Chapter 06.
스트림으로 데이터 수집

2019. 09. 19 | Java in Action Seminar | 박대원

wisoft

이 장의 내용

- Collectors 클래스로 컬렉션을 만들고 사용하기
- 하나의 값으로 데이터 스트림 리듀스하기
- 특별한 리듀싱 요약 연산
- 데이터 그룹화와 분할
- 자신만의 커스텀 컬렉터 개발

1.

미리보기

```
words.stream()
  .map(word -> word.split(""))
  .flatMap(Arrays::stream)
  .collect(toList());
```

```
Optional<Integer> sum = numbers.stream()
  .reduce((a, b) -> (a + b));
```

Q. 통화별로 트랜잭션 리스트를 그룹화 하시오.

2.

컬렉터란 무엇인가?

```
1  Map<Currency, List<Transaction>> transactionsByCurrencies =
2    transactions.stream()
3    .collect(groupingBy(Transaction::getCurrency));
```

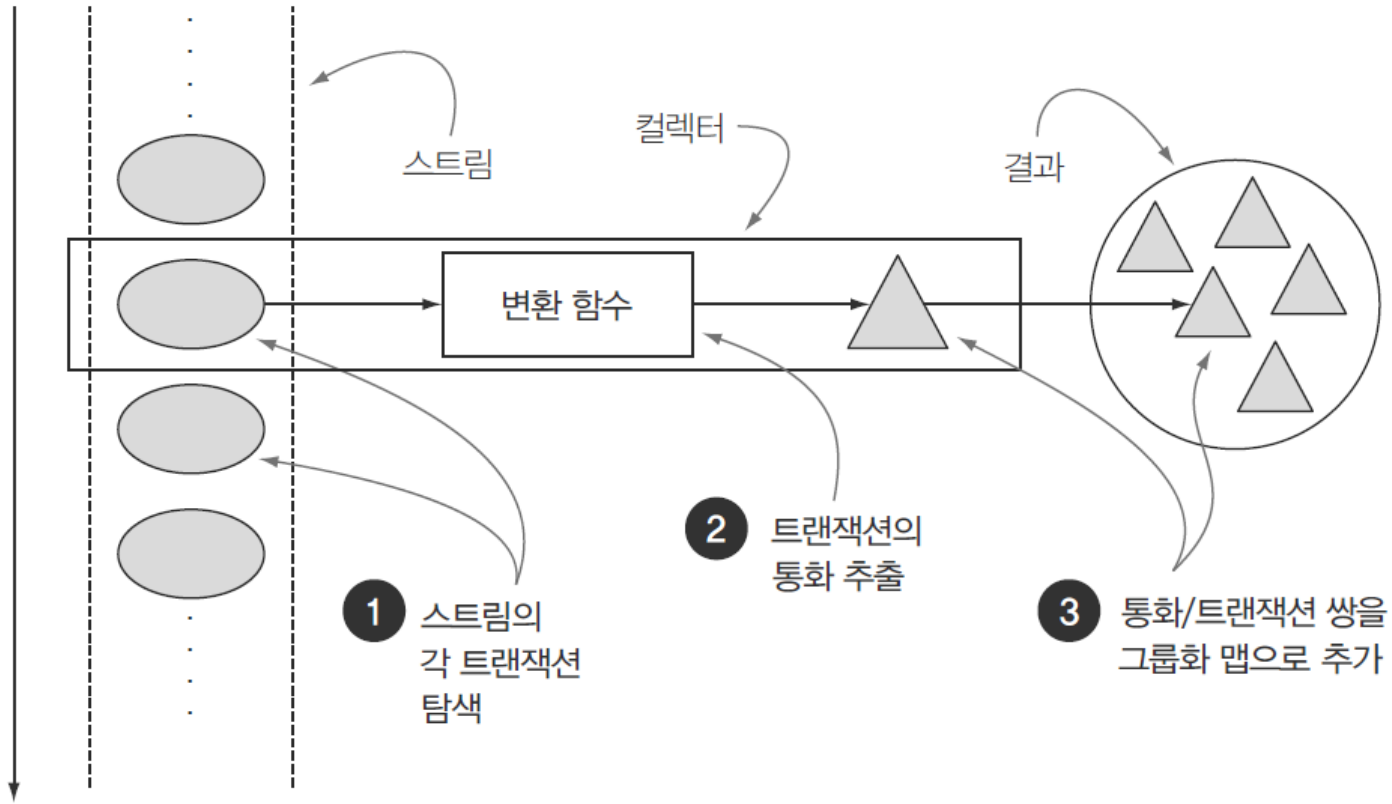
groupImperatively(); < *groupFunctionally*();

2.

컬렉터란 무엇인가?

고급 리듀싱 기능을 수행하는 컬렉터

그림 6-1 통화별로 트랜잭션을 그룹화하는 리듀싱 연산



2.

컬렉터란 무엇인가?

미리 정의된 컬렉터

1. 스트림 요소를 하나의 값으로 리듀스하고 요약

2. 요소 그룹화

3. 요소 분할

2.

리듀싱과 요약

counting()

```
long howManyDishes = menu.stream().count();
```

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

```
Collectors.counting() -> counting()
```

2.

리듀싱과 요약

스트림값에서 최댓값과 최솟값 검색

```
1  Comparator<Dish> dishCaloriesComparator =
2    Comparator.comparingInt(Dish::getCalories);
3
4  Optional<Dish> mostCalorieDish =
5    menu.stream()
6    .collect(maxBy(dishCaloriesComparator));
```

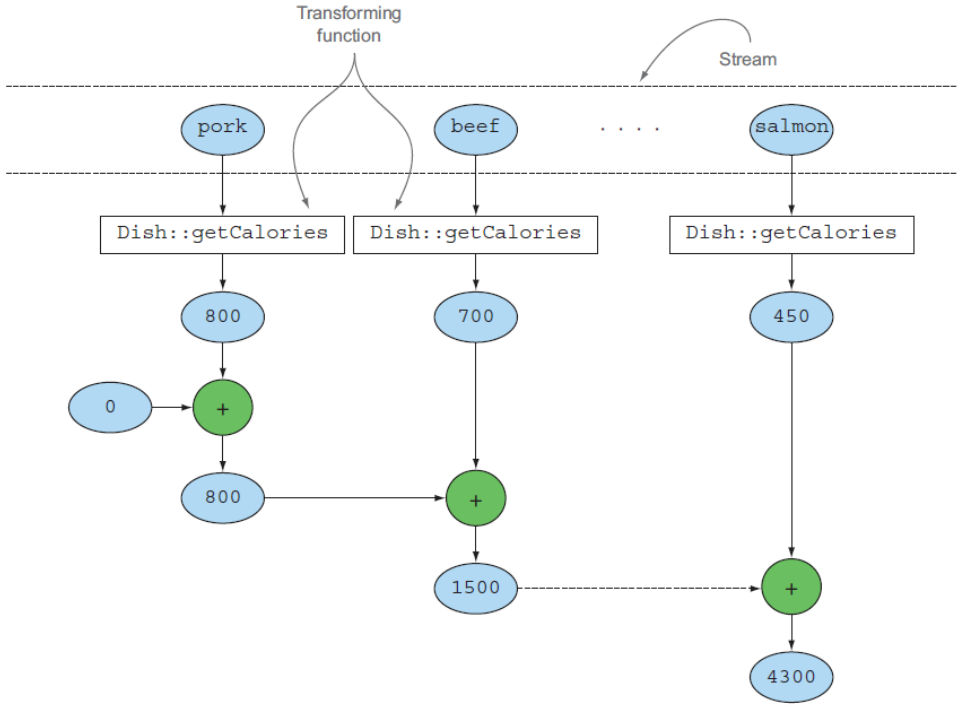
이처럼 스트림에 있는 객체의 숫자 필드의 합계나 평균 등을 반환하는 연산에도 리듀싱이 자주 사용되는데, 이런 연산을 **요약**summarization **연산**이라 부른다.

2.

리듀싱과 요약

요약 연산

```
1  int totalCalories = menu.stream().collect(summingInt(Dish::getCalories));
```




```
1  IntSummaryStatistics menuStatistics =
2    menu.stream().collect(summarizingInt(Dish::getCalories));
```

```
IntSummaryStatistics{count=9, sum=4300, min=120, average=477.777778, max=800}
```

2.

리듀싱과 요약

문자열 연결

```
1 String shortMenu = menu.stream().collect(joining());
```

```
Short menu: porkbeefchickenfrench friesriceseason fruitpizzaprawnssalmon
```

```
1 String shortMenu = menu.stream().map(Dish::getName).collect(joining(", "));
```

```
Short menu comma separated: pork, beef, chicken, french fries, rice, season fruit, pizza, prawns, salmon
```

2.

리듀싱과 요약

범용 리듀싱 요약 연산

```
int totalCalories = menu.stream().collect(
  reducing(0, Dish::getCalories, (i, j) -> i + j));
```

1. 첫 번째 인수는 리듀싱 연산의 시작값이거나 스트림에 인수가 없을 때의 반환값이다.

2.두 번째 인수는 요리를 칼로리 정수로 변환할 때 사용한 변환 함수이다.

3. 세 번째 인수는 같은 종류의 두 항목을 하나의 값으로 더하는 BinaryOperator다.

2.

리듀싱과 요약

퀴즈

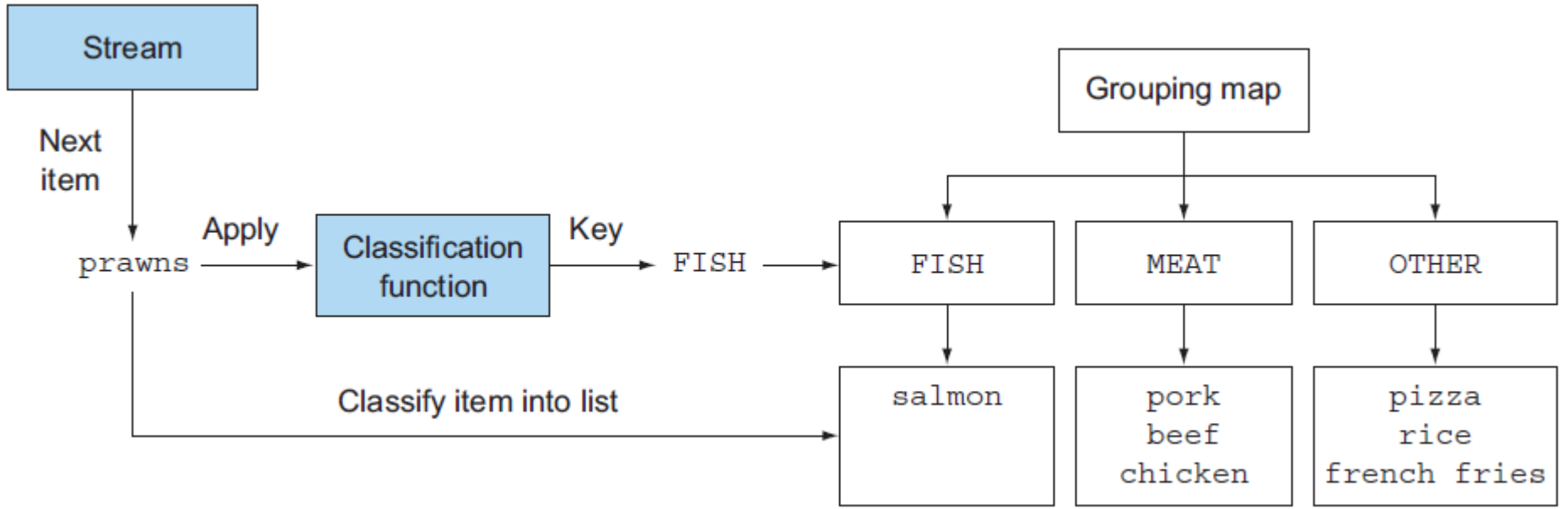
```
String shortMenu = menu.stream().map(Dish::getName).collect(joining());

1 String shortMenu = menu.stream().map(Dish::getName)
    .collect( reducing( (s1, s2) -> s1 + s2 ) ).get();

2 String shortMenu = menu.stream()
    .collect( reducing( (d1, d2) -> d1.getName() + d2.getName() )).get();

3 String shortMenu = menu.stream()
    .collect( reducing( "", Dish::getName, (s1, s2) -> s1 + s2 ) );
```

3.

그룹화

```
1  Map<Dish.Type, List<Dish>> dishesByType =
2    menu.stream().collect(groupingBy(Dish::getType));
```

```
Dishes grouped by type: {MEAT=[pork, beef, chicken], OTHER=[french fries, rice, season fruit, pizza], FISH=[prawns, salmon]}
```



3.

그룹화

다수준 그룹화

Listing 6.2 Multilevel grouping

```
Map<Dish.Type, Map<CaloricLevel, List<Dish>>> dishesByTypeCaloricLevel =
menu.stream().collect(
      groupingBy(Dish::getType,
        groupingBy(dish -> {
                if (dish.getCalories() <= 400) return CaloricLevel.DIET;
                else if (dish.getCalories() <= 700) return CaloricLevel.NORMAL;
                else return CaloricLevel.FAT;
            } )
      )
);
```

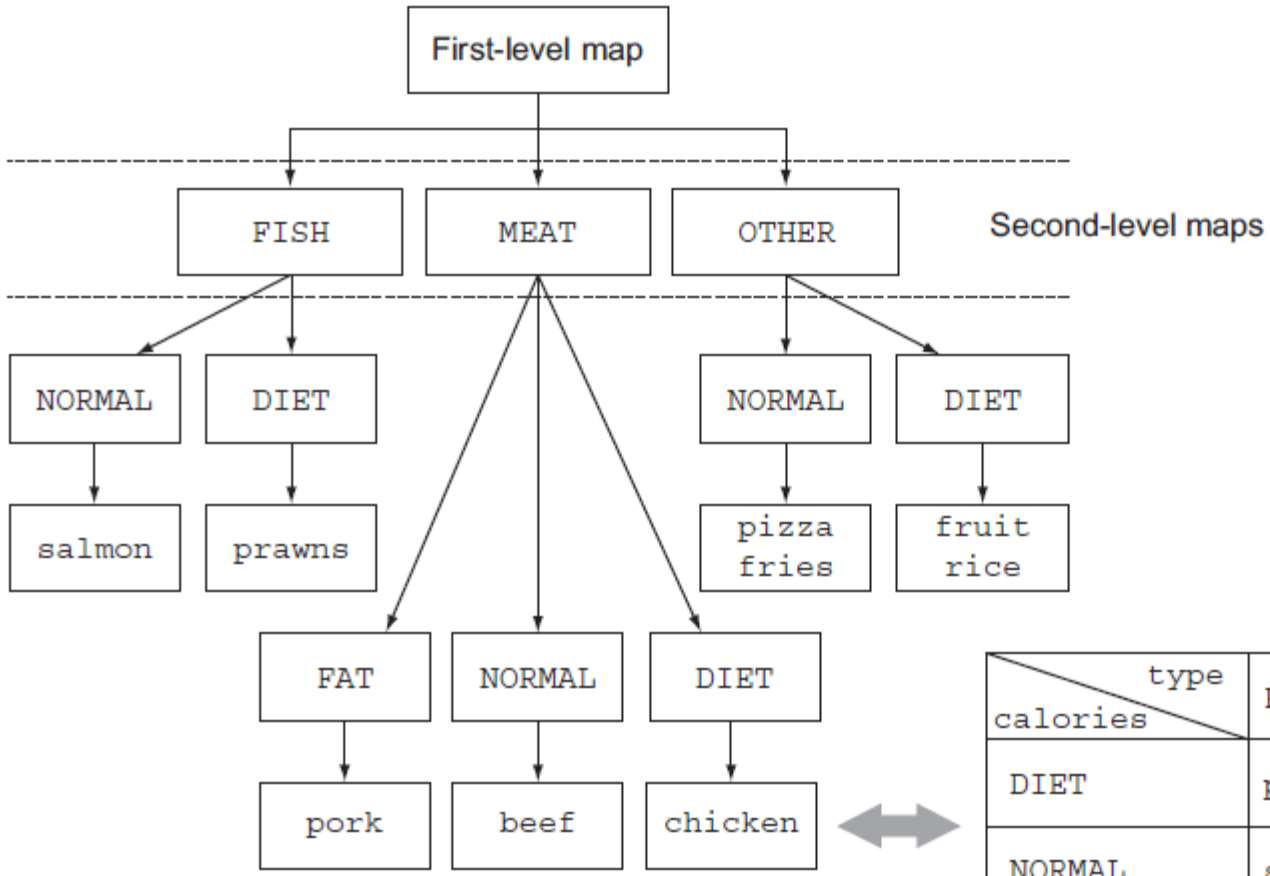
First-level classification function

Second-level classification function

```
{MEAT={DIET=[chicken], NORMAL=[beef], FAT=[pork]},
FISH={DIET=[prawns], NORMAL=[salmon]},
OTHER={DIET=[rice, seasonal fruit], NORMAL=[french fries, pizza]}}
```

3.

그룹화



type / calories	FISH	MEAT	OTHER
DIET	prawns	chicken	fruit rice
NORMAL	salmon	beef	pizza fries
FAT		pork	

3.

그룹화

컬렉터 결과를 다른 형식에 적용하기

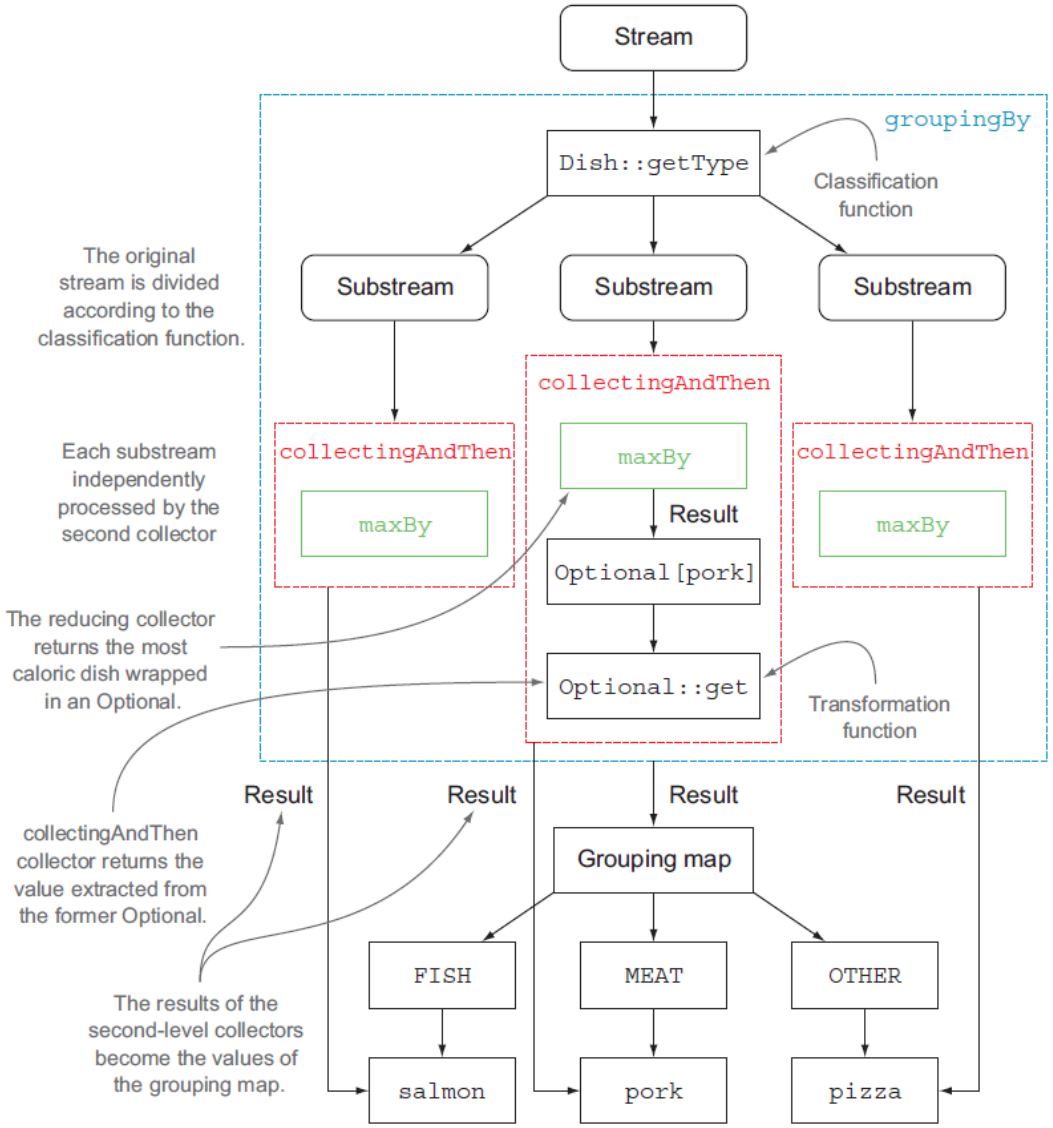
```
1  Map<Dish.Type, Dish> mostCaloricByType =
2    menu.stream()
3    .collect(groupingBy(Dish::getType,
4    collectingAndThen(maxBy(comparingInt(Dish::getCalories)), Optional::get)));
```

{FISH=Optional[salmon], OTHER=Optional[pizza], MEAT=Optional[pork]}

{FISH=salmon, OTHER=pizza, MEAT=pork}

3.

그룹화


Stream
groupingBy
Dish::getType
Classification function
The original stream is divided according to the classification function.
Substream
Substream
Substream
collectingAndThen
collectingAndThen
maxBy
collectingAndThen
Each substream independently processed by the second collector
maxBy
Result
maxBy
Optional[pork]
The reducing collector returns the most caloric dish wrapped in an Optional.
Optional::get
Transformation function
Result
Result
Result
Result
collectingAndThen collector returns the value extracted from the former Optional.
Grouping map
FISH
MEAT
OTHER
The results of the second-level collectors become the values of the grouping map.
salmon
pork
pizza

4.

분할

```
1 Map<Boolean, List<Dish>> partitionedMenu =
2   menu.stream().collect(partitioningBy(Dish::isVegetarian));
```

```
{false=[pork, beef, chicken, prawns, salmon],
true=[french fries, rice, season fruit, pizza]}
```

```
1 List<Dish> vegetarianDishes = partitionedMenu.get(true);
```

```
1 List<Dish> vegetarianDishes =
2 menu.stream().filter(Dish::isVegetarian).collect(toList());
```

4.

분할

분할의 장점

분할 함수가 반환하는 참, 거짓 두가지 요소의 스트림 리스트를 모두 유지한다는 것이 분할의 장점이다.

```
Map<Boolean, Map<Dish.Type, List<Dish>>> vegetarianDishesByType =
  menu.stream().collect(
                partitioningBy(Dish::isVegetarian,
                               groupingBy(Dish::getType)));
```

```
{false={FISH=[prawns, salmon], MEAT=[pork, beef, chicken]},
true={OTHER=[french fries, rice, season fruit, pizza]}}
```

```
Map<Boolean, Dish> mostCaloricPartitionedByVegetarian =
  menu.stream().collect(
    partitioningBy(Dish::isVegetarian,
      collectingAndThen(maxBy(comparingInt(Dish::getCalories)),
                        Optional::get)));
```

```
{false=pork, true=pizza}
```

4.

분할

퀴즈

```
1 menu.stream().collect(partitioningBy(Dish::isVegetarian,
    partitioningBy(d -> d.getCalories() > 500)));
```

```
{ false={false=[chicken, prawns, salmon], true=[pork, beef]},
true={false=[rice, season fruit], true=[french fries, pizza]}}
```

```
2 menu.stream().collect(partitioningBy(Dish::isVegetarian,
    partitioningBy(Dish::getType)));
```

partioningBy는 불리언을 반환하는 프레디케이트를 요구하므로
컴파일 되지 않는다. Dish::getType은 프레디케이트로 사용할 수 없다.

```
3 menu.stream().collect(partitioningBy(Dish::isVegetarian,
    counting()));
```

```
{false=5, true=4}
```

4.

정리

팩토리 메서드	반환 형식	사용 예제
toList	List<T>	스트림의 모든 항목을 리스트로 수집
Ex : List<Dish> dishes = menuStream.collect(toList());		
toSet	Set<T>	스트림의 모든 항목을 중복이 없는 집합으로 수집
Ex : Set<Dish> dishes = menuStream.collect(toSet());		
toCollection	Collection<T>	스트림의 모든 항목을 발행자가 제공하는 컬렉션으로 수집
Ex : Collection<Dish> dishes = menuStream.collect(toCollection(), ArrayList::new);		
counting	Long	스트림의 항목 수 계산
Ex : long howManyDishes = menuStream.collect(counting());		
summingInt	Integer	스트림 항목의 정수 프로퍼티의 값을 더함
Ex : int totalCalories = menuStream.collect(summingInt(Dish::getCalories));		
averagingInt	Double	스트림 항목의 정수 프로퍼티의 평균값 계산
Ex : double avgCalories = menuStream.collect(averagingInt(Dish::getCalories));		
summarizingInt	IntSummary Statistics	스트림 내 항목의 최댓값, 최솟값, 합계, 평균 등의 정수 정보 통계 수집
Ex : IntSummaryStatistics menuStatistics = menuStream.collect(summaryInt(Dish::getCalories));		

4. 정리

팩토리 메서드	반환 형식	사용 예제
joining	String	스트림의 각 항목에 toString 메서드를 호출한 결과 문자열 연결
Ex : String shortMenu = menuStream.map(Dish::getName).collect(joining(“, “);		
maxBy, minBy	Optional<T>	주어진 비교자를 이용해 스트림의 최대값, 최소값 요소를 Optional<T>로 반환.
Ex : Optional<Dish> fattest = menuStream.collect(maxBy(comparingInt(Dish::getCalories)));		
reducing	리듀싱 연산에 따름	누적자를 초깃값으로 설정한 다음 BinaryOperator로 스트림의 각 요소를 반복적으로 누적자와 합쳐 스트림을 하나의 값으로 리듀싱
Ex : int totalCalories = menuStream.collect(reducing(0, Dish::getCalories, Integer::sum));		
collecting AndThen	변환 함수에 따름	다른 컬렉터를 감싸고 그 결과에 반환 함수를 적용
Ex : int howManyDishes = menuStream.collect(collectingAndThen(toList(), List::size));		
groupingBy	Map<K, List<T>>	하나의 프로퍼티값을 기준으로 스트림의 항목을 그룹화하며 기준 프로퍼티값을 결과 맵으로 사용
Ex : Map<Dish.Type, List<Dish>> dishesByType = menuStream.collect(groupingBy(Dish::getType));		
partitioningBy	Map<Boolean, List<T>>	프레디케이트를 스트림의 각 항목에 적용한 결과로 항목 분할
Ex : Map<Boolean, List<Dish>> vegetarianDishes = menuStream.collect(partitioningBy(Dish::isVegetarian));		

5.
Collector 인터페이스

Listing 6.4 The Collector Interface

```
public interface Collector<T, A, R> {
    Supplier<A> supplier();
    BiConsumer<A, T> accumulator();
    Function<A, R> finisher();
    BinaryOperator<A> combiner();
    Set<Characteristics> characteristics();
}
```

1. T는 수집될 스트림 항목의 제네릭 형식이다.
2. A는 누적자, 즉 수집 과정에서 중간 결과를 누적하는 객체의 형식이다.
3. R은 수집 연산 결과 객체의 형식(항상 그런 것은 아니지만 대개 컬렉션 형식)이다.

```
1 public class ToListCollector<T> implements Collector<T, List<T>, List<T>>
```

5.
Collector 인터페이스

Collector 인터페이스의 메서드 살펴보기

```
Supplier<A> supplier();
BiConsumer<A, T> accumulator();
Funtion<A, R> finisher();
BinaryOperator<A> combiner();
Set<Characteristics> characteristics();
```

5.
Collector 인터페이스

supplier 메서드: 새로운 결과 컨테이너 만들기

supplier 메서드는 빈 결과로 이루어진 Supplier를 반환해야한다.
즉, supplier는 수집 과정에서 빈 누적자 인스턴스를 만드는 파라미터가 없는 함수다.

```
1  public Supplier<List<T>> supplier() {
2    return () -> new ArrayList<T>();
3  }
```

생성자 참조 전달 방법

```
1  public Supplier<List<T>> supplier() {
2    return ArrayList::new;
3  }
```

5.
Collector 인터페이스

accumulator 메서드 : 결과 컨테이너 요소 추가하기

accumulator 메서드는 리듀싱 연산을 수행하는 함수를 반환한다.
스트림에서 n번째 요소를 탐색할 때 두 인수, 즉 누적자와 n번째 요소를 함수에 적용한다.

```
1  public BiConsumer<List<T>, T> accumulator() {
2    return (list, item) -> list.add(item);
3  }
```

생성자 참조 전달 방법

```
1  public BiConsumer<List<T>, T> accumulator() {
2    return List::add;
3  }
```

5. Collector 인터페이스

finisher 메서드 : 최종 변환값을 결과 컨테이너로 적용하기

finisher 메서드는 스트림 탐색을 끝내고 누적자 객체를 최종 결과로 변환하면서 누적 과정을 끝낼 때 호출할 함수를 반환해야 한다.

```
1  public Function<List<T>, List<T>> finisher() {
2      return i -> i;
3    }
```

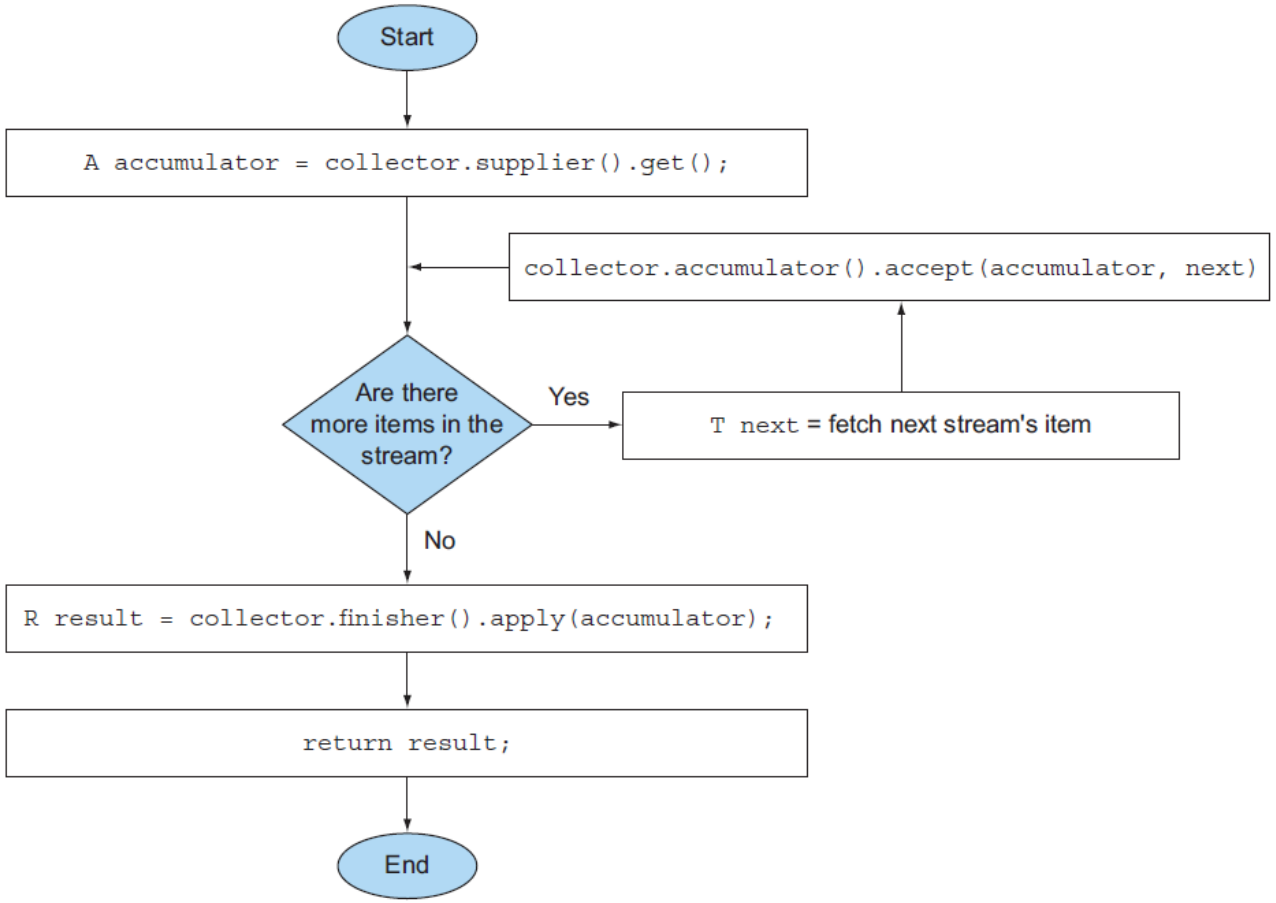
Funtion.identity() 사용법

```
1  public Function<List<T>, List<T>> finisher() {
2      return Function.identity();
3    }
```

```
1  static <T> Function<T, T> identity() {
2      return t -> t;
3  }
```

5. Collector 인터페이스

순차적 스트림 리듀싱



5.
Collector 인터페이스

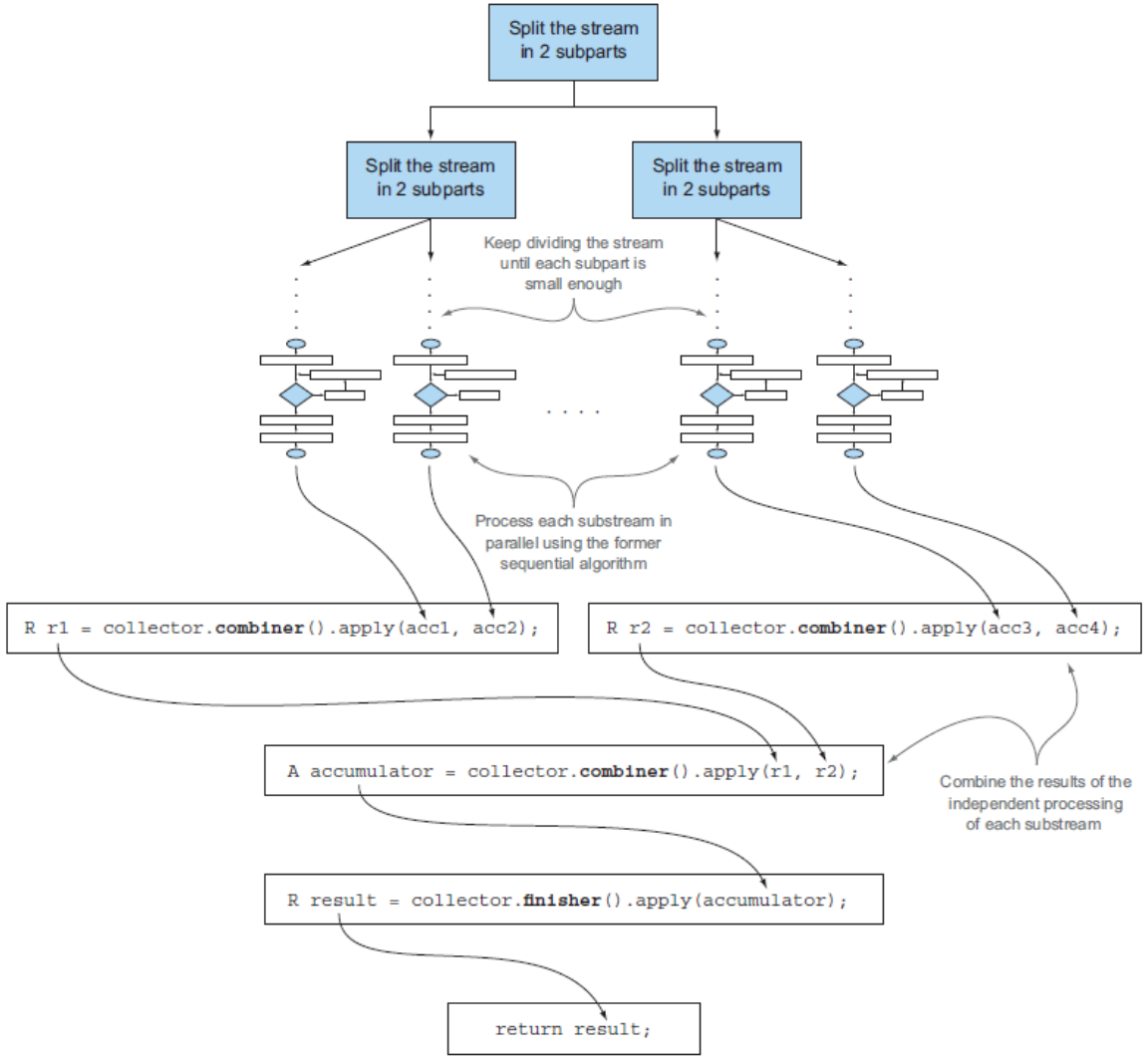
combiner 메서드 : 두 결과 컨테이너 병합

combiner 메서드는 스트림의 서로 다른 서브파트를 병렬로 처리할 때 누적자가 이 결과를 어떻게 처리할지 정의한다.

```
public BinaryOperator<List<T>> combiner() {
  return (list1, list2) -> {
    list1.addAll(list2);
    return list1;
  };
}
```

5. Collector 인터페이스

병렬화 리듀싱 과정에서 combiner 메서드 활용



5. Collector 인터페이스

Characteristics 메서드

characteristics 메서드는 컬렉터의 연산을 정의하는 Characteristics 형식의 불변 집합을 반환한다.

Characteristics는 다음 세 항목을 포함하는 열거형이다.

- **UNORDERED** : 리듀싱 결과는 스트림 요소의 방문 순서나 누적 순서에 영향을 받지 않는다.
- **CONCURRENT** : 다중 스레드에서 accumulator 함수를 동시에 호출할 수 있으며 이 컬렉터는 스트림의 병렬 리듀싱을 수행할 수 있다. 컬렉터의 플래그에 UNORDERED를 함께 설정하지 않았다면 데이터 소스가 정렬되어 있지 않은 상황(Ex : 집합)에서만 병렬 리듀싱을 수행할 수 있다.
- **IDENTITY_FINISH** : finisher 메서드가 반환하는 함수는 단순히 identity를 적용할 뿐이므로 이를 생략할 수 있다. 따라서 최종 결과로 누적자 객체를 바로 사용할 수 있다. 또한 누적자 A를 결과 R로 안전하게 형변환할 수 있다.

ToListCollector에서 스트림의 요소를 누적하는 데 사용한 리스트가 최종 결과 형식이므로 IDENTITY_FINISH다.
리스트의 순서는 상관없으므로 UNORDERED이고, CONCURRENT이다.
요소의 순서가 무의미한 데이터 소스여야 병렬로 수행할 수있다.

5. Collector 인터페이스

응용하기

코 드 확 인

5. Collector 인터페이스

커스텀 컬렉터를 구현해서 성능 개선하기

책에있는 소수와 비소수로 나누는 예제를 커스텀 컬렉터로 구현해보자.

기존 코드

```java
public static Map<Boolean, List<Integer>> partitionPrimes(int n) {
  return IntStream.rangeClosed(2, n).boxed()
      .collect(partitioningBy(candidate -> isPrime(candidate)));
}
```

```java
public static boolean isPrime(int candidate) {
  return IntStream.rangeClosed(2, candidate-1)
      .limit((long) Math.floor(Math.sqrt(candidate)) - 1)
      .noneMatch(i -> candidate % i == 0);
}
```

```
Numbers partitioned in prime and non-prime: {false=[4, 6, 8, 9, 10, 12, 14, ...],
                                             true=[2, 3, 5, 7, 11, 13, 17,...]}
```

5. Collector 인터페이스

커스텀 컬렉터 구현 단계

1단계 : Collector 클래스 시그니처 정의

```
public interface Collector<T, A, R>  <- Collector 인터페이스의 정의
```

T는 스트림 요소의 형식, A는 중간 결과를 누적하는 객체의 형식, R는 collect 연산의 최종 결과.
따라서,

```
public class PrimeNumbersCollector impliments Collector<Integer,                  <- 스트림 요소 형식
                                                 Map<Boolean, List<Integer>>,  <- 누적자 형식
                                                 Map<Boolean, List<Integer>>>  <- 결과 형식
```

2단계 : 리듀싱 연산 구현

Collector 인터페이스에 선언된 다섯 메서드를 구현해야한다. supplier 메서드는 누적자를 만드는 함수를 반환해야 한다.

```
public Supplier<Map<Boolean, List<Integer>>> supplier() {
    return () -> new HashMap<Boolean, List<Integer>>() {{
        put(true, new ArrayList<>());
        put(false, new ArrayList<>());
    }};
}
```

스트림 요소를 어떻게 수집할 지 결정하는 것은 accumulator 메서드이므로 컬렉터에서 가장 중요한 메서드라 할 수 있다.
accumulator는 **최적화의 핵심**이기도 하다.

```
public BiConsumer<Map<Boolean, List<Integer>>, Integer> accumulator() {
    return (Map<Boolean, List<Integer>> acc, Integer candidate) -> {
        acc.get( isPrime(acc.get(true), candidate) )
        .add(candidate);
    };
}
```

5. Collector 인터페이스

커스텀 컬렉터 구현 단계

3단계 : 병렬 실행할 수 있는 컬렉터 만들기(가능하다면)

병렬 수집 과정에서 두 부분 누적자를 합칠 수 있는 combiner 메서드.

```
public BinaryOperator<Map<Boolean, List<Integer>>> combiner() {
    return (Map<Boolean, List<Integer>> map1, Map<Boolean, List<Integer>> map2) -> {
        map1.get(true).addAll(map2.get(true));
        map1.get(false).addAll(map2.get(false));
        return map1;
    };
}
```

하지만 이 예제에선 알고리즘 자체가 순차적이어서 컬렉터를 실제 병렬로 수행할 수 없다.
따라서 combiner 메서드는 호출될 일이 없으므로 빈 구현으로 남겨둘 수 있다.

4단계 : finisher 메서드와 컬렉터의 characteristics 메서드

accumulator의 형식은 컬렉터 결과 형식과 같으므로 변환 과정이 필요 없다.

```
public Function<Map<Boolean, List<Integer>>, Map<Boolean, List<Integer>>> finisher() {
        return i -> i;  //혹은 Function.identity();
}
```

커스텀 컬렉터는 CONCURRENT도 아니고 UNORDERED도 아니지만 IDENTITY_FINISH이므로 다음처럼 구현할 수 있다.

```
public Set<Characteristics> characteristics() {
   return Collections.unmodifiableSet(EnumSet.of(IDENTITY_FINISH));
}
```

..

마치며

- collect는 스트림의 요소를 요약 결과로 누적하는 컬렉터라는 다양한 방법을 인수로 갖는 최종 연산이다.
- 스트림의 요소를 하나로 리듀스하고 요약하는 컬렉터뿐 아니라 최솟값, 최댓값, 평균값을 계산하는 컬렉터 등이 미리 정의되어 있다.
- groupingBy 메서드로 요소를 그룹화 하거나, partitioningBy 메서드로 스트림의 요소를 분할할 수 있다.
- 컬렉터는 다수준의 그룹화, 분할, 리듀싱 연산에 적합하게 설계되어 있다.
- Collector 인터페이스에 정의된 메서드를 구현해서 커스텀 컬렉터를 개발할 수 있다.

감 사 합 니 다