스트림 API - 컬렉터

소유자	종수 김
태그	

컬렉터 - 1

스트림이 중간 연산을 거쳐 최종 연산으로써 데이터를 처리할 때, 그 결과물이 필요할 경우. (Ex. List, Map, Set 등) Collectors를 활용.

- 필요한 대부분의 기능이 Collectors에 이미 있으므로, 인터페이스를 직접 구현하기보다 사용법을 익히는 것도 좋음.

▼ 예제

다음 표는 Collectors에서 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

```
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) {
        // 수정 가능 List 반환
        List<String> list = Stream.of("Java", "Spring", "JPA")
                .collect(Collectors.toList());
        list.add("Hello");
        System.out.println(list);

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

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

        // 타입 지정
        TreeSet<Integer> treeSet = Stream.of(3, 4, 5, 2, 1,7)
                .collect(Collectors.toCollection(TreeSet::new)); // TreeSet은 정렬
        treeSet.add(6);
        System.out.println(treeSet);
    }
}
```

- Java 16부터는 toList()가 가능. - 기본적으로 불변 리스트
- 만약 사용한다면 static import 추천

Map

```
package stream.collectors;

import java.util.*;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class Collectors2Basic {
    public static void main(String[] args) {
        // Map으로 변환 가능
        Map<String, Integer> map1 = Stream.of("Apple", "Banana", "Tomato")
                .collect(Collectors.toMap(
                        name -> name, // key
                        name -> name.length() // value
                ));
        System.out.println(map1);

//        // key가 중복이라면, 중복 에러
//        Map<String, Integer> map2 = Stream.of("Apple", "Apple", "Tomato")
//                .collect(Collectors.toMap(
//                        name -> name, // key
//                        name -> name.length() // value
//                ));
//        System.out.println(map2);

        // 키 중복 대안 (병합)
        Map<String, Integer> map3 = Stream.of("Apple", "Apple", "Tomato")
                .collect(Collectors.toMap(
                        name -> name, // key
                        name -> name.length(), // value
                        (oldValue, newValue) -> oldValue + newValue // 중복될 경우어
                ));

        // Map 타입 지정
        Map<String, Integer> map4 = Stream.of("Apple", "Apple", "Tomato")
                .collect(Collectors.toMap(
                        name -> name, // key
```

```
                name -> name.length(), // value
                (oldValue, newValue) -> oldValue + newValue, // 중복될 경우에
                LinkedHashMap::new // 타입 지정 시 원하는 타입으로 변경
        ));
        System.out.println(map4.getClass());
    }
}
```

- toMap(keyMapper, valueMapper): 각 요소에 대한 키, 값을 지정해서 Map을 생성.
- 키가 중복되면 예외가 발생
- 병합을 통해 중복 키가 나오더라도 예외처리를 피할 수 있음.
- 마지막 인자를 통해 결과를 원하는 클래스로 생성 가능.

컬렉터 - 2

그룹과 분할 수집

Ex) 학생 데이터가 있을 때 반 별로 그룹화.

▼ 예제

```
package stream.collectors;

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

public class Collectors3Group {
    public static void main(String[] args) {
        // 첫 글자 알파벳을 기준으로 그룹화
        List<String> names = List.of("Apple", "Avocado", "Banana", "Blueberr
        Map<String, List<String>> grouped = names.stream()
                .collect(Collectors.groupingBy(name -> name.substring(0, 1)));
        System.out.println(grouped);
```

```
        // 짝수 여부로 분할(파티셔닝)
        List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
        Map<Boolean, List<Integer>> partitioned = numbers.stream()
                .collect(Collectors.partitioningBy(n -> n % 2 == 0));
        System.out.println(partitioned);
    }
}
```

- groupingBy는 특정 기준에 따라 스트림 요소를 여러 그룹으로 묶음.
  - Map<기준, List<요소>> 형태
- partitioningBy는 단순하게 true와 false 두 그룹으로 나눔.
  - Map<Boolean, List<요소>> 형태

최솟값 최댓값 수집

▼ 예제

```
package stream.collectors;

import java.util.List;
import java.util.Map;
import java.util.Optional;
import java.util.stream.Collectors;
import java.util.stream.IntStream;
import java.util.stream.Stream;

public class Collectors3MinMax {
    public static void main(String[] args) {
        // 다운스트림 컬렉터에서 유용하게 사용 가능.
        Integer max1 = Stream.of(1, 2, 3)
                .collect(Collectors.maxBy((o1, o2) -> o2 - o1))
                .get();
        System.out.println(max1);

        Integer max2 = Stream.of(4, 5, 6)
                .max((o1, o2) -> o1 - o2)
                .get();
        System.out.println(max2);
```

```
        Integer max3 = Stream.of(1, 2, 3)
                .max(Integer::compare)
                .get();
        System.out.println(max3);

        //기본형 특화 스트림
        int max4 = IntStream.rangeClosed(1, 6)
                .max()
                .getAsInt();
        System.out.println(max4);
    }
}
```

- Collectors.maxBy, minBy를 통해 최대 최소값을 구할 수 있음.
- 다만 스트림 자체가 제공하는 max(), min(), 기본형 특화 스트림(IntStream)을 사용 시 더 편리하나, 위처럼 사용하는 경우 다운스트림에서 더 유리

통계 수집

▼ 예제

```
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) {
        // 개수
        Long count1 = Stream.of(1, 2, 3)
                .collect(Collectors.counting());
        System.out.println(count1);

        long count2 = Stream.of(1, 2, 3, 4)
```

```
                .count();
        System.out.println(count2);

        Double average1 = Stream.of(1, 2, 2)
                .collect(Collectors.averagingInt(i -> i));
        System.out.println(average1);

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

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

        // 통계
        IntSummaryStatistics stats1 = Stream.of("Apple", "Banana", "Tomato"
                .collect(Collectors.summarizingInt(String::length));
        System.out.println(stats1.getCount());
        System.out.println(stats1.getSum());
        System.out.println(stats1.getMin());
        System.out.println(stats1.getMax());
        System.out.println(stats1.getAverage());
    }
}
```

리듀싱 수집

Collectors.reducing은 최종적으로 하나의 값으로 요소들을 합치는 방식을 지정.

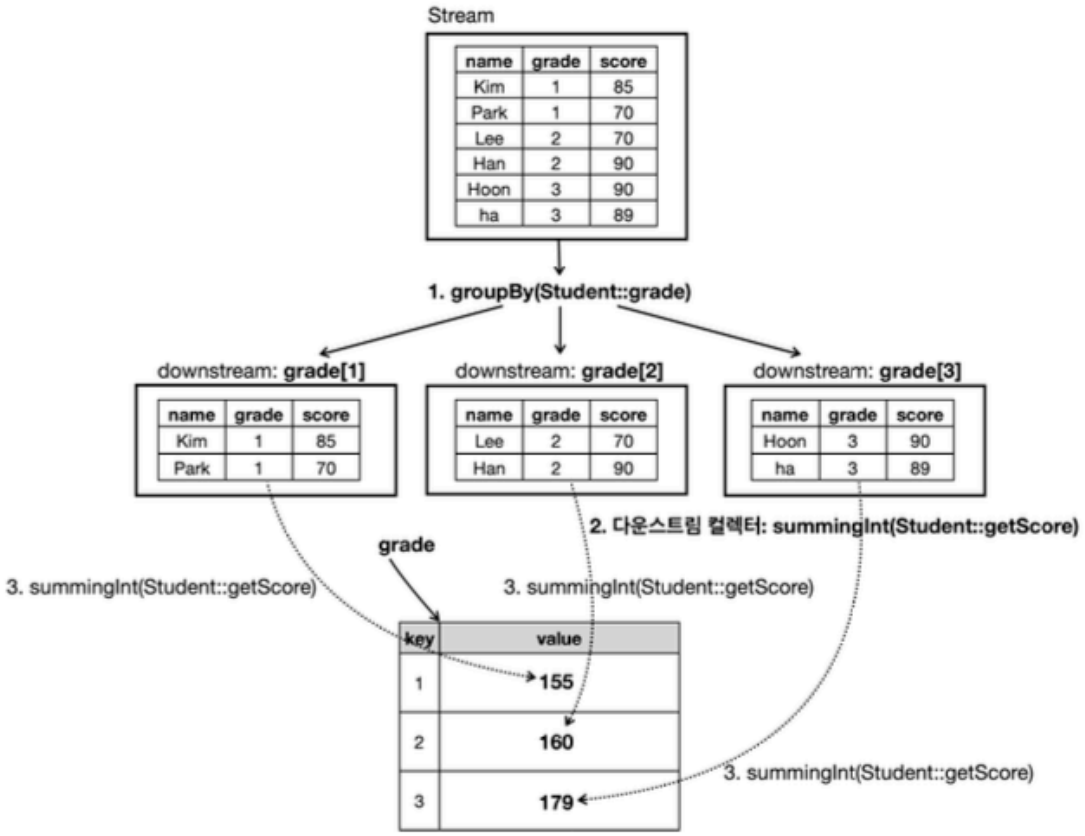
▼ 예제

```
package stream.collectors;

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

public class Collectors4Reducing {
    public static void main(String[] args) {
        List<String> names = List.of("a", "b", "c", "d");

        // Collection의 Reducing은 주로 다운 스트림에 활용
        String joined1 = names.stream()
                .collect(Collectors.reducing(
                        (s1, s2) -> s1 + "," + s2
                )).get();
        System.out.println(joined1);

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


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

        // String 객체 자체가 제공
        String joined4 = String.join(",", "a", "b", "c", "d");
        System.out.println(joined4);
    }
}
```

다운 스트림 컬렉터 - 1

다운 스트림 컬렉터가 필요한 이유

- groupingBy를 사용하면, 일단 요소가 그룹별로 묶이지만, 그룹 내 요소를 구체적으로 어떻게 처리할지는 기본적으로 toList()만 적용된다.
- 만약 '그룹별 총합, 평균, 최대/최소, 매핑 결과, 통계 등을 바로 얻고싶다면 ?
- 그룹화 된 이후 각 그룹 내부에서 추가적인 연산 또는 결과물 (Ex. 평균, 합계, 최댓값, 최솟값, 통계, 다른 타입으로 변환 등)을 정의하는 역할이 바로 **다운 스트림 컬렉터 (Downstream Collector)**



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

다운 스트림 컬렉터란 ?

- Collectors.groupingBy 또는 Collectors.partitioningBy에서 **두 번째 인자로 전달되는 Collector**
- 분류된 각 그룹 내부의 요소들을 다시 한 번 어떻게 처리할 지 정의하는 역할.

```
// 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
));
```

- 다운 스트림 컬렉터를 명시하지 않으면 toList()로 그룹별 요소들을 List화
- 다운 스트림 컬렉터는 그룹화(또는 분할)를 먼저 한 뒤, 각 그룹(또는 파티션) 내부의 요소들을 어떻게 처리할 것인가 ? 를 지정.

▼ 예제

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>

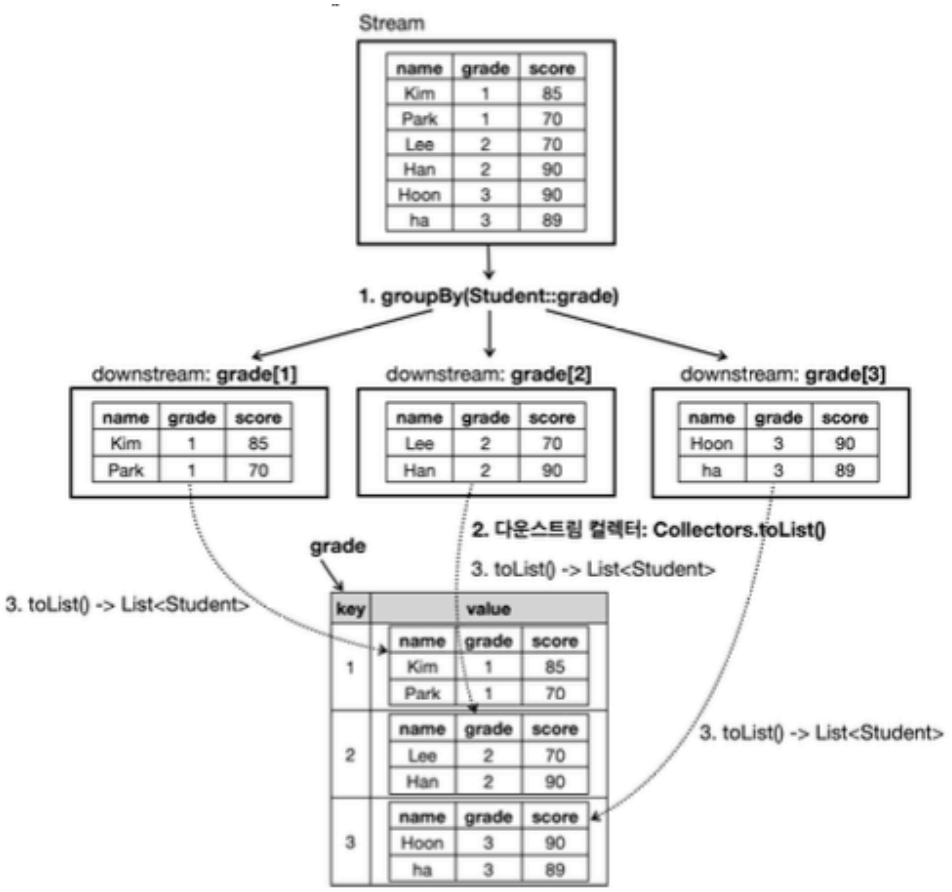
이 표는 다운 스트림 컬렉터의 대표적인 예시이다. `groupingBy(...)`, `partitioningBy(...)` 에서 두 번째 인자로 활용되거나, 스트림의 `collect()` 에서 직접 쓰이기도 한다.

```
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)
        );

        // 1. 학년 별 그룹화
        Map<Integer, List<Student>> collect1_1 = students.stream()
                .collect(Collectors.groupingBy(
                        Student::getGrade // 그룹화 기준 : 학년 (key 1,2,3 grade)
//                        ,Collectors.toList() // 생략 가능, 기본 toList()
                ));
        System.out.println(collect1_1);

        // 2. 학년 별 학생들의 이름
        Map<Integer, List<String>> collect1_2 = students.stream()
                .collect(Collectors.groupingBy(
                        Student::getGrade,
                        Collectors.mapping(Student::getName, // 다운 스트림 1 : 학생
                                Collectors.toList())
                ));
        System.out.println(collect1_2);

        // 3. 학년 별 학생 수
        Map<Integer, Long> collect1_3 = students.stream()
                .collect(Collectors.groupingBy(
                        Student::getGrade,
                        Collectors.counting()
                ));
        System.out.println(collect1_3);

        // 4. 학년 별 학생 평균 성적
        Map<Integer, Double> collect1_4 = students.stream()
```

```
            .collect(Collectors.groupingBy(
                    Student::getGrade,
                    Collectors.averagingInt(Student::getScore)
            ));
        System.out.println(collect1_4);
    }
}
```

1. 다운 스트림 컬렉터 - Collectors.toList()



`groupingBy(Student::getGrade)`

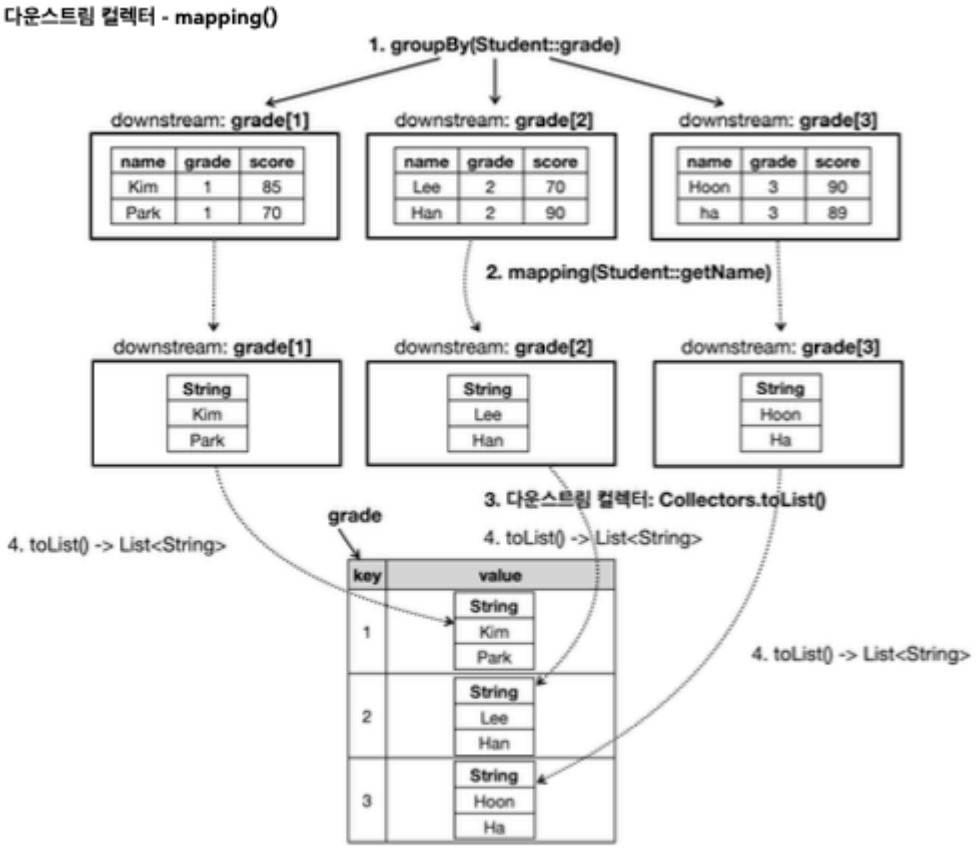
- 학년(grade)을 기준으로 학생(Student) 객체를 그룹화한다.

- **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.

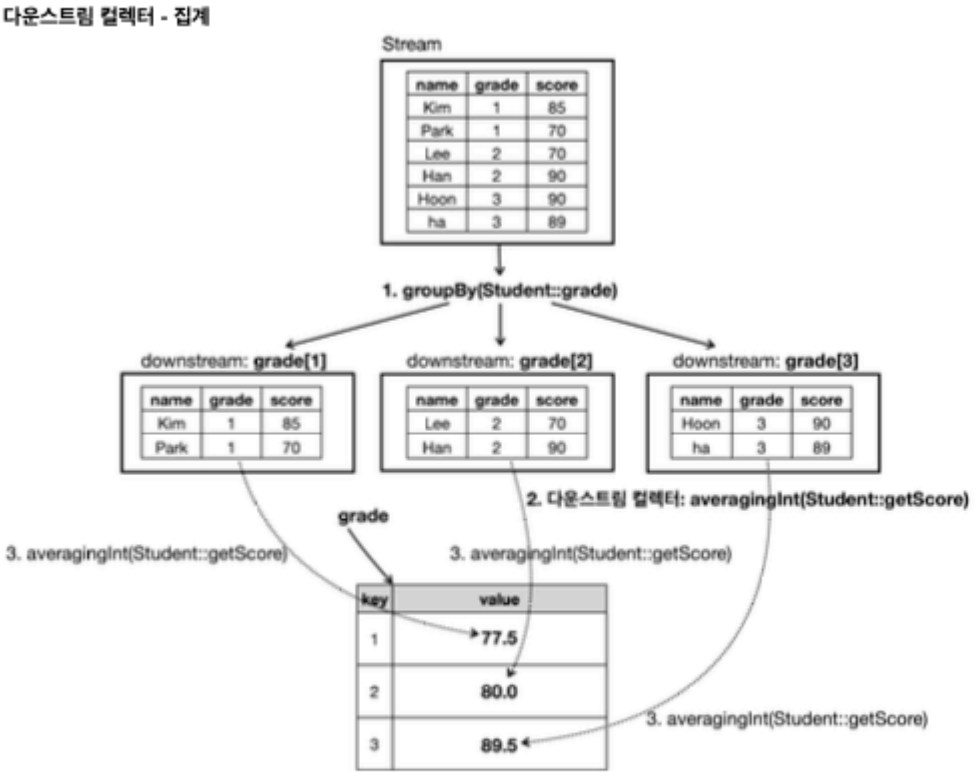
2. 다운 스트림 컬렉터 - mapping()




```
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.

3. 다운 스트림 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>`.

다운 스트림 컬렉터 - 2

▼ 예제

```
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)
        );

        // 1. 학년 별로 학생들 그룹화
        Map<Integer, List<Student>> collect1 = students.stream()
                .collect(Collectors.groupingBy(Student::getGrade));
        System.out.println(collect1);

        // 2. 학년 별로 가장 점수가 높은 학생, reducing
        Map<Integer, Optional<Student>> collect2 = students.stream()
                .collect(Collectors.groupingBy(
                        Student::getGrade
                        , Collectors.reducing((s1, s2) -> s1.getScore() > s2.getScore
                ));
        System.out.println(collect2);

        // 3. 학년 별로 가장 점수가 높은 학생, maxBy
        Map<Integer, Optional<Student>> collect3 = students.stream()
                .collect(Collectors.groupingBy(
                        Student::getGrade,
//                        Collectors.maxBy((s1, s2) -> s1.getScore() - s2.getScore())
//                        Collectors.maxBy(Comparator.comparingInt(Student::getS
                        Collectors.minBy(Comparator.comparingInt(Student::getSco
                ));
        System.out.println(collect3);

        // 4. 학년 별로 가장 점수가 높은 학생의 이름 (collectingAndThen + maxBy)
        // 학년 별 그룹 -> 그룹별 가장 점수가 높은 학생 -> 이름 매핑
        Map<Integer, String> collect4 = students.stream()
                .collect(Collectors.groupingBy(
                        Student::getGrade,
//                        Collectors.maxBy(Comparator.comparingInt(Student::getS
                        Collectors.collectingAndThen(
                                Collectors.maxBy(Comparator.comparingInt(Student::g
```

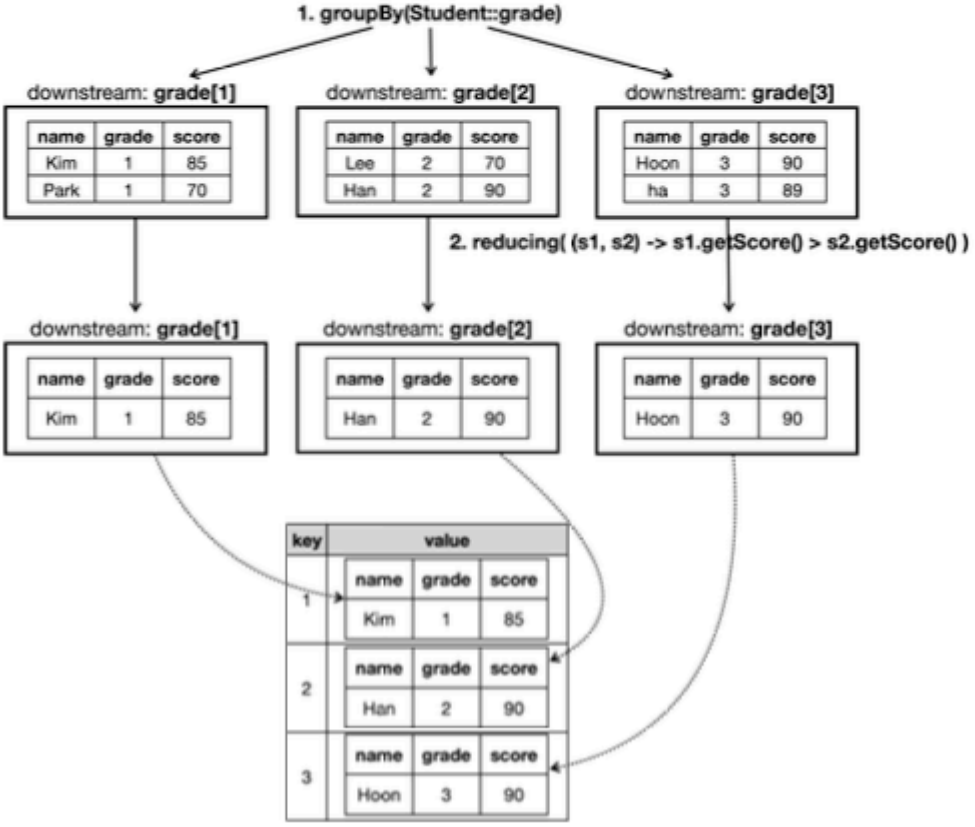
```
                        optionalStudent -> optionalStudent.get().getName()
                    )
            ));
        System.out.println(collect4);


    }

}
```

1. 학년별 학생 목록 (collect1)

학년별 학생 목록(collect1)

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

다운스트림 컬렉터 - 리듀싱



2. 학년 별 최대 점수 학생 구하기(reducing)

- Collectors.reducing()은 그룹 내부의 학생들을 하나씩 비교하며 축소(reduce)하는 로직 적

- `(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.

3. 학년 별 최대 점수 학생 구하기(maxBy)

- `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.

4. 학년 별 최대 점수 학생의 “이름”만 구하기
   - collectingAndThen은 다운 스트림 컬렉터가 만든 결과를 한 번 더 후처리.

- 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.

mapping() vs collectingAndThen()

- mapping : 그룹화 (또는 분할)된 각 그룹 내의 개별 요소들을 다른 값으로 변환한 뒤 그 변환된 값 들을 다시 다른 Collector로 수집할 수 있게 해줌.
- collectingAndThen() : 다운 스트림 컬렉터가 최종 결과를 만든 뒤에 한 번 더 후처리 1차 Collector → 후처리 함수 순서로 작업

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 통해 groupingBy(), partitioningBy()로 그룹화 / 분할을 한 뒤 내부 요소를 어떻게 가공하고 수집할 지 자유롭게 설계.
  - mapping(), counting(), summarizingInt(), reducing(), maxBy(), minBy(), summingInt(), averagingInt() 등 다양한 Collector 메서드를 조합하여 복잡한 요구사항도 한 번의 파이프라인으로 처리 가능.

정리

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.

- 가독성, 선연형 코드, 지연 연산에 따른 최적화

- 간단한 데이터 필터링, 변환 ~ 대규모 그룹화 / 집계처리까지 여러 복잡한 반복문 없이 직관적인 코드 작성 가능.