

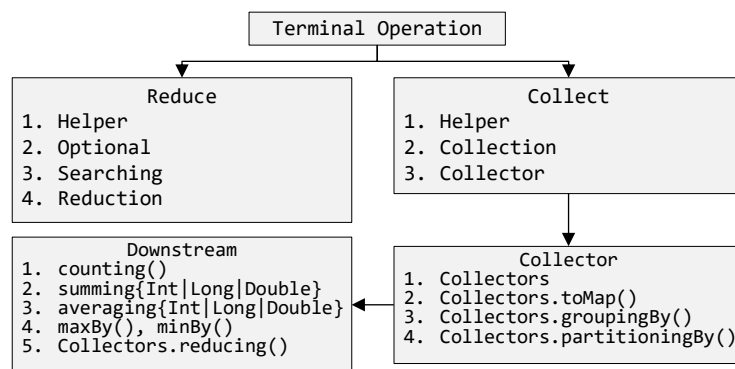
Grouping

sanghyuck.na@lge.com

```
static class Issue implements Comparable<Issue> {String id;int
priority;public Issue(String id, int priority) {this.id =
id;this.priority = priority;}static Issue of(String id, int priority)
{return new Issue(id, priority);}public String getId() {return
id;}public int getPriority() {return priority;}@Override public int
compareTo(Issue o) {return id.compareToIgnoreCase(o.id);}} List<Issue>
newIssueList() {return List.of(Issue.of("JIRA-51", 2), Issue.of("JIRA-
52", 1), Issue.of("JIRA-53", 2), Issue.of("JIRA-54", 3));}
```

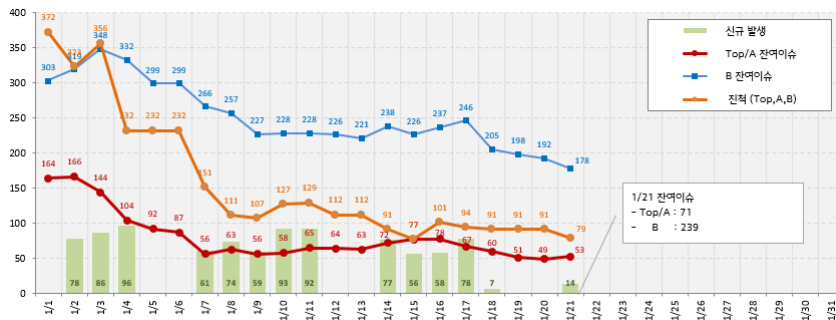
1

목차



2

Issue Grouping



ID	Priority
JIRA-51	2
JIRA-52	1
JIRA-53	2
JIRA-54	3



Priority	Count
1	1
2	2
3	1

3

groupingBy

- 주어진 함수로 결정된 key로 그룹화(Map 반환)하는 Collector
 - classifier: 주어진 데이터를 입력으로 하고 key를 반환하는 Mapping 함수
 - downstream: Downstream Reduction을 구현하는 Collector
 - mapFactory: 최종 결과를 담은 새로운 Map 생성 함수

```
Collector<T,?,Map<K,List<T>>>
    groupingBy(Function<? super T, ? extends K> classifier)
Collector<T,?,Map<K,D>>
    groupingBy(Function<? super T, ? extends K> classifier
        , Collector<? super T,A,D> downstream)
Collector<T,?,M>
    groupingBy(Function<? super T, ? extends K> classifier
        , Supplier<M> mapFactory
        , Collector<? super T,A,D> downstream)
```


4

Classifier

- Function<? **super** T, ? **extends** K> classifier
 - 입력 데이터타입 T를 반환타입 K로 구하는 함수로 Map의 Key가 되고 본래 Stream값은 Value전환

```
<T,K> Collector<T, ?, Map<K,List<T>>>
  groupingBy(Function<? super T, ? extends K> classifier)
```

```
Map<Integer, List<Issue>> g1 = newIssueList().stream()
  .collect(Collectors.groupingBy(Issue::getPriority));
```

ID	Priority		Priority	List<Issue>
JIRA-51	2		1	[JIRA-52=1]
JIRA-52	1		2	[JIRA-51=2, JIRA-53=2]
JIRA-53	2		3	[JIRA-54=3]
JIRA-54	3			


5

Downstream

- Collector<? **super** T, A, D> downstream
 - 최종 결과값 Map의 Value이 되는 부분집계결과로, Downstream Collector란 그 집계결과를 만들어내는 Collector
 - 부모에서 downstream은 입력데이터타입 T, 중간집계결과타입 A를 받아 최종결과타입 D를 만들어 부모 Map의 Value로 타입정의

```
<T,K,A,D> Collector<T, ?, Map<K,D>>
  groupingBy(Function<? super T, ? extends K> classifier,
    Collector<? super T,A,D> downstream)
```

```
Map<Integer, List<Issue>> g2 = newIssueList().stream()
  .collect(Collectors.groupingBy(Issue::getPriority,
    Collectors.toList()));
```

ID	Priority		Priority	List<Issue>
JIRA-51	2		1	[JIRA-52=1]
JIRA-52	1		2	[JIRA-51=2, JIRA-53=2]
JIRA-53	2		3	[JIRA-54=3]
JIRA-54	3			

6

Downstream

- 중첩 반복문에서 집계 내부에 또다른 집계동작
 - Downstream Collector는 특수타입이 아닙니다. 일반적인 Collector입니다.

```
Collectors.groupingBy(Issue::getPriority, Collectors.toList())
(Up)Stream(1st): Map<Integer , >
Downstream(2nd): List<Issue>
```

```
Map<Integer, List<Entry>> map = new Map<>();
for (T e: t) {
    K key = classifier.accept(e);
    D lst = map.get(key);
    if (lst == null)
        lst = down.supplier().get();
    for(T ee : e)
        down.accumulator().accept(lst, v);
    down.combiner().apply(?, lst);
    map.put(key, collector.finisher(lst));
}
```

7

Map Factory

- Supplier<M> mapFactory
 - 최종결과타입과 중간결과타입 지정

```
<T,K,D,A,M extends Map<K,D>> Collector<T, ?, M>
groupingBy(Function<? super T, ? extends K> classifier,
Supplier<M> mapFactory,
Collector<? super T,A,D> downstream)
```

```
TreeMap<Integer, Set<Issue>> g5 = newIssueList().stream()
.collect(Collectors.groupingBy(Issue::getPriority,
TreeMap::new, Collectors.toSet()));
```

ID	Priority
JIRA-51	2
JIRA-52	1
JIRA-53	2
JIRA-54	3



Priority	Set<Issue>
1	[JIRA-52=1]
2	[JIRA-51=2, JIRA-53=2]
3	[JIRA-54=3]

8

도전하세요!

- 몇 번째 `groupingBy`를 사용하면 결과타입을 구할 수 있나요?
- `groupingBy`절을 완성하세요

```
Map<Integer, TreeSet<Issue>> g6 = newIssueList().stream()
    .collect(Collectors.groupingBy(?));

HashMap<Integer, PriorityQueue<Issue>> g7 = newIssueList()
    .stream().collect(Collectors.groupingBy(?));
```

9

groupingByConcurrent

- 동시 집계
 - Unordered: 교환적(Associative)이고 결합적인(Commutative) 연산으로 동작
 - Concurrent: Accumulator 연산에서 사용되는 임시중간연산결과 저장소는 일부 연산자 간 공유되어 사용

```
Collector<T,?, ConcurrentMap<K, List<T>>>
    groupingByConcurrent(Function<? super T,? extends K> classifier)

Collector<T,?, ConcurrentMap<K,D>>
    groupingByConcurrent(Function<? super T,? extends K> classifier,
        Collector<? super T,A,D> downstream)

Collector<T,?,M>
    groupingByConcurrent(Function<? super T,? extends K> classifier,
        Supplier<M> mapFactory,
        Collector<? super T,A,D> downstream)
```

10

Downstream – toCollection

- Connection 타입지정
 - T: 입력 데이터 타입, C: 반환 데이터 타입
 - Collector 타입지정은 Downstream 결과타입지정에 자주 출현

```
<T,C extends Collection<T>> Collector<T, ?, C>
    toCollection(Supplier<C> collectionFactory)
```

```
Map<Integer, ArrayDeque<Issue>> g7 = newIssueList()
    .stream().collect(Collectors.groupingBy(
        Issue::getPriority,
        Collectors.toCollection(ArrayDeque::new)));
```

ID	Priority
JIRA-51	2
JIRA-52	1
JIRA-53	2
JIRA-54	3



Priority	ArrayDeque<Issue>
1	[JIRA-52=1]
2	[JIRA-51=2, JIRA-53=2]
3	[JIRA-54=3]

11

Downstream – mapping

- 입력변환 Collector
 - 입력 타입 T를 타입 U로 변환
 - Downstream 입력타입지정에 자주 사용

```
Collector<T,?,R> mapping
    (Function<? super T,? extends U> mapper,
     Collector<? super U,A,R> downstream)
```

ID	Priority
JIRA-51	2
JIRA-52	1
JIRA-53	2
JIRA-54	3



Priority	LinkedList<String>
1	[JIRA-52]
2	[JIRA-51, JIRA-53]
3	[JIRA-54]

Issue: T

U:String
(mapper)R:LinkedList<String>
(downstream)

13

Downstream

- Map: groupingBy
 - Pipeline Output Type 지정
- String: mapping
 - DownStream Input Type T 지정
- HashSet: toCollection
 - Downstream Output Type 지정

```
Map<Integer, LinkedList<String>> g8 = newIssueList().stream().
    collect(Collectors.groupingBy(Issue::getPriority,
        Collectors.mapping(Issue::getId,
            Collectors.toCollection(LinkedList::new))));
```

14

Downstream – collectingAndThen

- 출력변환 Collector
 - 결과타입R을 타입RR로 변환
 - Downstream 결과타입지정에 빈번히 사용

```
<T,A,R,RR> Collector<T,A,RR> collectingAndThen(
    Collector<T,A,R> downstream, Function<R,RR> finisher)
```

ID	Priority
JIRA-51	2
JIRA-52	1
JIRA-53	2
JIRA-54	3



Priority	LinkedList<String>
1	JIRA-52
2	JIRA-51, JIRA-53
3	JIRA-54

{Key, list.get(0)}



Priority	ID
1	JIRA-52
2	JIRA-51
3	JIRA-54

16

Multilevel reduction

- 한 개 이상 Downstream collector 포함
- Inbound operator
 - groupingBy(), mapping(), toCollection()
- Outbound operator
 - collectingAndThen()

```
Map<Integer, String> g12 = newIssueList().stream().collect(
    Collectors.groupingBy(Issue::getPriority,
        Collectors.collectingAndThen(
            Collectors.mapping(Issue -> String
                Issue::getId,
                Collectors.toCollection(
                    List<Issue> -> LinkedList<Issue>
                    LinkedList::new)),
            (v) -> v.get(0)
        ) LinkedList<Issue> -> String
    ));
```


17

partitioningBy

- predicate에 따라 두개로 분류하는 Collector
 - Predicate: 입력값을 True 또는 False로 분류하는 조건
 - Downstream

```
<T> Collector<T,?,Map<Boolean,List<T>>>
    partitioningBy(Predicate<? super T> predicate)
<T,D,A> Collector<T,?,Map<Boolean,D>>
    partitioningBy(Predicate<? super T> predicate
        , Collector<? super T,A,D> downstream)
```

```
Predicate<Issue> predicate = t -> t.getPriority() <= 1;
Map<Boolean, List<Issue>> m = newIssueList()
    .stream().collect(Collectors.partitioningBy(predicate));
```

ID	Priority		Boolean	List<Issue>
JIRA-51	2		false	[JIRA-51=2, JIRA-53=2, JIRA-54=3]
JIRA-52	1		true	[JIRA-52=1]
JIRA-53	2			
JIRA-54	3			

Priority <= 1

18

도전 하세요!

- 다음 partitioningBy() 구문을 GroupingBy로 바꿔보세요.

```
Predicate<Issue> predicate = t -> t.getPriority() <= 1;

Map<Boolean, List<Issue>> m = newIssueList()
    .stream().collect(
        Collectors.partitioningBy(predicate));

System.out.println(m);
```

19

정리

- groupingBy()
 - groupingByConcurrent()
- Downstream
 - toCollection
 - mapping
 - collectingAndThen
- partitioningBy()
 - predicate

20

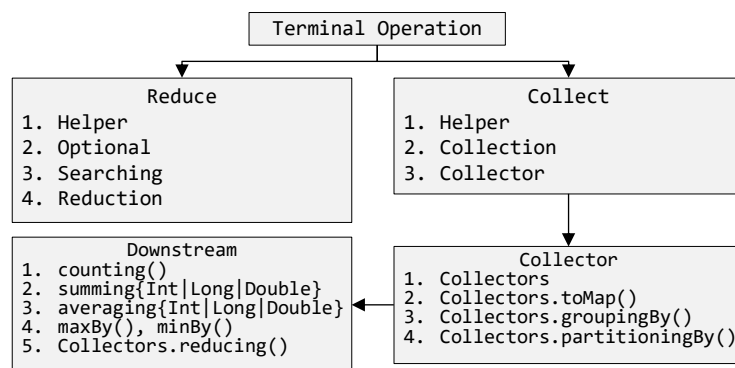
Downstream Reducing

sanghyuck.na@lge.com

```
static class Product {String id;int price;public Product(String id, int price) {this.id = id;this.price = price;}static Product of(String id, int price) {return new Product(id, price);}public String getId() {return id;}public int getPrice() {return price;}@Override public String toString() {return "Product [id=" + id + ", price=" + price + "]}";}List<Product> newProductList() {return List.of(Product.of("A", 100),Product.of("B", 200), Product.of("B", 300),Product.of("C", 400));}}
```

21

목차



22

정리

	Terminal operators
1	reduce()
2	collect()
	Collector
1	toList(), toSet(), toMap()
2	counting(), maxBy(), minBy(...) summing(Int Long Double)() averaging(Int Long Double)() summarizing(Int Long Double)()
	groupingBy(), partitioningBy()
3	mapping()
4	toCollection()
5	collectingAndThen()
6	reducing()

23

reduce vs reducing

- Downstream의 집계연산
 - 최종 결과값의 집계연산은 reduce() 사용한다면, 그 내부에 적용되는 집계연산
 - Downstream 집계연산은 1번 이상 Pipeline에 적용될 수 있습니다(Multi-level reduction)

```
Collector<T,?,Optional<T>> reducing (BinaryOperator<T> op)
Collector<T,?,T> reducing (T identity, BinaryOperator<T> op)
Collector<T,?,U> reducing (U identity,
    Function<? super T,? extends U> mapper,
    BinaryOperator<U> op)
```

```
Optional<T> reduce (BinaryOperator<T> accumulator)
T reduce (T identity, BinaryOperator<T> acc)
U reduce (U identity,
    BiFunction<U,? super T,U> accumulator,
    BinaryOperator<U> combiner)
```

24

Reducing Helper

- 자주 사용하는 Reducing 연산자

```
<T> Collector<T,?,Long> counting()

<T> Collector<T,?,Integer> summing{Int|Long|Double}
    (To{Int|Long|Double}Function<? super T> mapper)
<T> Collector<T,?,Integer> averaging{Int|Long|Double}
    (To{Int|Long|Double}Function<? super T> mapper)

<T> Collector<T,?,Optional<T>>
    maxBy(Comparator<? super T> comparator)
<T> Collector<T,?,Optional<T>>
    minBy(Comparator<? super T> comparator)
```

```
Map<String, Long> r = newEntryStream()
    .collect(Collectors.groupingBy(Map.Entry::getKey,
        Collectors.counting()));
```

25

Reducing

- groupingBy나 partitioningBy의 downstream collector로 사용
 - Identity: Stream.reduce 연산과 같이, identity는 reduction 연산의 초기 값을 의미하며 아무런 데이터가 없는 경우 기본값으로 사용
 - BinaryOperator: 모든 데이터에 적용되는 기본단위함수로 2개를 1개 값으로 집계
 - Mapper: Reduction 대상 값을 변환하는 함수

```
Collector<T,?,Optional<T>> reducing
    (BinaryOperator<T> op)

Collector<T,?,T> reducing (T identity,
    BinaryOperator<T> op)

Collector<T,?,U> reducing(U identity
    , Function<? super T,? extends U> mapper
    , BinaryOperator<U> op)
```

26

Binary Operator

- 집계 연산자
 - 입력데이터 타입에 대한 집계연산정의
 - Collector의 Finisher타입은 Optional<T>
 - Min, Max, Average같이 초기값이 미 정의된 연산에 사용

```
<T> Collector<T,?,Optional<T>>
    reducing(BinaryOperator<T> op)
```

ID	Price
A	100
B	200
B	300
C	400



ID	Max
A	{A,100}
B	{B,300}
C	{C,400}

Map<String, Optional<Product>>

27

Identity

- 초기값
 - 입력데이터 타입의 집계에 대한 초기값을 정의
 - Sum, Count이 초기값이 정의된 연산에 사용

```
<T> Collector<T,?,T>
    reducing(T identity, BinaryOperator<T> op)
```

ID	Price
A	100
B	200
B	300
C	400



ID	Count
A	1
B	2
C	1

Map<String, Long>

28

Mapper

- 집계대상 변환
 - 입력데이터타입T를 집계대상 데이터 타입U로 변환
 - `Collectors.mapping()`기능을 하나로 제공

```
<T,U> Collector<T,?,U> reducing(U identity
    , Function<? super T,? extends U> mapper
    , BinaryOperator<U> op)
```

ID	Price
A	100
B	200
B	300
C	400



ID	Count
A	1
B	2
C	1

Map<String, Long>

29

도전 하세요!

- 모든아이템 5, 10, 20, 50을 하나의 문자열로 결합(concatenation)하려고 합니다. 다음 ?을 완성하세요
 - 단 `Collectors.reducin`을 꼭 사용하세요!

```
List<Integer> lst = List.of(5, 10, 20, 50);

String out = lst.stream().collect( ? );

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

```
5102050
```

30

정리

- Downstream reduction
 - Collectors helper method
 - Collectors.reducing()

