9. 스트림 API3 - 컬렉터

#1.인강/0.자바/7.자바-고급3편

- /컬렉터1
- /컬렉터2
- /다운 스트림 컬렉터1
- /다운 스트림 컬렉터2
- /정리

컬렉터1

스트림이 중간 연산을 거쳐 최종 연산으로써 데이터를 처리할 때, 그 결과물이 필요한 경우가 많다. 대표적으로 "리스트나 맵 같은 자료 구조에 담고 싶다"거나 "통계 데이터를 내고 싶다"는 식의 요구가 있을 때, 이 최종 연산에 `Collectors`를 활용한다.

collect 연산(예: `stream.collect(...)`)은 **반환값**을 만들어내는 최종 연산이다. `collect(Collector<? super T, A, R> collector)` 형태를 주로 사용하고, `Collectors` 클래스 안에 준비된 여러 메서드를 통해서 다양한 수집 방식을 적용할 수 있다.

참고: 필요한 대부분의 기능이 `Collectors`에 이미 구현되어 있기 때문에, `Collector` 인터페이스를 직접 구현하는 것보다는 `Collectors`의 사용법을 익히는 것이 중요하다.

Collectors의 주요 기능 표 정리

다음 표는 `Collectors`에서 자주 쓰이는 메서드와 그 설명을 간단히 정리한 것이다.

기능	메서드 예시	설명	반환 타입
List로 수집	toList() toUnmodifiableList()	스트림 요소를 List로 모은다. toUnmodifiableList()는 불변 리스트를 만든다.	`List<T>`

Set으로 수집	toSet() toCollection(HashSet::new)	스트림 요소를 Set으로 모은다. 중복 요소는 자동으로 제거된다. 특정 Set 타입으로 모으려면 toCollection() 사용.	`Set<T>`
Map으로 수집	toMap(keyMapper, valueMapper) toMap(keyMapper, valueMapper, mergeFunction, mapSupplier)	스트림 요소를 Map에 (키, 값) 형태로 수집한다. 중복 키가 생기면 mergeFunction으로 해결하고, mapSupplier로 맵 타입을 지정할 수 있다.	`Map<K, V>`
그룹화	groupingBy(classifier) groupingBy(classifier, downstreamCollector)	특정 기준 함수(classifier)에 따라 그룹별로 스트림 요소를 묶는다. 각 그룹에 대해 추가로 적용할 다운스트림 컬렉터를 지정할 수 있다.	`Map<K, List<T>>` 또는 `Map<K, R>`
분할	partitioningBy(predicate) partitioningBy(predicate, downstreamCollector)	predicate 결과가 true와 false 두 가지로 나뉘어, 2개 그룹으로 분할한다.	`Map<Boolean, List<T>>` 또는 `Map<Boolean, R>`
통계	counting(), summingInt(), averagingInt(), summarizingInt() 등	요소의 개수, 합계, 평균, 최소, 최댓값 등을 구하거나, IntSummaryStatistics 같은 통계 객체로도 모을 수 있다.	`Long`, `Integer`, `Double`, `IntSummaryStatistics` 등
리듀싱	reducing(...)	스트림의 reduce()와 유사하게, Collector 환경에서 요소를 하나로 합치는 연산을 할 수 있다.	`Optional<T>` 혹은 다른 타입

문자열 연결	joining(delimiter, prefix, suffix)	문자열 스트림을 하나로 합쳐서 연결한다. 구분자(delimiter), 접두사(prefix), 접미사(suffix) 등을 붙일 수 있다.	`String`
매핑	mapping(mapper, downstream)	각 요소를 다른 값으로 변환(mapper)한 뒤 다운스트림 컬렉터로 넘긴다.	다운스트림 결과 타입에 따름

가장 기본적인 수집 예시

```
package stream.collectors;

import java.util.List;
import java.util.Set;
import java.util.TreeSet;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class Collectors1Basic {

    public static void main(String[] args) {
        // 기본 기능
        List<String> list = Stream.of("Java", "Spring", "JPA")
                .collect(Collectors.toList()); // 수정 가능 리스트 반환
        System.out.println("list = " + list);

        // 수정 불가능 리스트
        List<Integer> unmodifiableList = Stream.of(1, 2, 3)
                .collect(Collectors.toUnmodifiableList());
        //unmodifiableList.add(4); // 런타임 예외
        System.out.println("unmodifiableList = " + unmodifiableList);

        Set<Integer> set = Stream.of(1, 2, 2, 3, 3, 3)
                .collect(Collectors.toSet());
        System.out.println("set = " + set);
```

```
        // 타입 지정
        Set<Integer> treeSet = Stream.of(3, 4, 5, 2, 1)
                .collect(Collectors.toCollection(TreeSet::new));
        System.out.println("treeSet = " + treeSet); // TreeSet은 정렬 상태를 유지
    }
}
```

실행 결과

```
list = [Java, Spring, JPA]
unmodifiableList = [1, 2, 3]
set = [1, 2, 3]
treeSet = [1, 2, 3, 4, 5]
```

이 예시에서는 스트림을 다양한 컬렉션으로 수집하는 방법을 보여준다.

- `toList()`는 수정 가능한 `ArrayList`로 수집한다.
- `toUnmodifiableList()`는 자바 10부터 제공하는 불변 리스트를 만들어서 수정할 수 없다.
- `toSet()`은 중복을 제거한 채로 Set에 수집한다.
- `toCollection(TreeSet::new)`처럼 `toCollection()`을 사용하면 원하는 컬렉션 구현체를 직접 지정할 수 있다. 예제에서는 `TreeSet`을 선택해 정렬 상태를 유지하게 했다.

참고: `Collectors.toList()` 대신에 자바 16 부터는 `stream.toList()`를 바로 호출할 수 있다. 이 기능은 불변 리스트를 제공한다.

참고: Collectors를 사용할 때는 static import 사용을 추천한다.

Map 수집

```
package stream.collectors;

import java.util.LinkedHashMap;
import java.util.Map;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class Collectors2Map {
```

```
    public static void main(String[] args) {
        Map<String, Integer> map1 = Stream.of("Apple", "Banana", "Tomato")
                .collect(Collectors.toMap(
                        name -> name,           // keyMapper
                        name -> name.length()  // valueMapper
                ));
        System.out.println("map1 = " + map1);

        // 키 중복 예외: java.lang.IllegalStateException: Duplicate key
/*
        Map<String, Integer> map2 = Stream.of("Apple", "Apple", "Banana")
                .collect(Collectors.toMap(
                        name -> name,           // keyMapper
                        name -> name.length()  // valueMapper
                ));
        System.out.println("map2 = " + map2);
*/

        // 키 중복 대안 (병합)
        Map<String, Integer> map3 = Stream.of("Apple", "Apple", "Banana")
                .collect(Collectors.toMap(
                        name -> name,
                        name -> name.length(),
                        (oldVal, newVal) -> oldVal + newVal // 중복될 경우 기존 값
+ 새 값
                ));
        System.out.println("map3 = " + map3);

        // Map의 타입 지정
        LinkedHashMap<String, Integer> map4 = Stream.of("Apple", "Apple",
"Banana")
                .collect(Collectors.toMap(
                        name -> name,
                        String::length,
                        (oldVal, newVal) -> oldVal + newVal, // key 중복 발생 시
조정 로직
                        LinkedHashMap::new // 결과 Map 타입 지정
                ));
        System.out.println("map4 = " + map4.getClass());
    }
}
```

실행 결과

```
map1 = {Apple=5, Tomato=6, Banana=6}
map3 = {Apple=10, Banana=6}
map4 = class java.util.LinkedHashMap
```

- `toMap(keyMapper, valueMapper)` : 각 요소에 대한 키, 값을 지정해서 `Map`을 만든다.
- 키가 중복되면 `IllegalStateException`이 발생한다(주석 해제 시 `map2` 예제).
- `(oldVal, newVal) -> oldVal + newVal` 같은 병합 함수를 지정하면, 중복 키가 나오더라도 기존 값과 새 값을 합쳐서 처리한다.
- 마지막 인자로 `LinkedHashMap::new`를 넘기면, 결과를 `LinkedHashMap`으로 얻을 수 있다(입력 순서를 유지).

실행 결과 - 주석 해제

```
map1 = {Apple=5, Tomato=6, Banana=6}
Exception in thread "main" java.lang.IllegalStateException: Duplicate key
Apple (attempted merging values 5 and 5)
    at java.base/
java.util.stream.Collectors.duplicateKeyException(Collectors.java:135)
    at java.base/
```

- `map2`에서는 동일한 키("Apple")가 두 번 생겼는데, 병합 함수가 없으므로 중복된 키 때문에 예외가 발생한 것이다.
- 이런 상황을 처리하려면 세 번째 인자로 병합 함수를 넘기거나, 스트림 단계에서 키가 중복되지 않도록 미리 걸러야 한다.

컬렉터2

그룹과 분할 수집

```
package stream.collectors;
```

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

public class Collectors3Group {

    public static void main(String[] args) {
        // 첫 글자 알파벳을 기준으로 그룹화
        List<String> names = List.of("Apple", "Avocado", "Banana",
"Blueberry", "Cherry");
        Map<String, List<String>> grouped = names.stream()
                .collect(Collectors.groupingBy(name -> name.substring(0, 1)));
        System.out.println("grouped = " + grouped);

        // 짝수(even)인지 여부로 분할(파티셔닝)
        List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6);
        Map<Boolean, List<Integer>> partitioned = numbers.stream()
                .collect(Collectors.partitioningBy(n -> n % 2 == 0));
        System.out.println("partitioned = " + partitioned);
    }
}
```

실행 결과

```
grouped = {A=[Apple, Avocado], B=[Banana, Blueberry], C=[Cherry]}
partitioned = {false=[1, 3, 5], true=[2, 4, 6]}
```

- `groupingBy(...)`는 특정 **기준**(예: 첫 글자)에 따라 스트림 **요소**를 여러 그룹으로 묶는다. 결과는 `Map<기준, List<요소>>` 형태다.
- `partitioningBy(...)`는 단순하게 `true`와 `false` 두 그룹으로 나눈다. 예제에서는 짝수(`true`), 홀수(`false`)로 분할했다.

최솟값 최댓값 수집

```java
package stream.collectors;

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

```java
public class Collectors4MinMax {
    public static void main(String[] args) {
        // 다운스트림 컬렉터에서 유용하게 사용
        Integer max1 = Stream.of(1, 2, 3)
                .collect(Collectors.maxBy(
                        ((i1, i2) -> i1.compareTo(i2)))
                ).get();
        System.out.println("max1 = " + max1);

        Integer max2 = Stream.of(1, 2, 3)
                .max((i1, i2) -> i1.compareTo(i2)).get();
        System.out.println("max2 = " + max2);

        Integer max3 = Stream.of(1, 2, 3)
                .max((Integer::compareTo)).get();
        System.out.println("max3 = " + max3);

        // 기본형 특화 스트림 사용
        int max4 = IntStream.of(1, 2, 3)
                .max().getAsInt();
        System.out.println("max4 = " + max4);
    }
}
```

실행 결과

```
max1 = 3
max2 = 3
max3 = 3
max4 = 3
```

- `Collectors.maxBy(...)`나 `Collectors.minBy(...)`를 통해 최소, 최댓값을 구할 수 있다.
- 다만 스트림 자체가 제공하는 `max()`, `min()` 메서드를 쓰면 더 간단하다.
- 기본형 특화 스트림(`IntStream` 등)을 쓰면 `.max().getAsInt()`처럼 바로 기본형으로 결과를 얻을 수 있다.
- `Collectors`의 일부 기능은 스트림에서 직접 제공하는 기능과 중복된다. `Collectors`의 기능들은 뒤에서 설명할 다운 스트림 컬렉터에서 유용하게 사용할 수 있다.

통계 수집

```
package stream.collectors;

import java.util.IntSummaryStatistics;
import java.util.stream.Collectors;
import java.util.stream.IntStream;
import java.util.stream.Stream;

public class Collectors4Summing {
    public static void main(String[] args) {
        // 다운스트림 컬렉터에서 유용하게 사용
        long count1 = Stream.of(1, 2, 3)
                .collect(Collectors.counting());
        System.out.println("count1 = " + count1);

        long count2 = Stream.of(1, 2, 3)
                .count();
        System.out.println("count2 = " + count2);

        // 다운스트림 컬렉터에서 유용하게 사용
        double average1 = Stream.of(1, 2, 3)
                .collect(Collectors.averagingInt(i -> i));
        System.out.println("average1 = " + average1);

        // 기본형 특화 스트림으로 변환
        double average2 = Stream.of(1, 2, 3)
                .mapToInt(i -> i)
                .average().getAsDouble();
        System.out.println("average2 = " + average2);

        // 기본형 특화 스트림 사용
        double average3 = IntStream.of(1, 2, 3)
                .average().getAsDouble();
        System.out.println("average3 = " + average3);

        // 통계
        IntSummaryStatistics stats = Stream.of("Apple", "Banana", "Tomato")
                .collect(Collectors.summarizingInt(String::length));
        System.out.println(stats.getCount());    // 3
        System.out.println(stats.getSum());      // 17 (5+6+6)
```

```
        System.out.println(stats.getMin());      // 5
        System.out.println(stats.getMax());      // 6
        System.out.println(stats.getAverage());  // 5.66...
    }
}
```

실행 결과

```
count1 = 3
count2 = 3
average1 = 2.0
average2 = 2.0
average3 = 2.0
3
17
5
6
5.666666666666667
```

- `counting()`은 요소 개수를 구한다.
- `averagingInt()`는 요소들의 평균을 구한다.
- `summarizingInt()`는 합계, 최솟값, 최댓값, 평균 등 다양한 통계 정보를 담은 `IntSummaryStatistics` 객체를 얻는다.
- 자주 쓰이는 통계 메서드로 `summingInt()`, `maxBy()`, `minBy()`, `counting()` 등이 있다.
- `Collectors`의 일부 기능은 스트림에서 직접 제공하는 기능과 중복된다. `Collectors`의 기능들은 뒤에서 설명할 다운 스트림 컬렉터에서 유용하게 사용할 수 있다.

리듀싱 수집

```
package stream.collectors;

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

public class Collectors5Reducing {

    public static void main(String[] args) {
```

```
        List<String> names = List.of("a", "b", "c", "d");

        // 컬렉션의 리듀싱은 주로 다운 스트림에 활용 (스트림이 제공하는 리듀싱이 있으므로)
        // 모든 이름을 하나의 문자열로 이어 붙이기
        String joined1 = names.stream()
                .collect(Collectors.reducing(
                        (s1, s2) -> s1 + "," + s2
                )).get();
        System.out.println("joined1 = " + joined1);

        String joined2 = names.stream()
                .reduce((s1, s2) -> s1 + "," + s2).get();
        System.out.println("joined2 = " + joined2);

        // 문자열 전용 기능
        String joined3 = names.stream()
                .collect(Collectors.joining(","));
        System.out.println("joined3: " + joined3);

        String joined4 = String.join(",", "a", "b", "c", "d");
        System.out.println("joined4: " + joined4);
    }
}
```

실행 결과

```
joined1 = a,b,c,d
joined2 = a,b,c,d
joined3: a,b,c,d
joined4: a,b,c,d
```

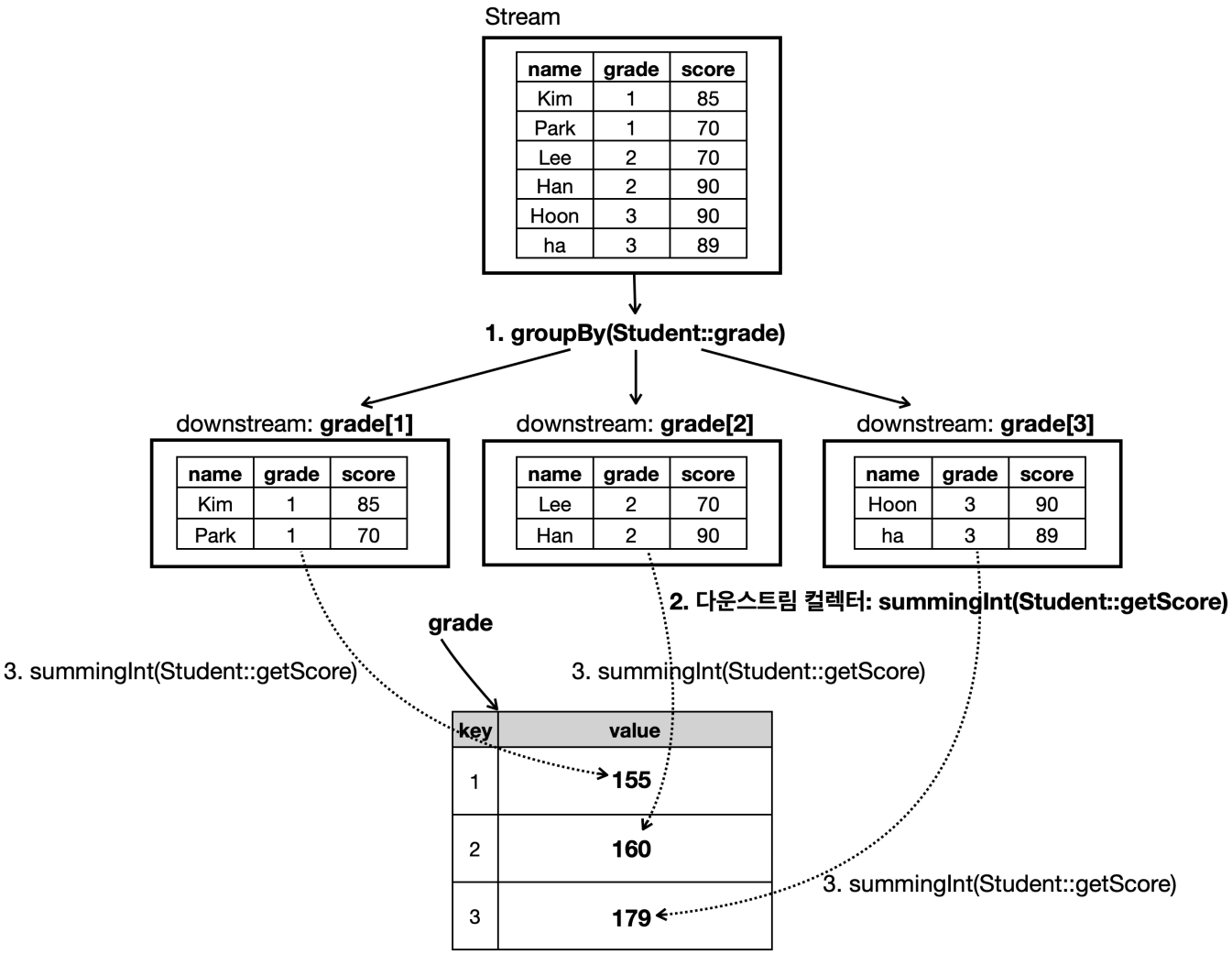
- `Collectors.reducing(...)`은 최종적으로 하나의 값으로 요소들을 합치는 방식을 지정한다. 여기서는 문자열들을 ,``로 이어붙였다.
- 스트림 자체의 `reduce(...)` 메서드와 유사한 기능이다.
- 문자열을 이어 붙일 때는 `Collectors.joining()`이나 `String.join()`을 쓰는 게 더 간편하다.
- `Collectors`의 일부 기능은 스트림에서 직접 제공하는 기능과 중복된다. `Collectors`의 기능들은 뒤에서 설명할 다운 스트림 컬렉터에서 유용하게 사용할 수 있다.

다운 스트림 컬렉터1

다운 스트림 컬렉터가 필요한 이유

- `groupingBy(...)`를 사용하면 일단 요소가 그룹별로 묶이지만, 그룹 내 요소를 구체적으로 어떻게 처리할지는 기본적으로 `toList()`만 적용된다.
- 그런데 실무에서는 "그룹별 **총합**, **평균**, **최대/최솟값**, **매핑된 결과**, **통계**" 등을 바로 얻고 싶을 때가 많다.
- 예를 들어, "학년별로 학생들을 그룹화한 뒤, 각 학년 그룹에서 **평균 점수**를 구하고 싶다"는 상황에서는 단순히 `List<Student>`로 끝나는 것이 아니라, 그룹 내 학생들의 점수를 합산하고 평균을 내는 동작이 더 필요하다.
  이처럼 **그룹화된 이후 각 그룹 내부에서 추가적인 연산 또는 결과물(예: 평균, 합계, 최댓값, 최솟값, 통계, 다른 타입으로 변환 등)을 정의하는** 역할을 하는 것이 바로 다운 스트림 컬렉터(Downstream Collector)이다.
- 이때 **다운 스트림 컬렉터를 활용**하면 "그룹 내부"를 다시 한번 모으거나 집계하여 원하는 결과를 얻을 수 있다.
  - 예: `groupingBy(분류함수, counting())` → 그룹별 개수
  - 예: `groupingBy(분류함수, summingInt(Student::getScore))` → 그룹별 점수 합계
  - 예: `groupingBy(분류함수, mapping(Student::getName, toList()))` → 그룹별 학생 이름 리스트

다운 스트림 컬렉터 예시 그림



- 각 학년별로 그룹화를 한 다음, 그룹화한 학년별 점수의 합을 구하는 방법

다운 스트림 컬렉터란?

- `Collectors.groupingBy(...)` 또는 `Collectors.partitioningBy(...)`에서 **두 번째 인자**로 전달되는 `Collector`를 가리켜 "다운 스트림 컬렉터"라 한다.
- 예를 들어 `Collectors.groupingBy(classifier, downstreamCollector)` 형태로 사용될 때, `downstreamCollector`는 `classifier`에 의해 분류된 각 그룹 내부의 요소들을 **다시 한 번** 어떻게 처리할지를 정의하는 역할을 한다.
- 예를 들어 학년 별로 그룹화 한다면, 그룹화한 각 학년별 요소들을 다시 한 번 어떻게 처리할지 정의하는 것이다.

```
// 예시
Map<KeyType, DownstreamResult> result =
    stream.collect(Collectors.groupingBy(
        element -> 분류 기준 Key,         // 1) groupingBy용 분류 함수
        downstreamCollector             // 2) 그룹 내부를 처리할 다운 스트림 컬렉터
    ));
```

- 만약 다운 스트림 컬렉터를 명시하지 않으면, 기본적으로 `Collectors.toList()`가 적용되어서 **그룹별 요소들을 List**로 모은다.
- 그러나 그룹별 **개수를 세거나**, **평균을 구하거나**, **특정 필드를 뽑아서 맵핑**하거나 등등의 작업이 필요하다면, 적절한 다운 스트림 컬렉터를 추가로 지정해야 한다.
- 다운 스트림 컬렉터는 그룹화(또는 분할)를 먼저 한 뒤, 각 그룹(또는 파티션) 내부의 요소들을 어떻게 처리할 것인가? 를 지정하는 데 사용된다.
    - 예를 들어, `groupingBy(분류 함수, counting())`라면 "각 그룹에 속한 요소들의 **개수**"를 구하는 다운 스트림 컬렉터가 된다.
    - 또 `groupingBy(분류 함수, averagingInt(속성))`라면 "각 그룹에 속한 요소들의 **속성 평균**"을 구하게 된다.
    - 여러 `Collector`를 중첩할 수도 있다. 예: `groupingBy(분류 함수, mapping(다른 함수, toList()))`처럼 "각 그룹에서 특정 속성만 매핑한 뒤 List로 수집하기" 등을 할 수 있다.

다운 스트림 컬렉터의 종류

Collector	사용 메서드 예시	설명	예시 반환 타입
counting()	Collectors.counting()	그룹 내(혹은 스트림 내) 요소들의 개수를 센다.	Long
summingInt() 등	Collectors.summingInt(...) Collectors.summingLong(...)	그룹 내 요소들의 특정 정수형 속성을 모두 합산한다.	Integer, Long 등
averagingInt() 등	Collectors.averagingInt(...) Collectors.averagingDouble(...)	그룹 내 요소들의 특정 속성 평균값을 구한다.	Double
minBy(), maxBy()	Collectors.minBy(Comparator) Collectors.maxBy(Comparator)	그룹 내 최소, 최댓값을 구한다.	Optional<T>

summarizingInt() 등	Collectors.summarizingInt(...) Collectors.summarizingLong(...)	개수, 합계, 평균, 최소, 최댓값을 동시에 구할 수 있는 SummaryStatistics 객체를 반환한다.	IntSummaryStatistics 등
mapping()	Collectors.mapping(변환 함수, 다운스트림)	각 요소를 다른 값으로 변환한 뒤, 변환된 값들을 다시 다른 Collector로 수집할 수 있게 한다.	다운스트림 반환 타입에 따라 달라짐
collectingAndThen()	Collectors.collectingAndThen(다른 컬렉터, 변환 함수)	다운 스트림 컬렉터의 결과를 최종적으로 한 번 더 가공(후처리)할 수 있다.	후처리 후의 타입
reducing()	Collectors.reducing(초깃값, 변환 함수, 누적 함수) Collectors.reducing(누적 함수)	스트림의 reduce()와 유사하게, 그룹 내 요소들을 하나로 합치는 로직을 정의할 수 있다.	누적 로직에 따라 달라짐
toList(), toSet()	Collectors.toList() Collectors.toSet()	그룹 내(혹은 스트림 내) 요소를 리스트나 집합으로 수집한다. toCollection(...)으로 구현체 지정 가능	List<T>, Set<T>

이 표는 다운 스트림 컬렉터의 대표적인 예시이다. `groupingBy(...)`, `partitioningBy(...)`에서 두 번째 인자로 활용되거나, 스트림의 `collect()`에서 직접 쓰이기도 한다.

다운 스트림 컬렉터 예제 1

다운 스트림 컬렉터를 실제로 사용해보자. 예제를 위해 간단한 학생(`Student`) 클래스를 먼저 만들자.

```
package stream.collectors;

public class Student {
```

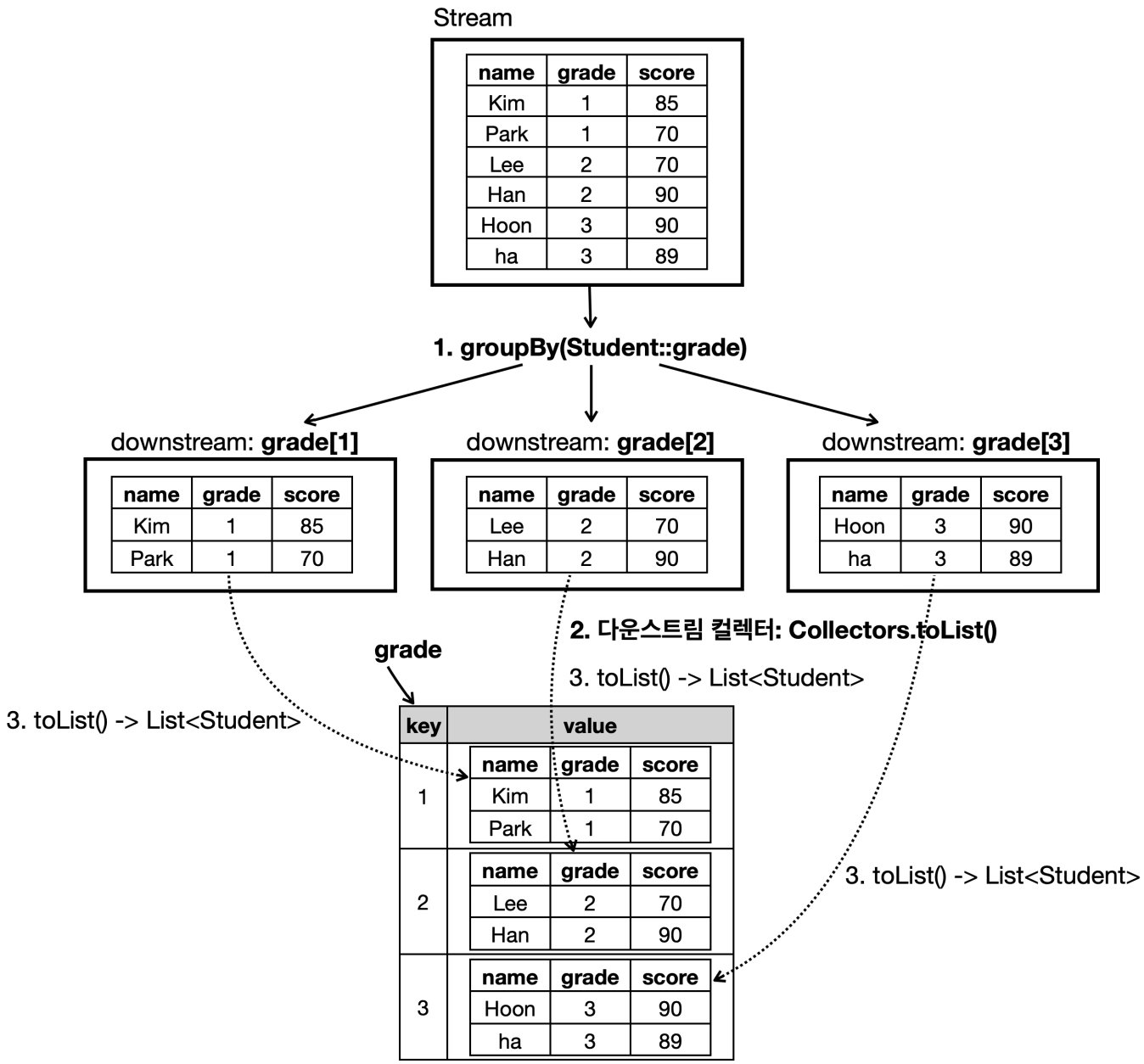
```
    private String name;
    private int grade;
    private int score;

    // Constructor, Getter, Setter
    public Student(String name, int grade, int score) {
        this.name = name;
        this.grade = grade;
        this.score = score;
    }

    public String getName() {
        return name;
    }

    public int getGrade() {
        return grade;
    }

    public int getScore() {
        return score;
    }

    @Override
    public String toString() {
        return "Student{" +
                "name='" + name + '\'' +
                ", grade=" + grade +
                ", score=" + score +
                '}';
    }
}
```

```
package stream.collectors;

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

public class DownStreamMain1 {
```

```
    public static void main(String[] args) {
        List<Student> students = List.of(
                new Student("Kim", 1, 85),
                new Student("Park", 1, 70),
                new Student("Lee", 2, 70),
                new Student("Han", 2, 90),
                new Student("Hoon", 3, 90),
                new Student("Ha", 3, 89)
        );

        // 1단계: 학년별로 학생들을 그룹화 해라.
        Map<Integer, List<Student>> collect1_1 = students.stream()
                .collect(Collectors.groupingBy(
                        Student::getGrade,  // 그룹화 기준: 학년
                        Collectors.toList() // 다운스트림1: 학생을 리스트로 수집
                ));
        System.out.println("collect1_1 = " + collect1_1);

        // 다운스트림에서 toList() 생략 가능
        Map<Integer, List<Student>> collect1_2 = students.stream()
                .collect(Collectors.groupingBy(Student::getGrade));
        System.out.println("collect1_2 = " + collect1_2);

        // 2단계: 학년별로 학생들의 이름을 출력해라.
        Map<Integer, List<String>> collect2 = students.stream()
                .collect(Collectors.groupingBy(
                        Student::getGrade, // 그룹화 기준: 학년
                        Collectors.mapping(Student::getName, // 다운스트림 1: 학생
-> 이름 변환
                                Collectors.toList() // 다운스트림 2: 변환된 값(이름)
을 List로 수집
                        )
                ));
        System.out.println("collect2 = " + collect2);

        // 3단계: 학년별로 학생들의 수를 출력해라.
        Map<Integer, Long> collect3 = students.stream()
                .collect(Collectors.groupingBy(
                        Student::getGrade,
                        Collectors.counting()
                ));
        System.out.println("collect3 = " + collect3);
```

```
        // 4단계: 학년별로 학생들의 평균 성적 출력해라.
        Map<Integer, Double> collect4 = students.stream()
                .collect(Collectors.groupingBy(
                        Student::getGrade,
                        Collectors.averagingInt(Student::getScore)
                ));
        System.out.println("collect4 = " + collect4);
    }

}
```

다운스트림 컬렉터 - Collectors.toList()



`groupingBy(Student::getGrade)`

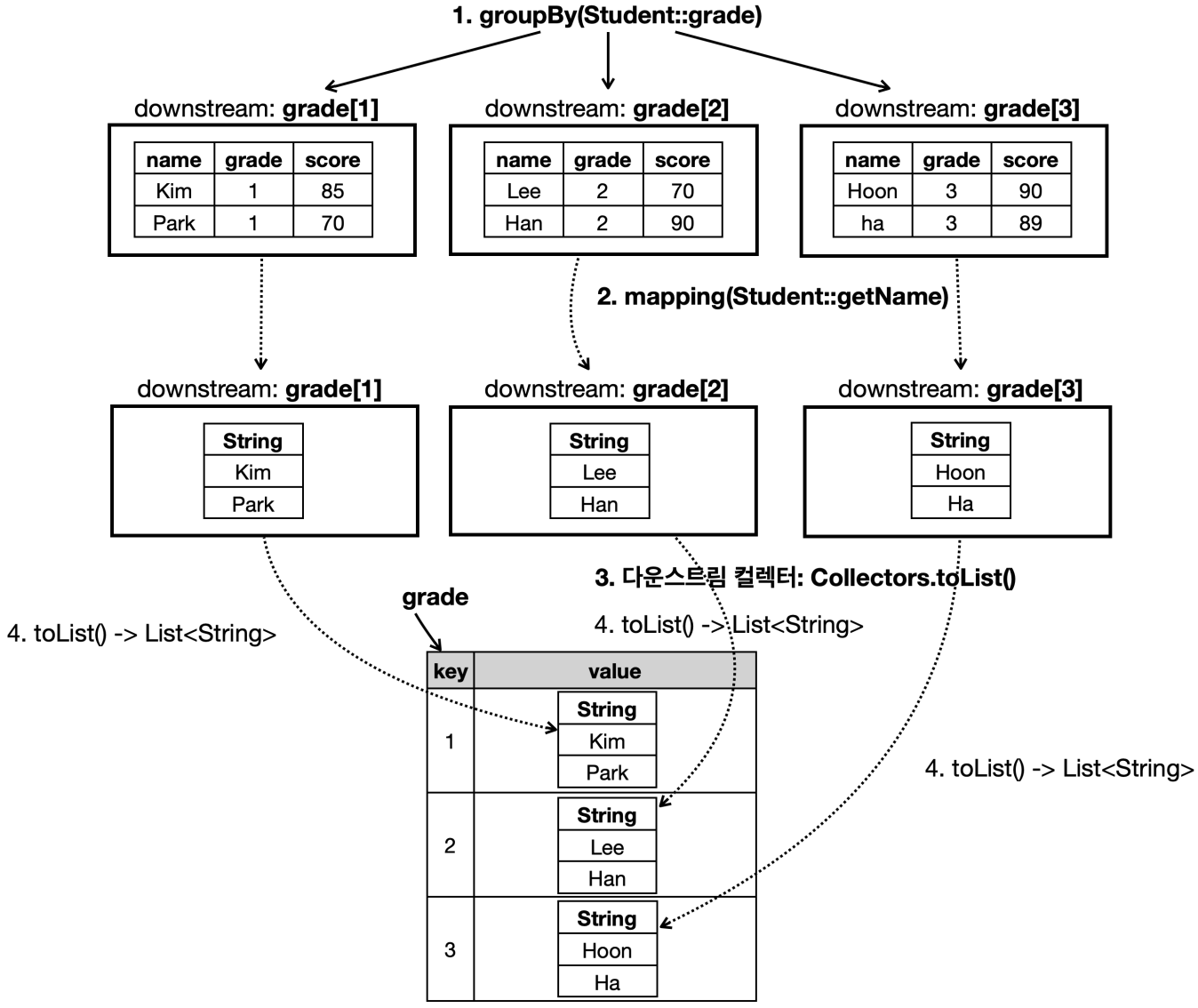
- 학년(grade)을 기준으로 학생(Student) 객체를 그룹화한다.

- **다운 스트림 컬렉터를 생략**하면 자동으로 `Collectors.toList()`가 적용되어 `Map<Integer, List<Student>>` 형태가 된다.

`groupingBy(Student::getGrade, toList())`

- 명시적으로 다운 스트림 컬렉터를 `toList()`로 지정한 것. 결과는 같음.

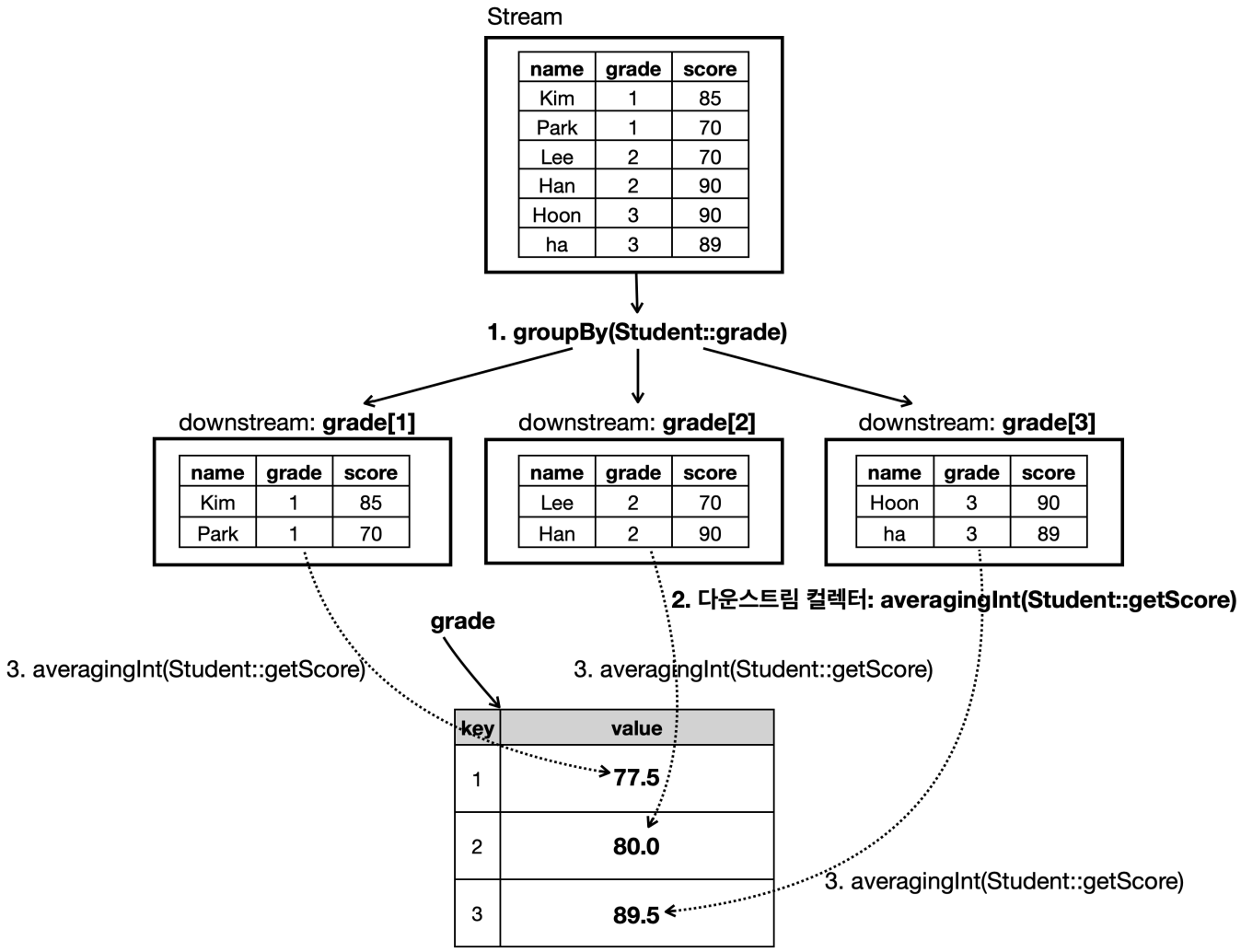
다운스트림 컬렉터 - mapping()



`groupingBy(Student::getGrade, mapping(Student::getName, toList()))`

- 스트림의 `map`을 떠올리면 된다.
- 먼저 "학년"으로 그룹화한 뒤, 그 그룹 내부에서 다시 학생(Student)을 "이름(String)"으로 매핑(mapping). 그리고 최종적으로 그 이름들을 리스트에 담는다.
- 즉, **그룹별로 학생들의 이름 목록**을 얻는다.

다운스트림 컬렉터 - 집계



`groupingBy(Student::getGrade, counting())`

- 그룹별로 **학생 수**를 구한다. 결과는 `Map<Integer, Long>`.

`groupingBy(Student::getGrade, averagingInt(Student::getScore))`

- 그룹별로 학생들의 **점수 평균**을 구한다. 결과는 `Map<Integer, Double>`.

실행 결과

```
collect1_1 = {1=[Student{name='Kim', grade=1, score=85}, Student{name='Park',
grade=1, score=70}], 2=[Student{name='Lee', grade=2, score=70},
Student{name='Han', grade=2, score=90}], 3=[Student{name='Hoon', grade=3,
score=90}, Student{name='Ha', grade=3, score=89}]}

collect1_2 = {1=[Student{name='Kim', grade=1, score=85}, Student{name='Park',
grade=1, score=70}], 2=[Student{name='Lee', grade=2, score=70},
Student{name='Han', grade=2, score=90}], 3=[Student{name='Hoon', grade=3,
score=90}, Student{name='Ha', grade=3, score=89}]}
```

```
collect2 = {1=[Kim, Park], 2=[Lee, Han], 3=[Hoon, Ha]}
collect3 = {1=2, 2=2, 3=2}
collect4 = {1=77.5, 2=80.0, 3=89.5}
```

다운 스트림 컬렉터2

이번에는 다른 예시로 다운 스트림 컬렉터를 알아보자.

다운 스트림 컬렉터 예제 2

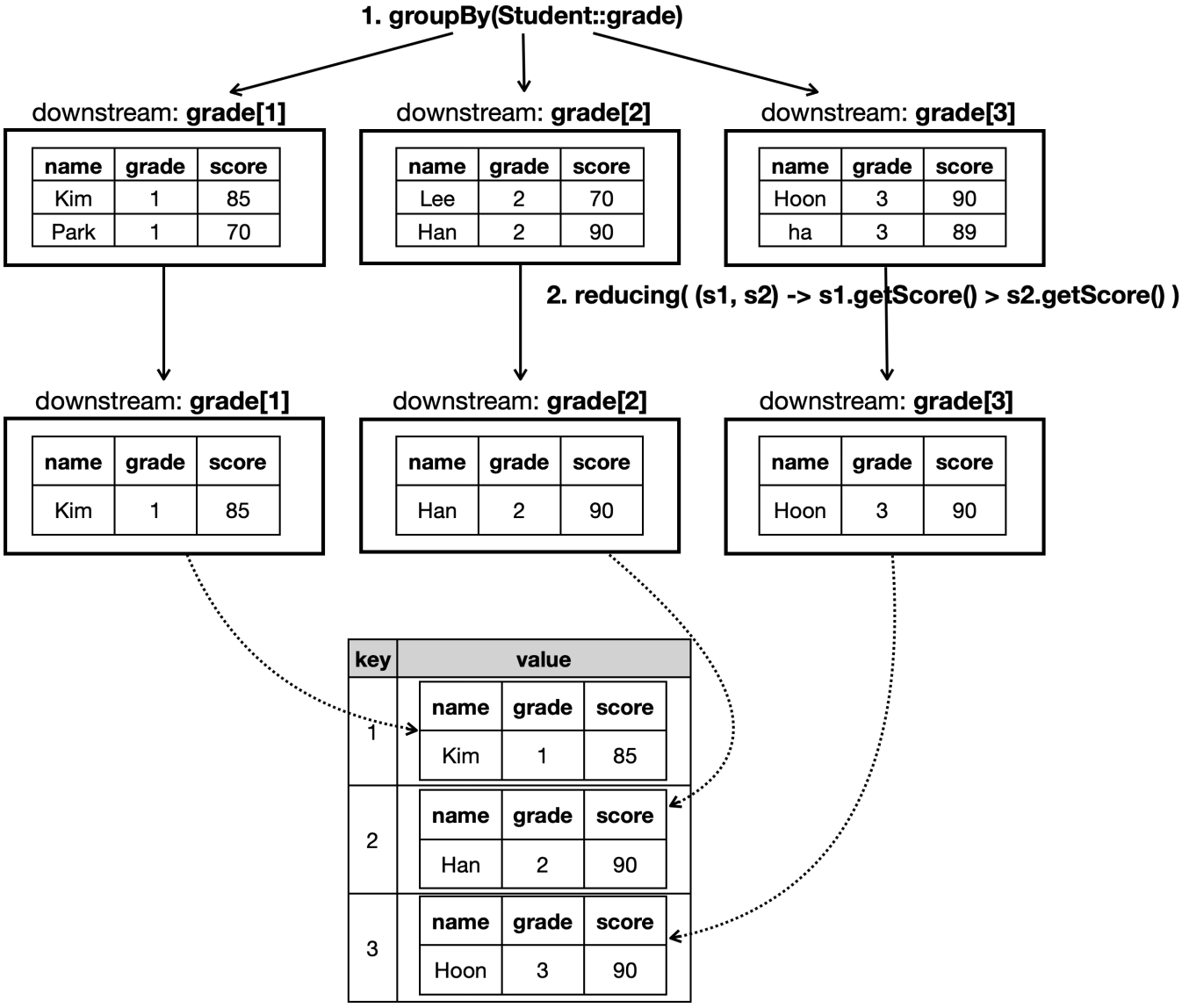
```
package stream.collectors;

import java.util.Comparator;
import java.util.List;
import java.util.Map;
import java.util.Optional;
import java.util.stream.Collectors;

public class DownStreamMain2 {
    public static void main(String[] args) {
        List<Student> students = List.of(
                new Student("Kim", 1, 85),
                new Student("Park", 1, 70),
                new Student("Lee", 2, 70),
                new Student("Han", 2, 90),
                new Student("Hoon", 3, 90),
                new Student("Ha", 3, 89)
        );

        // 1단계: 학년별로 학생들을 그룹화 해라.
        Map<Integer, List<Student>> collect1 = students.stream()
                .collect(Collectors.groupingBy(Student::getGrade));
        System.out.println("collect1 = " + collect1);

        // 2단계: 학년별로 가장 점수가 높은 학생을 구해라. reducing 사용
        Map<Integer, Optional<Student>> collect2 = students.stream()
```

```
                .collect(Collectors.groupingBy(
                        Student::getGrade,
                        Collectors.reducing((s1, s2) -> s1.getScore() >
s2.getScore() ? s1 : s2)
                ));
        System.out.println("collect2 = " + collect2);

        // 3 단계: 학년별로 가장 점수가 높은 학생을 구해라. maxBy 사용
        Map<Integer, Optional<Student>> collect3 = students.stream()
                .collect(Collectors.groupingBy(
                        Student::getGrade,
                        //Collectors.maxBy((s1, s2) -> s1.getScore() >
s2.getScore() ? 1 : -1)
                        //Collectors.maxBy(Comparator.comparingInt(student ->
student.getScore()))

Collectors.maxBy(Comparator.comparingInt(Student::getScore))
                ));
        System.out.println("collect3 = " + collect3);

        // 4단계: 학년별로 가장 점수가 높은 학생의 이름을 구해라 (collectingAndThen + maxBy
사용)
        // 학년별 그룹 -> 그룹별 가장 점수가 높은 학생 -> 그룹별 학생 -> 그룹별 이름
        Map<Integer, String> collect4 = students.stream()
                .collect(Collectors.groupingBy(
                        Student::getGrade,
                        Collectors.collectingAndThen(

Collectors.maxBy(Comparator.comparingInt(Student::getScore)),
                                sOpt -> sOpt.get().getName()
                        )
                ));
        System.out.println("collect4 = " + collect4);
    }

}
```

학년별 학생 목록(collect1)

- 단순히 `groupingBy(Student::getGrade)`만 사용해, `Map<Integer, List<Student>>` 형태로 수집한다.

다운스트림 컬렉터 - 리듀싱



학년별 최대 점수 학생 구하기(reducing 사용, collect2)

- `Collectors.reducing(...)` 은 그룹 내부의 학생들을 하나씩 비교하며 축소(reduce)하는 로직을 적용한다.
- `(s1, s2) -> s1.getScore() > s2.getScore() ? s1 : s2` 라는 식으로, 그룹 내의 학생 2명을 비교해 더 큰 점수를 가진 `Student` 를 반환하도록 했다. 그룹 내부의 모든 학생에 대해서 해당 로직을 적용한다. 따라서 각 그룹 별로 최종 1명의 학생이 남는다.
- 최종 결과는 `Map<Integer, Optional<Student>>` 형태이다.
  - 처음부터 학생이 하나도 없다면 결과도 없다. 따라서 Optional을 반환한다.

학년별 최대 점수 학생 구하기(maxBy 사용, collect3)

- `Collectors.maxBy(Comparator.comparingInt(Student::getScore))` 를 쓰면 간단히 최댓값 비교를 할 수 있다.
- 최종 결과는 `Map<Integer, Optional<Student>>` 형태이다.

학년별 최대 점수 학생의 "이름"만 구하기(collect4)

- `collectingAndThen`은 **다운 스트림 컬렉터**가 만든 결과를 한 번 더 후처리(And Then)할 수 있도록 해준다.
- 여기서는 `maxBy(...)`로 `Optional<Student>`가 만들어지면, 그 안에서 `Student::getName`을 꺼내 최종적으로 `String`이 되도록 변환하고 있다.
- 따라서 결과는 `Map<Integer, String>` 형태가 되며, 각 학년별로 점수가 가장 높은 학생의 이름만 구한다.

실행 결과

```
collect1 = {1=[Student{name='Kim', grade=1, score=85}, Student{name='Park',
grade=1, score=70}], 2=[Student{name='Lee', grade=2, score=70},
Student{name='Han', grade=2, score=90}], 3=[Student{name='Hoon', grade=3,
score=90}, Student{name='Ha', grade=3, score=89}]}

collect2 = {1=Optional[Student{name='Kim', grade=1, score=85}],
2=Optional[Student{name='Han', grade=2, score=90}],
3=Optional[Student{name='Hoon', grade=3, score=90}]}

collect3 = {1=Optional[Student{name='Kim', grade=1, score=85}],
2=Optional[Student{name='Han', grade=2, score=90}],
3=Optional[Student{name='Hoon', grade=3, score=90}]}

collect4 = {1=Kim, 2=Han, 3=Hoon}
```

mapping() vs collectingAndThen()

- `mapping()` : 그룹화(또는 분할)된 각 그룹 내의 **개별 요소들**을 **다른 값으로 변환**(mapping)한 뒤, 그 변환된 값들을 다시 **다른 Collector**로 수집할 수 있게 해준다.
- `collectingAndThen()` : 다운 스트림 컬렉터가 **최종 결과를 만든 뒤**에 **한 번 더 후처리**할 수 있도록 해준다. 즉, "1차 Collector → 후처리 함수" 순서로 작업한다.

요약 비교

구분	mapping()	collectingAndThen()

주된 목적	그룹 내 **개별 요소**를 변환한 뒤, 해당 변환 결과를 다른 Collector로 수집	그룹 내 요소들을 **이미 한 번** 수집한 결과를 **추가 가공**하거나 **최종 타입**으로 변환
처리 방식	(1) 그룹화 → (2) 각 요소를 변환 → (3) 리스트나 Set 등으로 수집	(1) 그룹화 → (2) 최댓값/최솟값/합계 등 수집 → (3) 결과를 후처리(예: Optional → String)
대표 예시	`mapping(Student::getName, toList())`	`collectingAndThen(maxBy(...), optional → optional.map(...) ...)`

핵심 포인트

- `mapping()`은 **그룹화된 요소 하나하나를 변환**하는 데 유용하고,
- `collectingAndThen()`은 이미 만들어진 **전체 그룹의 결과**를 **최종 한 번 더** 손보는 데 사용한다.

정리

다운 스트림 컬렉터를 이해하면, `groupingBy()`나 `partitioningBy()`로 그룹화/분할을 한 뒤 **내부 요소를 어떻게 가공하고 수집**할지 자유롭게 설계할 수 있다.

- `mapping()`, `counting()`, `summarizingInt()`, `reducing()`, `collectingAndThen()`, `maxBy()`, `minBy()`, `summingInt()`, `averagingInt()` 등 다양한 `Collector` 메서드를 조합하여 복잡한 요구사항도 단 한 번의 스트림 파이프라인으로 처리할 수 있다.

정리

우리는 **자바 스트림 API**의 핵심 개념과 활용 방법을 다양하게 살펴보았다. 직접 만든 `MyStreamV3`와 비교하면서 자바 스트림이 어떻게 작동하고, 어떤 장점이 있는지 알 수 있었을 것이다. 중요한 포인트들을 정리하자면 다음과 같다.

1. **스트림(Stream)이란?**
    - 자바 8부터 추가된 **데이터 처리 추상화** 도구로, 컬렉션/배열 등의 요소들을 일련의 단계(파이프라인)로 연결해 가공, 필터링, 집계할 수 있다.
    - 내부 반복(forEach 등)을 지원해, "어떻게 반복할지"보다는 "무엇을 할지"에 집중하는 **선언형 프로그래밍** 스타일을 구현한다.

2. **중간 연산(Intermediate Operation)과 최종 연산(Terminal Operation)**
   - **중간 연산**: `filter`, `map`, `distinct`, `sorted`, `limit` 등. 스트림을 변환하거나 필터링하는 단계. **지연(Lazy) 연산**이라서 실제 데이터 처리는 최종 연산을 만나기 전까지 미뤄진다.
   - **최종 연산**: `forEach`, `toList`, `count`, `min`, `max`, `reduce`, `collect` 등. 스트림 파이프라인을 종료하며 실제 연산을 수행해 결과를 반환한다.
   - 한 번 최종 연산을 수행하면 스트림은 소멸되므로, **재사용할 수 없다.**

3. **지연 연산(Lazy Evaluation)**
   - 스트림은 중간 연산 시점에 곧바로 처리하지 않고, 내부에 "어떤 연산을 할 것인지"만 저장해둔다.
   - 최종 연산이 호출되는 순간에야 중간 연산들을 한 번에 적용하여 결과를 만든다.
   - 덕분에 **단축 평가(Short-Circuit)** 같은 최적화가 가능하다. 예를 들어 `findFirst()`, `limit()` 등으로 불필요한 연산을 건너뛸 수 있다.

4. **파이프라인(pipeline)과 일괄 처리(batch) 비교**
   - 우리가 직접 만든 `MyStreamV3`처럼 모든 요소를 한 번에 처리하고, 그 결과를 모아서 다음 단계로 넘어가는 방식을 **일괄 처리**라고 한다.
   - 자바 스트림은 요소 하나를 `filter` → 통과 시 바로 `map` → … → 최종 연산으로 넘기는 식의 **파이프라인 방식**으로 동작한다.
   - 파이프라인 구조와 지연 연산 덕분에, 필요 이상의 연산을 줄이고 메모리 효율도 높일 수 있다.

5. **기본형 특화 스트림(IntStream, LongStream, DoubleStream)**
   - 박싱/언박싱 오버헤드를 줄이고, 합계, 평균, 최솟값, 최댓값, 범위 생성 같은 **숫자 처리에 특화된 메서드**를 제공한다.
   - 일반 스트림보다 루프가 매우 큰 상황에서 성능상 이점이 있을 수 있고, `range()`, `rangeClosed()`를 통해 반복문 없이 손쉽게 범위를 다룰 수도 있다.

6. **Collector와 Collectors**
   - `collect` 최종 연산을 통해 스트림 결과를 **리스트나 맵, 통계 정보** 등 원하는 형태로 모을 수 있다.
   - `Collectors` 클래스는 `toList`, `toSet`, `groupingBy`, `partitioningBy`, `mapping`, `averagingInt` 같은 **다양한 수집용 메서드**를 제공한다.
   - 특히 `groupingBy`나 `partitioningBy`에 **다운 스트림 컬렉터**를 지정하면, "그룹별 합계, 평균, 최대/최솟값, 여러 형태로 다시 매핑" 등 **복합적인 요구사항**을 한 번에 처리할 수 있다.

이처럼 스트림은 **가독성**, **선언형 코드**, **지연 연산에 따른 최적화**라는 장점을 제공한다. 간단한 데이터 필터링이나 변환부터 대규모 그룹화/집계 처리까지, 여러 상황에서 복잡한 반복문 없이 직관적인 코드를 작성할 수 있다.
스트림을 이해하고 적절히 활용하면, 실무에서 매우 큰 데이터도 효율적이고 깔끔하게 처리할 수 있다.