateException: stream has already bee
        // stream.forEach(System.out::println); // 2. 스트림 중복 실행 예외 발생.

        // 대안 : 대상 리스트를 스트림으로 새로 생성해서 사용
        List<Integer> list = List.of(1, 2, 3);
        Stream.of(list).forEach(System.out::println);
        Stream.of(list).forEach(System.out::println);
    }
}
```

- 같은 리스트를 여러번 스트림을 통해 실행해야 한다면, 스트림을 새로 생성하면 됨.

3. 파이프라인 구성
    a. 중간 연산들이 이어지다가 최종 연산을 만나면 연산이 수행되고 종료
4. 지연 연산(Lazy Operation)
    a. 중간 연산은 필요할 때 까지 실제로 동작하지 않고, 최종 연산이 실행될 때 한 번에 처리
5. 병렬 처리(Parallel) 용이
    a. 스트림으로부터 병렬 스트림을 쉽게 만들 수 있어서, 멀티코어 환경에서 병렬 연산을 비교적 단순한 코드로 작성 가능

파이프라인 구성

1. 파이프라인 구성
    a. 중간 연산들이 이어지다가 최종 연산을 만나면 연산이 수행되고 종료

```
package stream.basic;

import lambda.lambda5.mystream.MyStreamV3;

import java.util.List;

public class LazyEvalMain1 {
    public static void main(String[] args) {
        List<Integer> data = List.of(1, 2, 3, 4, 5, 6);
        ex1(data);
        ex2(data);
    }
    private static void ex1(List<Integer> data) {
        System.out.println("== MyStreamV3 시작==");
        List<Integer> result = MyStreamV3.of(data)
                .filter(i -> {
                    boolean isEven = i % 2 == 0;
                    System.out.println("filter() 실행 : " + i + "(" + isEven + ")");
                    return isEven;
                })
                .map(i -> {
                    int mapped = i * 10;
                    System.out.println("map() 실행: " + i + " -> " + mapped);
                    return mapped;
                })
                .toList();

        System.out.println("result = " + result);
        System.out.println("== MyStreamV3 종료==");
    }
    private static void ex2(List<Integer> data) {
        System.out.println("== Stream API 시작==");
        List<Integer> result = data.stream()
                .filter(i -> {
                    boolean isEven = i % 2 == 0;
```

```
                System.out.println("filter() 실행 : " + i + "(" + isEven + ")");
                return isEven;
            })
            .map(i -> {
                int mapped = i * 10;
                System.out.println("map() 실행: " + i + " -> " + mapped);
                return mapped;
            })
            .toList();

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

```
== MyStreamV3 시작==
filter() 실행 : 1(false)
filter() 실행 : 2(true)
filter() 실행 : 3(false)
filter() 실행 : 4(true)
filter() 실행 : 5(false)
filter() 실행 : 6(true)
map() 실행: 2 -> 20
map() 실행: 4 -> 40
map() 실행: 6 -> 60
result = [20, 40, 60]
== MyStreamV3 종료==
== Stream API 시작==
filter() 실행 : 1(false)
filter() 실행 : 2(true)
map() 실행: 2 -> 20
filter() 실행 : 3(false)
filter() 실행 : 4(true)
map() 실행: 4 -> 40
filter() 실행 : 5(false)
filter() 실행 : 6(true)
```

```
map() 실행: 6 -> 60
result = [20, 40, 60]
== StreamAPI 종료==
```

- MyStreamV3는 **일괄 처리 방식**
- 자바 Stream API는 **파이프라인 방식**

일괄처리 / 파이프라인 처리

- 일괄 처리

일괄 처리(Batch Processing) 비유

예시: 쿠키 공장

1. **반죽 공정**: 반죽을 전부 만들어서 한쪽에 쌓아 둔다.
2. **굽기 공정**: 쌓아 둔 반죽을 한꺼번에 오븐에 넣어 다 구워서 다시 쌓아 둔다.
3. **포장 공정**: 구워진 쿠키들을 다시 한 번에 포장 기계로 몰아넣어 포장한다.

즉,

- 한 공정(반죽)을 **모든 쿠키에 대해** 다 끝내면,
- 그 다음 공정(굽기)도 **모든 쿠키에 대해** 일괄적으로 처리하고,
- 마지막에 포장 역시 **모든 쿠키에 대해** 한꺼번에 진행한다.

이것이 바로 **일괄 처리** 방식이다. 각 단계마다 **결과물을 모아두고**, 전체가 끝난 뒤에야 다음 단계로 넘긴다.
스트림 관점에서 비유하자면,

- `filter()` (조건을 체크하는 작업)를 **모든 데이터**에 대해 적용(**일괄 처리**)하고,
- 그 결과를 한꺼번에 모아서, 그 다음에 `map()` (변환을 담당하는 작업)을 **일괄 처리**하는 모습이다.

- 파이프라인 처리

파이프라인 처리(Pipeline Processing) 비유

예시: 조립 라인이 있는 자동차 공장

1. **프레임 조립** 담당: 차체 뼈대를 조립하면, 바로 다음 공정으로 넘긴다.
2. **엔진 장착** 담당: 프레임이 오면 곧바로 엔진을 달아주고, 다음 공정으로 넘긴다.
3. **도색** 담당: 엔진이 장착된 차체가 도착하면 즉시 도색을 하고, 다음 공정으로 보낸다.
4. ... (이후 공정들)
5. **출고**: 모든 공정이 끝난 차는 즉시 공장에서 출하한다.

일괄 처리와 가장 큰 차이는, 일괄 처리는 모든 작업을 끝내고 다음 단계로 넘긴다면, 파이프라인 처리 방식은 하나의 작업이 처리되면 바로 다음 단계로 넘긴다는 점이다.

자동차 한 대가 프레임 조립을 마치면, **곧바로** 그 다음 공정인 엔진 장착으로 넘어가고, 그 사이에 새로운 차량 프레임이 조립 담당에게 들어온다.

- 즉, **하나의 제품(자동차)**이 여러 공정을 **흐르듯이** 쭉 통과하고, 끝난 차량은 바로 출하된다.

이를 **파이프라인 처리**라고 한다. 각 공정이 끝난 제품을 즉시 다음 단계로 넘기면서, 공정들이 **연결(체이닝)** 되어 있는 형태이다.
자바 스트림 관점에서 비유하자면,

- `filter()` 공정을 통과하면, 해당 요소는 **곧바로** `map()` 공정으로 이어지고,
- 최종 결과를 가져야 하는 시점(`toList()`, `forEach()`, `findFirst()` 등)이 되어서야 **최종 출고**를 한다.

정리

일괄 처리

- 공정(중간 연산)을 단계 별로 쪼개서 데이터 전체를 한 번에 처리하고, 결과를 저장해두었다가 다음 공정을 또 한번에 수행.

파이프라인 처리

- 한 요소(제품)가 한 공정을 마치면, 즉시 다음 공정으로 넘어가는 구조
- 자동차 공장에서 조립 라인에 제품이 흐르는 모습

코드 분석

MyStreamV3

1. data(1,2,3,4,5,6)
2. filter(1,2,3,4,5,6) → 2,4,6 통과
3. map(2,4,6) → 20,40,60
4. list(20,40,60)

- data에 있는 요소를 한 번에 모두 꺼내서 filter에 적용

- filter()가 모든 요소에 대해 순서대로 전부 실행된 뒤, 조건에 통과한 요소에 대해 map()이 한 번에 실행
- map()의 실행이 모두 끝나고 20,40,60이 최종 list에 담김.
    - 즉 모든 요소의 변환이 끝난 뒤에야 최종 결과가 생성

Stream API

1. data(1) → filter(1) → false → skip
2. data(2) → filter(2) → true → map(2) → 20 실행 → list(20)
3. data(3) → filter(3) → false → skip
4. data(4) → filter(4) → true → map(4) → 40 실행 → list(40)
5. data(5) → filter(5) → false → skip
6. data(6) → filter(6) → true → map(6) → 60 실행 → list(60)

- data에 있는 요소를 하나씩 꺼내서, filter,map을 적용하는 구조
- 한 요소가 filter를 통과하면, 곧바로 map이 적용되는 **파이프라인 처리**
- 각각의 요소가 개별적으로 파이프라인을 따라 별도로 처리되는 방식.

지연 연산

파이프라인의 장점을 설명하기 전 지연 연산 설명

- 최종 연산을 수행해야 작동한다.

  자바 스트림은 toList()와 같은 최종 연산을 수행할 때만 작동.

  최종 연산을 제외하면, 동작하지 않는다.

```
private static void ex2(List<Integer> data) {
    System.out.println("== Stream API 시작==");
    data.stream()
            .filter(i -> {
                boolean isEven = i % 2 == 0;
                System.out.println("filter() 실행 : " + i + "(" + isEven + ")");
                return isEven;
            })
            .map(i -> {
                int mapped = i * 10;
                System.out.println("map() 실행: " + i + " → " + mapped);
```

```
                return mapped;
            });

//        System.out.println("result = " + result);
        System.out.println("== StreamAPI 종료==");
    }
    // 수행을 안함.
    == Stream API 시작==
        == StreamAPI 종료==
```

- MyStreamV3는 최종 연산이 없어도 filter, map이 바로바로 실행되어 모든 로그가 찍힘.
- 반면에, 자바 스트림은 최종 연산이 없으면 아무 일도 일어나지 않음.
- '스트림 API의 지연 연산'
  - 중간 연산들은 '이런 일을 할 것이다'라는 파이프 라인만 설정, 실제 연산은 최종 연산이 호출되기 전 까지 전혀 진행되지 않음.
  - 스트림은 filter, map을 호출할 때 전달한 람다를 내부에 저장만 해두고 실행하지는 않는 것.
    이후에 최종 연산이 호출되면 그 때 각각의 항목을 꺼내서 저장해둔 람다를 실행.

즉시 연산

- 우리가 만든 `MyStreamV3`는 **즉시(Eager) 연산**을 사용하고 있다. 예제 코드를 보면, `filter`, `map` 같은 중간 연산이 호출될 때마다 **바로 연산을 수행**하고, 그 결과를 내부 `List` 등에 저장해두는 방식을 취하고 있음을 확인할 수 있다.
- 그 결과, 최종 연산이 없어도 `filter`, `map` 등이 즉시 동작해버려 모든 로그가 찍히고, 필요 이상의 연산이 수행되기도 한다.

지연 연산

- 쉽게 이야기하면 스트림 API는 매우 게으르다(Lazy). 정말 꼭! 필요할 때만 연산을 수행하도록 최대한 미루고 미룬다.
- 그래서 연산을 반드시 수행해야 하는 최종 연산을 만나야 본인이 가지고 있던 중간 연산들을 수행한다.
- 이렇게 꼭 필요할 때 까지 연산을 최대한 미루는 것을 **지연(Lazy) 연산** 이라 한다.

지연 연산과 최적화

자바의 스트림은 지연 연산, 파이프라인 방식등 복잡하게 설계되어 있음.
이는 최적화를 위한 것.

예시.

Ex) 데이터 리스트 중에 짝수를 찾고, 찾은 짝수에 10을 곱한 뒤 계산한 짝수 중 첫 번째 항목만 찾는다면 ?

```
package stream.basic;

import lambda.lambda5.mystream.MyStreamV3;

import java.util.List;

public class LazyEvalMain3 {
    public static void main(String[] args) {
        List<Integer> data = List.of(1, 2, 3, 4, 5, 6);
        ex1(data);
        ex2(data);
    }
```

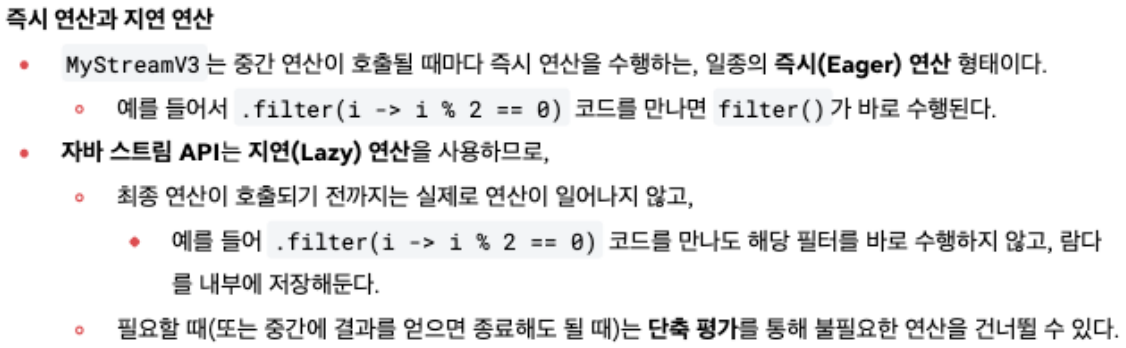
```
    private static void ex1(List<Integer> data) {
        System.out.println("== MyStreamV3 시작==");
        Integer result = MyStreamV3.of(data)
                .filter(i -> {
                    boolean isEven = i % 2 == 0;
                    System.out.println("filter() 실행 : " + i + "(" + isEven + ")");
                    return isEven;
                })
                .map(i -> {
                    int mapped = i * 10;
                    System.out.println("map() 실행: " + i + " -> " + mapped);
                    return mapped;
                })
                .getFirst();

        System.out.println("result = " + result);
        System.out.println("== MyStreamV3 종료==");
    }
    private static void ex2(List<Integer> data) {
        System.out.println("== Stream API 시작==");
        Integer result = data.stream()
                .filter(i -> {
                    boolean isEven = i % 2 == 0;
                    System.out.println("filter() 실행 : " + i + "(" + isEven + ")");
                    return isEven;
                })
                .map(i -> {
                    int mapped = i * 10;
                    System.out.println("map() 실행: " + i + " -> " + mapped);
                    return mapped;
                })
                .findFirst()
                .get();

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

```
== MyStreamV3 시작==
filter() 실행 : 1(false)
filter() 실행 : 2(true)
filter() 실행 : 3(false)
filter() 실행 : 4(true)
filter() 실행 : 5(false)
filter() 실행 : 6(true)
map() 실행: 2 -> 20
map() 실행: 4 -> 40
map() 실행: 6 -> 60
result = 20
== MyStreamV3 종료==
== Stream API 시작==
filter() 실행 : 1(false)
filter() 실행 : 2(true)
map() 실행: 2 -> 20
result = 20
== StreamAPI 종료==
```

- MyStreamV3
  - 모든 데이터에 대해 짝수를 걸러내고, 걸러진 결과에 전부 map을 한 후에 getFirst가 20을 반환
  - 모든 요소에 대한 필터 + 통과한 요소에 대해 map
  - 연산의 횟수가 불필요하게 많음
- Stream API
  - findFirst라는 최종 연산을 만나면,
    조건을 만족하는 요소 (2 → 20)을 찾은 순간 연산을 멈추고
    곧바로 결과를 반환.
  - filter(1) → false → 버림
  - filter(2) → true → map(2) → 20 → findFirst() 결과를 찾았으므로, 이후의 요소는 진행하지 않음.

이를 '단축 평가'(short-circuit)라고 하며, 조건을 만족하는 결과를 찾으면 더 이상 연산을 진행하지 않는 방식.

- 지연 연산과, 파이프라인 방식이 있기 때문에 가능한 최적화.



즉시 연산과 지연 연산

- `MyStreamV3`는 중간 연산이 호출될 때마다 즉시 연산을 수행하는, 일종의 **즉시(Eager) 연산** 형태이다.
  - 예를 들어서 `.filter(i -> i % 2 == 0)` 코드를 만나면 `filter()`가 바로 수행된다.
- **자바 스트림 API**는 **지연(Lazy) 연산**을 사용하므로,
  - 최종 연산이 호출되기 전까지는 실제로 연산이 일어나지 않고,
    - 예를 들어 `.filter(i -> i % 2 == 0)` 코드를 만나도 해당 필터를 바로 수행하지 않고, 람다를 내부에 저장해둔다.
  - 필요할 때(또는 중간에 결과를 얻으면 종료해도 될 때)는 **단축 평가**를 통해 불필요한 연산을 건너뛸 수 있다.

지연 연산 정리

스트림 API에서 지연 연산(Lazy Operation, 게으른 연산)이란, filter, map 같은 중간 연산들은 toList()와 같은 최종 연산(Terminal Operation)이 호출되기 전 까지 실제로 실행되지 않는다는 의미.

- 즉, 중간 연산들은 결과를 바로 계산하지 않고, "무엇을 할 지 "에 대한 설정만 저장
  - 쉽게 람다 함수만 내부에 저장해두고, 해당 함수를 실행하지 않음.
- 그리고 최종 연산(toList(), forEach(), findFirst() 등)이 실행되는 순간, 그때서야 중간 연산이 순차적으로 한 번에 수행 (저장해둔 람다를 실행)

장점.

1. **불필요한 연산의 생략(단축, Short-Circuiting)**
    - 예를 들어 `findFirst()`, `limit()` 같은 단축 연산을 사용하면, 결과를 찾은 시점에서 더 이상 나머지 요소들을 처리할 필요가 없다.
    - 스트림이 실제로 데이터를 처리하기 직전에, "이후 연산 중에 어차피 건너뛰어도 되는 부분"을 알아내 **불필요한 연산을 피할 수 있게** 한다.
2. **메모리 사용 효율**
    - 중간 연산 결과를 매 단계마다 별도의 자료구조에 저장하지 않고, 최종 연산 때까지 **필요할 때만 가져와서 처리**한다.
3. **파이프라인 최적화**
    - 스트림은 요소를 하나씩 꺼내면서(=순차적으로) `filter`, `map` 등 연산을 **묶어서** 실행할 수 있다.
    - 즉, 요소 하나를 꺼냈으면 → 그 요소에 대한 `filter` → 통과하면 `map` → ... → 최종 연산까지 진행하고, 다음 요소로 넘어간다.
    - 이렇게 하면 메모리를 절약할 수 있고, 짜잘짜잘하게 중간 단계를 저장하지 않아도 되므로, 내부적으로 효율적으로 동작한다.
    - 추가로 `MyStreamV3`와 같은 배치 처리 방식에서는 각 단계별로 모든 데이터를 한 번에 처리하기 때문에 지연 연산을 사용해도 이런 최적화가 어렵다. 자바 스트림 API는 필요한 데이터를 하나씩 불러서 처리하는 파이프라인 방식이므로 이런 최적화가 가능하다.

정리

지연 연산과 파이프라인 구조의 이해를 통해 스트림 API를 사용해 더 효율적이고 간결한 코드 작성 가능.

- 필요 시, 단축 연산을 적절히 활용하여 대량의 데이터를 다룰 때 불 필요한 연산 최소화 가능
- 코드는 선언적(무엇을 할 지 작성)이지만, 내부적으로는 '효율적으로 동작'
  따라서, 비교적 간단하게 성능 향상 가능.

따라서 **스트림의 지연 연산** 개념과 **단축 평가** 방식을 잘 이해해두면, 실무에서 대량의 컬렉션/데이터를 효율적으로 다룰 때 도움이 된다.
정리하면, **스트림 API의 핵심은 "어떤 연산을 할지" 파이프라인으로 정의해놓고, 최종 연산이 실행될 때 한 번에 처리**한다는 점이다.
이를 통해 "필요한 시점에만 데이터를 처리하고, 필요 이상으로 처리하지 않는다"는 효율성을 얻을 수 있다.