Chapter 09. 리팩터링, 테스팅, 디버깅

2019. 10. 17 | Java in Action Seminar | 박대원

wisoft

이 장의 내용

- 람다 표현식으로 코드 리팩터링하기
- 람다 표현식이 객체지향 설계 패턴에 미치는 영향

➔ 기존 코드에서 람다 표현식을 이용해
가독성과 유연성을 높이려면?

➔ 전략strategy, 템플릿 메서드template method, 옵저버observer,
의무 체인chain of responsibility, 팩토리factory 등의
디자인 패턴을 어떻게 간소화 할까?

- 람다 표현식 테스팅
- 람다 표현식과 스트림 API 사용 코드 디버깅

1.

리팩터링?

람다 표현식의 장점

1. 익명 클래스보다 좀 더 간결한 코드를 만든다.

2. 인수로 전달하려는 메서드가 이미 있을 경우, 메서드 참조를 이용할 수 있다.

3. 다양한 요구사항 변화에 대응할 수 있도록 동작을 파라미터화한다.

여기에 더해서 스트림 등의 기능을 이용해서 더 가독성이 좋고 유연한 코드로 **리팩터링.**

리팩터링refactoring : 사용자가 보는 외부 화면은 그대로 두면서 내부 논리나 구조를 바꾸고 개선하는 유지보수 행위.

1.1.
코드 가독성 개선

코드 가독성? == ‘어떤 코드를 다른 사람도 쉽게 이해할 수 있음’을 의미.

자바 8의 새 기능들을 이용해 코드의 가독성을 높일 수 있다.

- 익명 클래스를 람다 표현식으로 리팩터링하기
- 람다 표현식을 메서드 참조로 리팩터링하기
- 명령형 데이터 처리를 스트림으로 리팩터링하기

1.2.

익명 클래스를 람다 표현식으로 리팩터링

하나의 추성 메서드를 구현하는 익명 클래스는 람다 표현식으로 리팩터링 할 수 있다.

익명 클래스는 코드를 장황하게 만들고 쉽게 에러를 일으키기 때문에 리팩터링 해야한다.

하지만 모든 익명 클래스를 람다식으로 변환할 수 있는 것은 아니다.

1. 익명 클래스에 사용한 this와 super는 람다식에서 다른 의미를 갖는다.

   익명 클래스에서 this는 익명 클래스 자신을 가리키지만 람다에서는 람다를 감싸는 클래스를 가리킨다.

2. 익명 클래스는 감싸고 있는 클래스의 변수를 가릴 수 있다. (섀도 변수shadow variable)

➔ 코드 확인

1.3.

람다 표현식을 메서드 참조로 리팩터링하기

람다 표현식은 쉽게 전달할 수 있는 짧은 코드. 때에 따라서 가독성이 떨어질 수 있다.

이 대신 **메서드 참조**Method reference를 이용하면 보다 가독성을 높일 수 있다.

메서드 참조의 메서드명으로 코드의 의도를 명확하게 알릴 수 있기 때문이다.

➔ 코드 확인

1.3.

람다 표현식을 메서드 참조로 리팩터링하기

또한 comparing, maxBy 같은 정적 헬퍼 메서드를 활용하는 것도 좋다.
이들은 메서드 참조와 조화를 이루도록 설계되었다.

```
1 inventory.sort(
2   (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight()));
```

```
1 inventory.sort(comparing(Apple::getWeight));
```

최댓값이나 합계를 계산할 때 람다식과 저수준 리듀싱 연산을 조합하는 것보다
Collectors API를 사용하면 코드의 의도가 더 명확해진다.

```
1 int totalCalories =
2   menu.stream().map(Dish::getCalories)
3               .reduce(0, (c1, c2) -> c1 + c2);
```

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

1.4.

명령형 데이터 처리를 스트림으로 리팩터링하기

이론적으로는 반복자를 이용한 기존의 모든 컬렉션 처리 코드를 스트림 API를 바꿔야 한다.

➔ 스트림 API는 쇼트서킷, 게으름, 멀티코어 아키텍처 활용 등 이로운 기능을 제공.

필터링과 추출이 엉킨 명령형 코드

```
1 List<String> dishNames = new ArrayList<>();
2 for (Dish dish : menu) {
3   if (dish.getCalories() > 300) {
4     dishNames.add(dish.getName());
5   }
6 }
```

스트림 API 사용 코드

```
1 menu.stream().parallel()
2   .filter(d -> d.getCalories() > 300)
3   .map(Dish::getName)
4   .collect(toList());
```

1.5.
코드 유연성 개선

동작 파라미터화를 통해 다양한 람다를 전달해서 다양한 동작을 표현할 수 있다.

➔ 변화하는 요구사항에 대응할 수 있는 코드를 구현할 수 있다!

함수형 인터페이스 적용

먼저 람다식을 이용하려면 함수형 인터페이스가 필요하다.

이번에는 **조건부 연기 실행**conditional deferred execution과 **실행 어라운드**execute around

두 가지 자주 사용하는 패턴으로 람다 표현식 리팩터링을 살펴보자.

1.5. 코드 유연성 개선

조건부 연기 실행 패턴

실제 작업을 처리하는 코드 내부에 제어 흐름문이 복잡하게 얽힌 코드를 흔히 볼 수 있다.

로깅 관련 코드를 예로 살펴보자. (Java의 Logger 내장 클래스)

```
1 if (logger.isLoggable(Log.FINER)) {
2   logger.finer("Problem: " + generateDiagnostic());
3 }
```

문제점

- logger의 상태가 isLoggable이라는 메서드에 의해 클라이언트 코드로 노출된다.
- 메시지를 로깅할 때마다 logger 객체의 상태를 매번 확인해야 할까?

➔ log 메서드 사용

```
1 logger.log(Level.FINER, "Problem: " + generateDiagnostic());
```

아직 해결되지 않은 문제를 람다로 해결할 수 있다.

1.5. 코드 유연성 개선

자바 8에서 추가된 오버로드된 log 메서드

```
1 public void log(Level level, Supplier<String> msgSupplier)
```

다음처럼 호출할 수 있다.

```
1 logger.log(Level.FINER, () -> "Problem: " + generateDiagnostic());
```

log 메서드는 logger의 수준이 적절하게 설정되어 있을 때(Level.FINER)만
인수로 넘겨진 Supplier를 내부적으로 실행한다.

```
1 public void log(Level level, Supplier<String> msgSupplier) {
2   if (logger.isLoggable(level)) {
3     log(level, msgSupplier.get()); //람다 실행
4   }
5 }
```

이 기법으로 클라이언트 코드에서 객체 상태를 자주 확인하거나, 객체의 일부 메서드를 호출하는 상황이라면
내부적으로 객체의 상태를 확인한 다음에 메서드를 호출하도록 새로운 메서드를 구현하는 것이 좋다.

그러면 코드 가독성이 좋아지고 캡슐화도 강화된다!

1.5.
코드
유연성 개선

실행 어라운드 패턴

매번 같은 준비, 종료 과정을 반복적으로 수행하는 코드가 있다면 이를 람다로 변환할 수 있다.

```
@FunctionalInterface
public interface BufferedReaderProcessor {

  String process(BufferedReader bufferedReader) throws IOException;

}

public static String processFile(BufferedReaderProcessor bufferedReaderProcessor) throws IOException {
  try (BufferedReader br =
           new BufferedReader(new FileReader("src/main/resources/data.txt"))) {
    return bufferedReaderProcessor.process(br);
  }
}

String oneLine = processFile(br -> br.readLine());
String twoLine = processFile(br -> br.readLine() + br.readLine());
```

2.

람다로 객체지향 디자인 패턴 리팩터링 하기

디자인 패턴

공통적인 소프트웨어 문제를 설계할 때 재사용할 수 있는 검증된 청사진을 제공한다.

- 전략 (Strategy)
- 템플릿 메서드 (Template method)
- 옵저버 (Observer)
- 의무 체인(Chain of responsibility)
- 팩토리 (Factory)

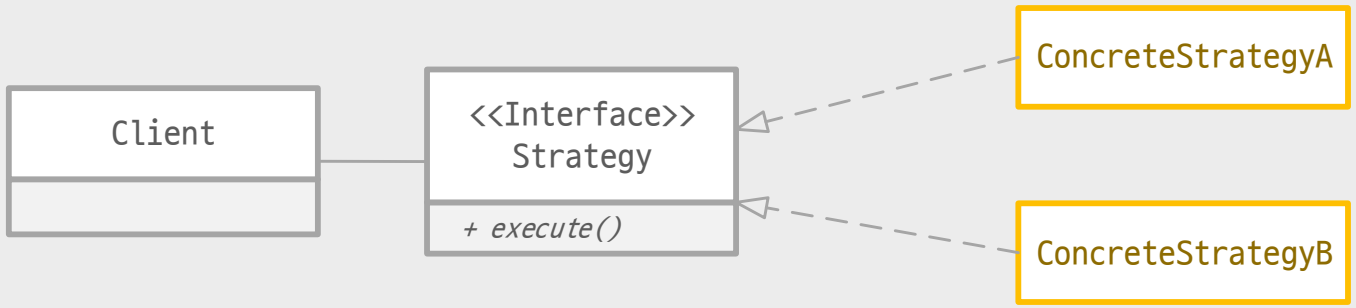
각 디자인 패턴에서 람다를 어떻게 활용할 수 있는지 알아보자!

2.1.

전략

전략 패턴은

한 유형의 알고리즘을 보유한 상태에서 런타임에 적절한 알고리즘을 선택하는 기법.



- Strategy : 알고리즘을 나타내는 인터페이스
- ConcreteStrategyA, B : 다양한 알고리즘을 나타내는 한개 이상의 인터페이스 구현체
- Client : 전략 객체를 사용하는 한 개 이상의 클라이언트

➔ 코드 확인

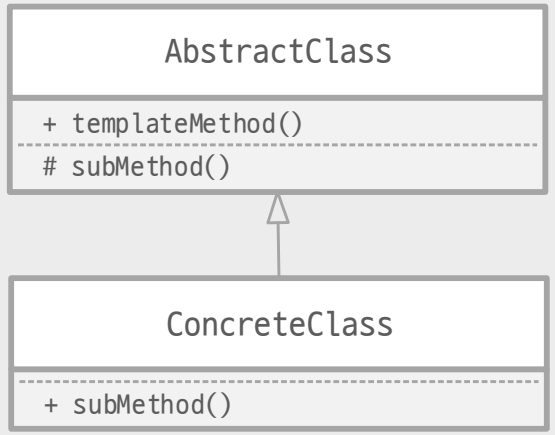
람다 표현식은 코드 조각(전략)을 캡슐화 한다.
즉, 람다 표현식으로 전략 디자인 패턴을 대신할 수 있다.

2.2.

템플릿 메서드

템플릿 메서드 패턴은

알고리즘의 개요를 제시한 다음에 알고리즘의 일부를 고칠 수 있는 유연함을 제공해야 할 때 사용.



→ 알고리즘 구현체

→ 기존 알고리즘 구현체에서 일부 수정

➔ 코드 확인

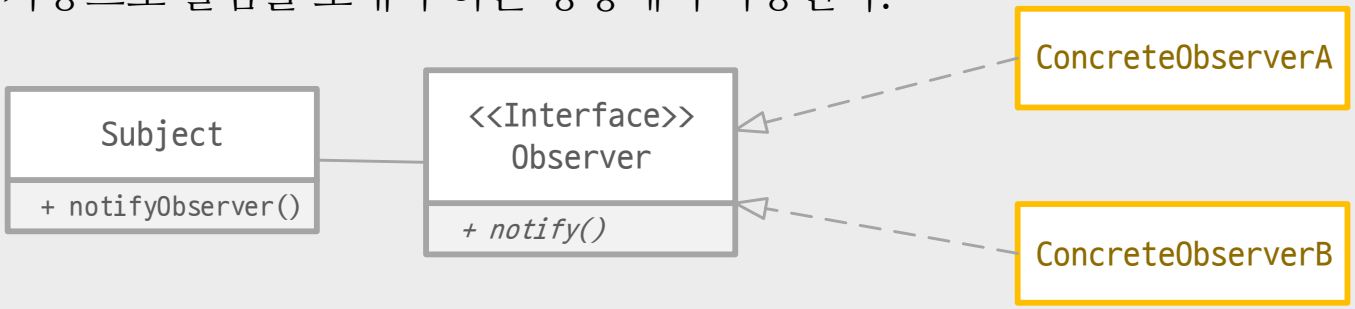
마찬가지로 람다 표현식을 이용하면 템플릿 메서드 디자인 패턴에서 발생하는 자잘한 코드도 제거할 수 있다.

2.3.

옵저버

옵저버 패턴은

어떤 이벤트가 발생했을 때 한 객체(**주제**subject)가 다른 객체 리스트(**옵저버**observer)에 자동으로 알림을 보내야 하는 상황에서 사용한다.



➔ 코드 확인

하지만 항상 람다 표현식을 사용하는 것은 아니다. 만약 옵저버가 상태를 가지고 여러 메서드를 정의하는 등 복잡한 구조를 가진다면 람다 표현식보다 기존의 클래스 구현방식을 고수하는 것이 바람직할 수도 있다.

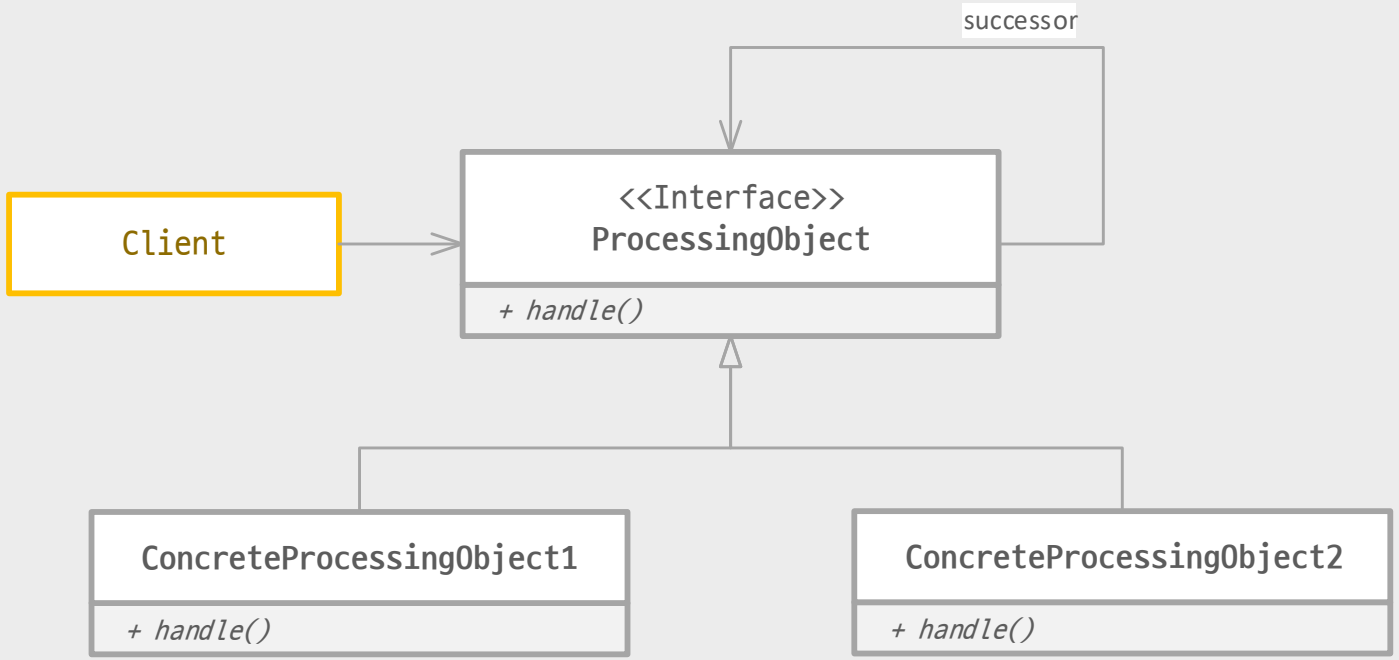
2.4.

의무 체인

의무 체인 패턴은

작업 처리 객체의 체인(동작 체인 등)을 만들 때는 의무 체인 패턴을 사용한다.

한 객체가 어떤 작업을 처리한 다음에 다른 객체로 결과를 전달하고, 다른 객체도 해야 할 작업을 처리한 다음에 또 다른 객체로 전달하는 식이다.



➔ 코드 확인

2.5.

팩토리

팩토리 패턴은

인스턴스화 로직을 클라이언트에 노출하지 않고 객체를 만들 때 사용한다.

```
1 public class ProductFactory {
2   public static Product createProduct(String name) {
3     switch(name) {
4       case "loan": return new Loan();
5       case "stock": return new Stock();
6       case "bond": return new Bond();
7       default: throw new RuntimeException("No such product " + name);
8     }
9   }
10 }
```

생성자와 설정을 외부로 노출하지 않음으로써 클라이언트가 단순하게 상품을 생산할 수 있다.

```
1 Product p = ProductFactory.createProduct("loan");
```

생성자도 메서드 참조처럼 접근할 수 있다.

➔ 코드 확인

3.

람다 테스팅

개발자의 최종 목표는 **제대로 작동하는 코드**를 구현하는 것. 깔끔한 코드를 구현하는게 아니다.

좋은 소프트웨어 공학자라면 프로그램이 의도대로 동작하는지 확인할 수 있는 **단위 테스팅**unit testing을 진행한다.

```java
public class Point {

  private final int x;
  private final int y;

  public Point(final int x, final int y) {
    this.x = x;
    this.y = y;
  }

  public int getX() {
    return x;
  }

  public int getY() {
    return y;
  }

  public Point moveRightBy(final int x) {
    return new Point(this.x + x, this.y);
  }

}

```

➔ 코드 확인

3.1.

보이는 람다 표현식의 동작 테스팅

이전 예제에서 moveRightBy는 public이므로 위 코드는 문제없이 작동한다.
하지만 람다는 익명이므로 테스트 코드 이름을 호출할 수 없다.
따라서 단위 테스팅이 필요하다면 람다를 필드에 저장해서 재사용할 수 있고, 람다의 로직을 테스트할 수 있다.

이제 메서드를 호출하는 것처럼 람다를 사용할 수 있다.
그전에 람다 표현식은 함수형 인터페이스의 인스턴스를 생성한다는 사실을 기억하자.

➔ 코드 확인

3.2.

람다를 사용하는 메서드의 동작에 집중하라

람다의 목표는 정해진 동작을 다른 메서드에 사용할 수 있도록 캡슐화 하는 것이다.
그러려면 세부 구현을 포함하는 람다 표현식을 공개하지 말아야 한다.

람다 표현식을 사용하는 메서드의 동작을 테스트함으로써
람다를 공개하지 않으면서도 람다 표현식을 검증할 수 있다.

➔ 코드 확인

3.4.

고차원 함수 테스팅

함수를 인수로 받거나 다른 함수를 반환하는 함수를 **고차원 함수**higher-order functions라고 한다.

이는 테스팅을 사용하기 좀 더 어렵다. 메서드가 람다를 인수로 받는다면 다른 람다로 메서드의 동작을 테스트할 수 있다.

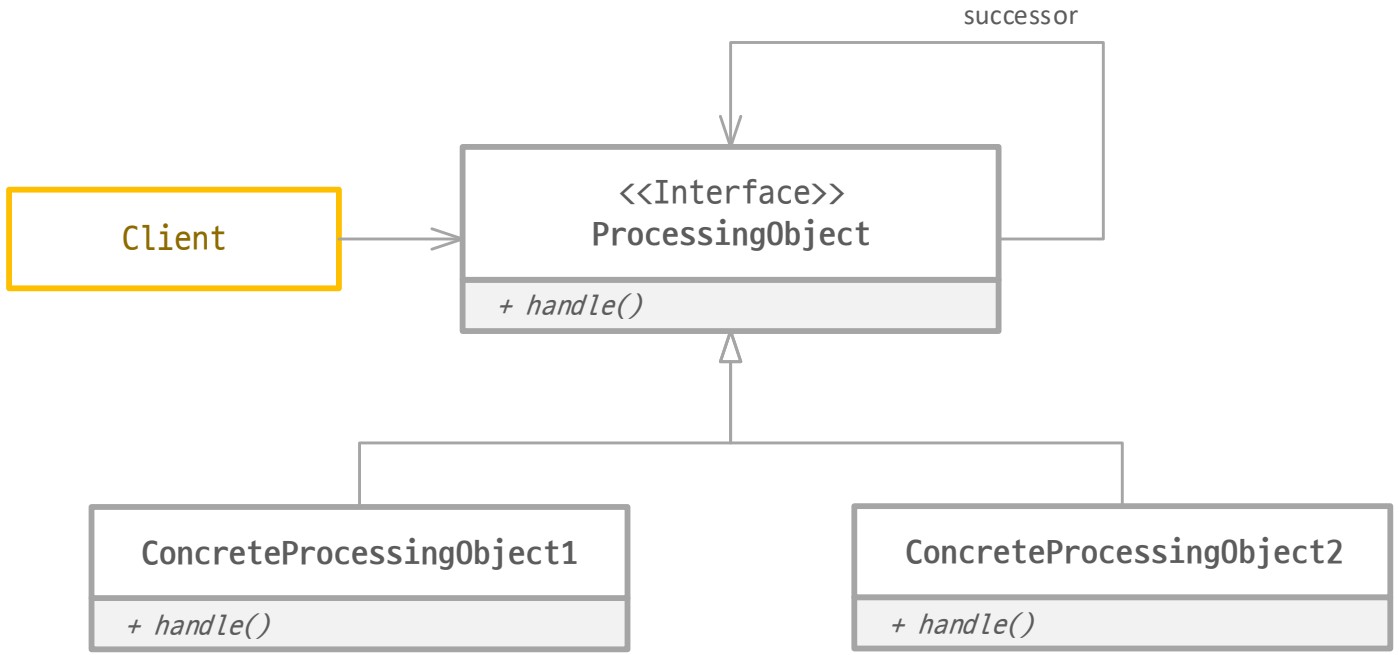
2.4.

의무 체인

의무 체인 패턴은

작업 처리 객체의 체인(동작 체인 등)을 만들 때는 의무 체인 패턴을 사용한다.

한 객체가 어떤 작업을 처리한 다음에 다른 객체로 결과를 전달하고, 다른 객체도 해야 할 작업을 처리한 다음에 또 다른 객체로 전달하는 식이다.



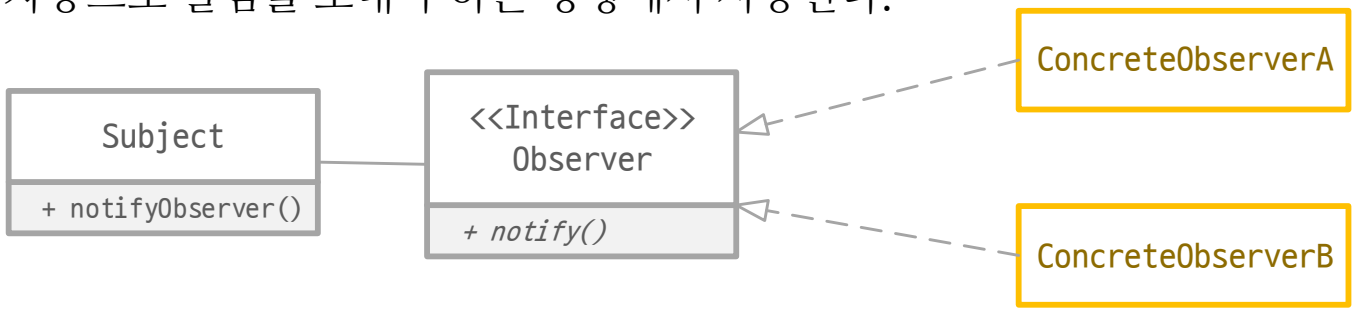
➔ 코드 확인

2.2.

옵저버

옵저버 패턴은

어떤 이벤트가 발생했을 때 한 객체(**주제**subject)가 다른 객체 리스트(**옵저버**observer)에 자동으로 알림을 보내야 하는 상황에서 사용한다.



➔ 코드 확인

하지만 항상 람다 표현식을 사용하는 것은 아니다. 만약 옵저버가 상태를 가지고 여러 메서드를 정의하는 등 복잡한 구조를 가진다면 람다 표현식보다 기존의 클래스 구현방식을 고수하는 것이 바람직할 수도 있다.

2.

컬렉터란 무엇인가?

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

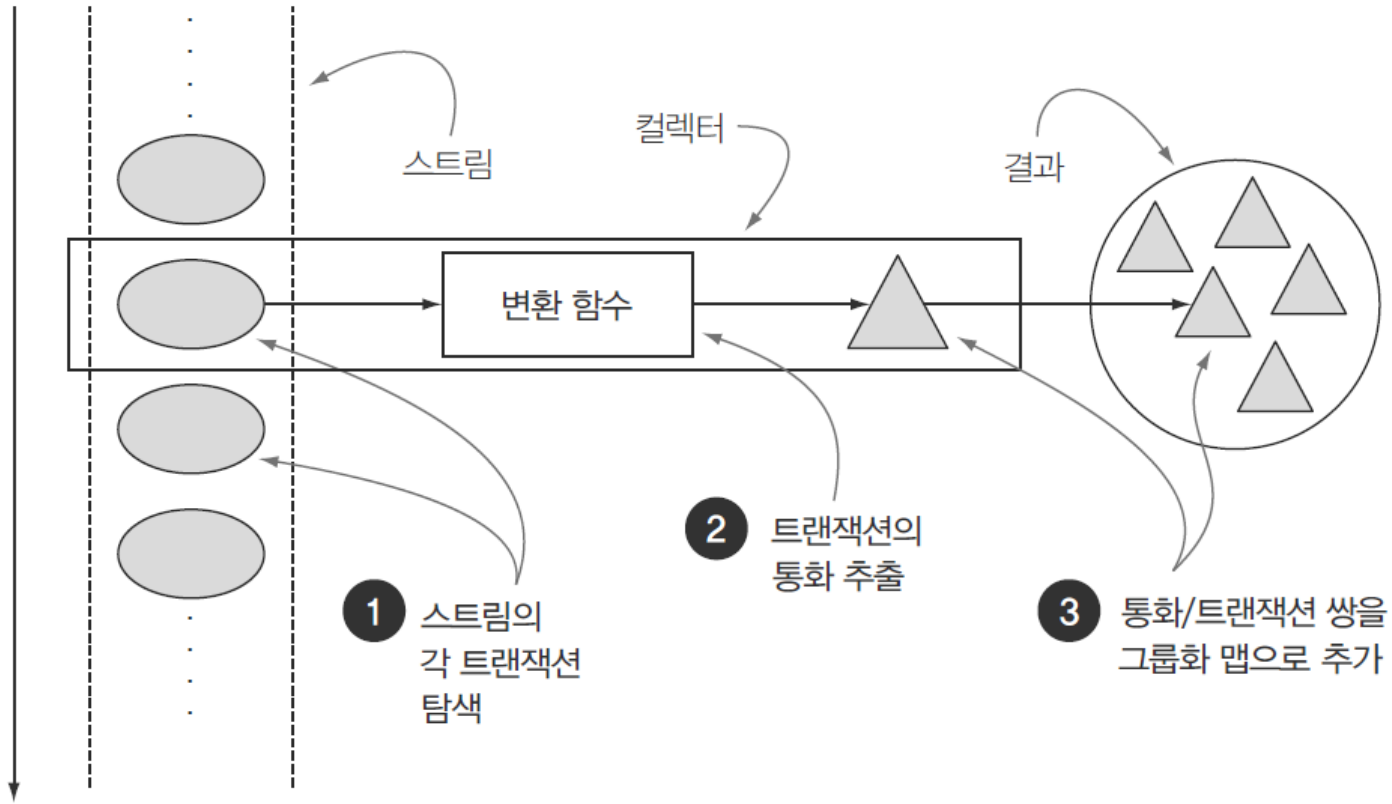
groupImperatively(); < *groupFunctionally*();

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));
```

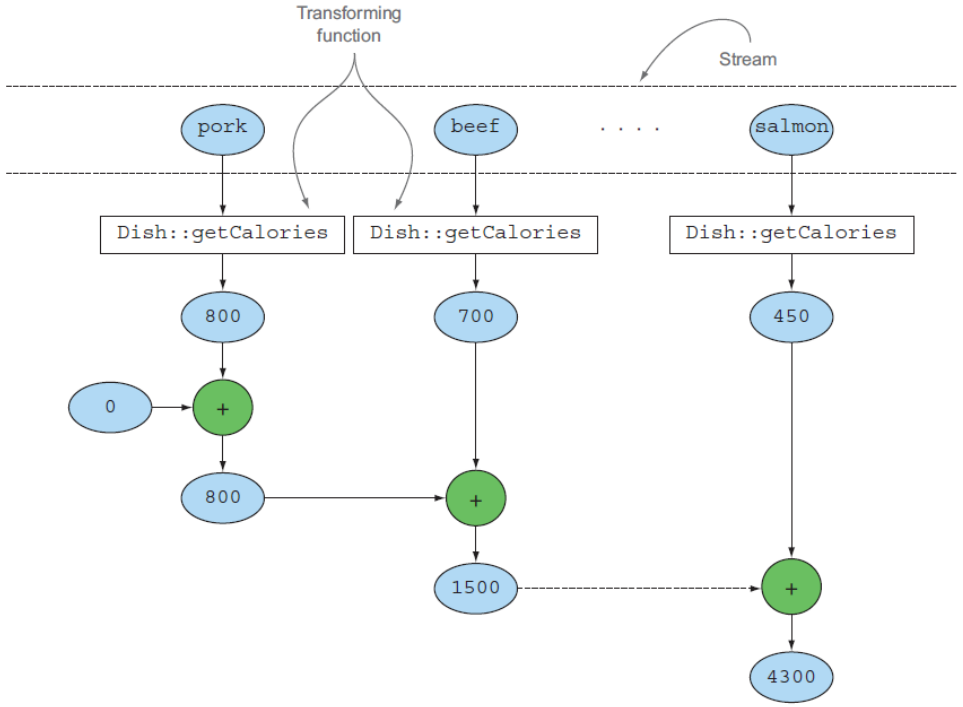
이처럼 스트림에 있는 객체의 숫자 필드의 합계나 평균 등을 반환하는 연산에도 리듀싱이 자주 사용되는데, 이런 연산을 **요약**summarization **연산**이라 부른다.

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

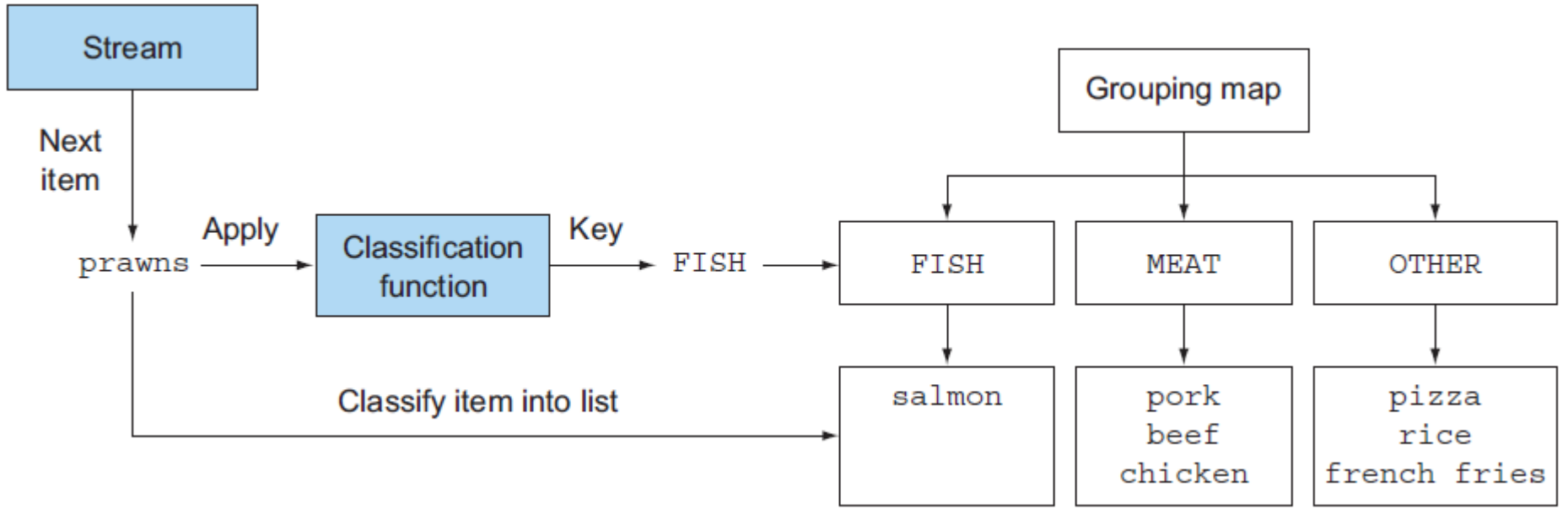
```
String shortMenu = menu.stream().map(Dish::getName).collect(joining());

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;
            } )
      )
);
```

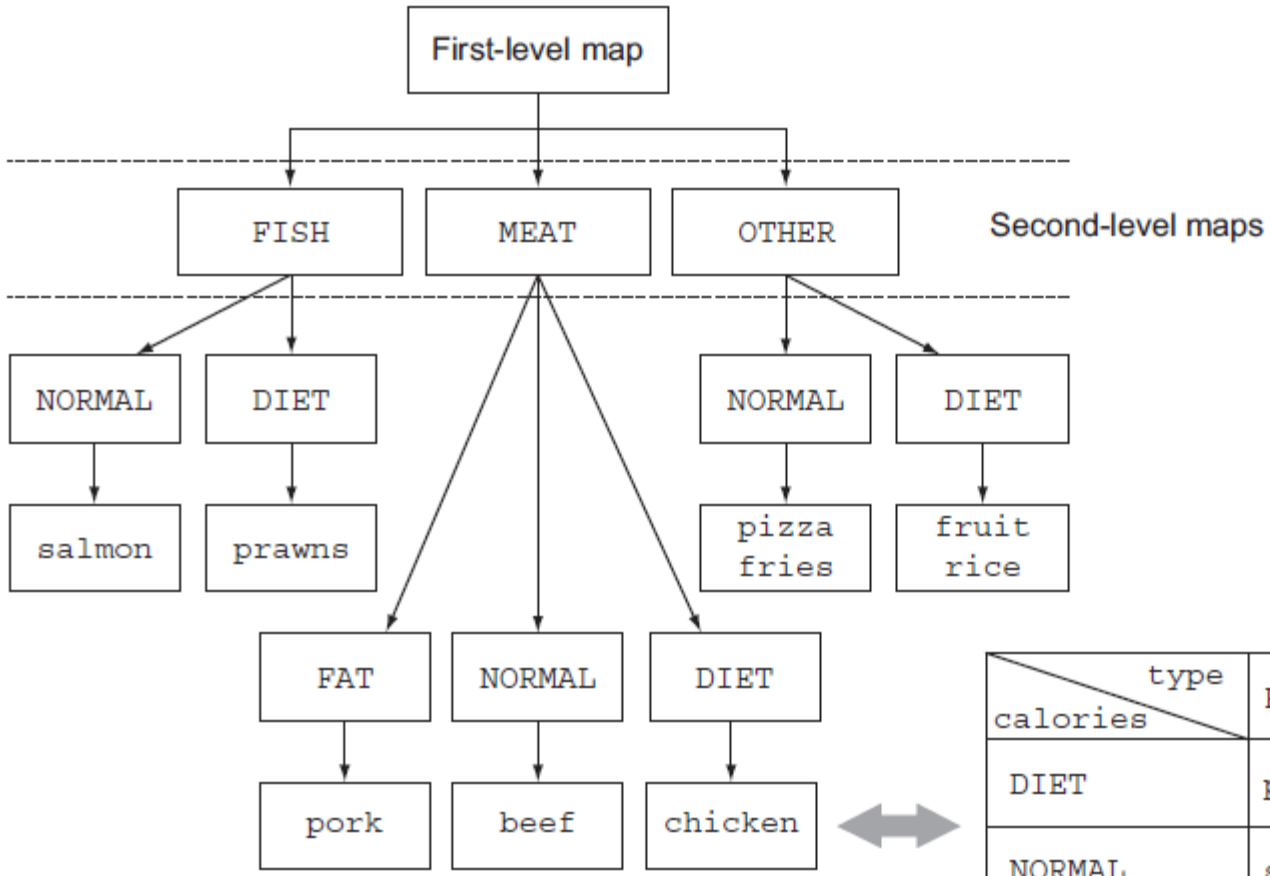
First-level classification function

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

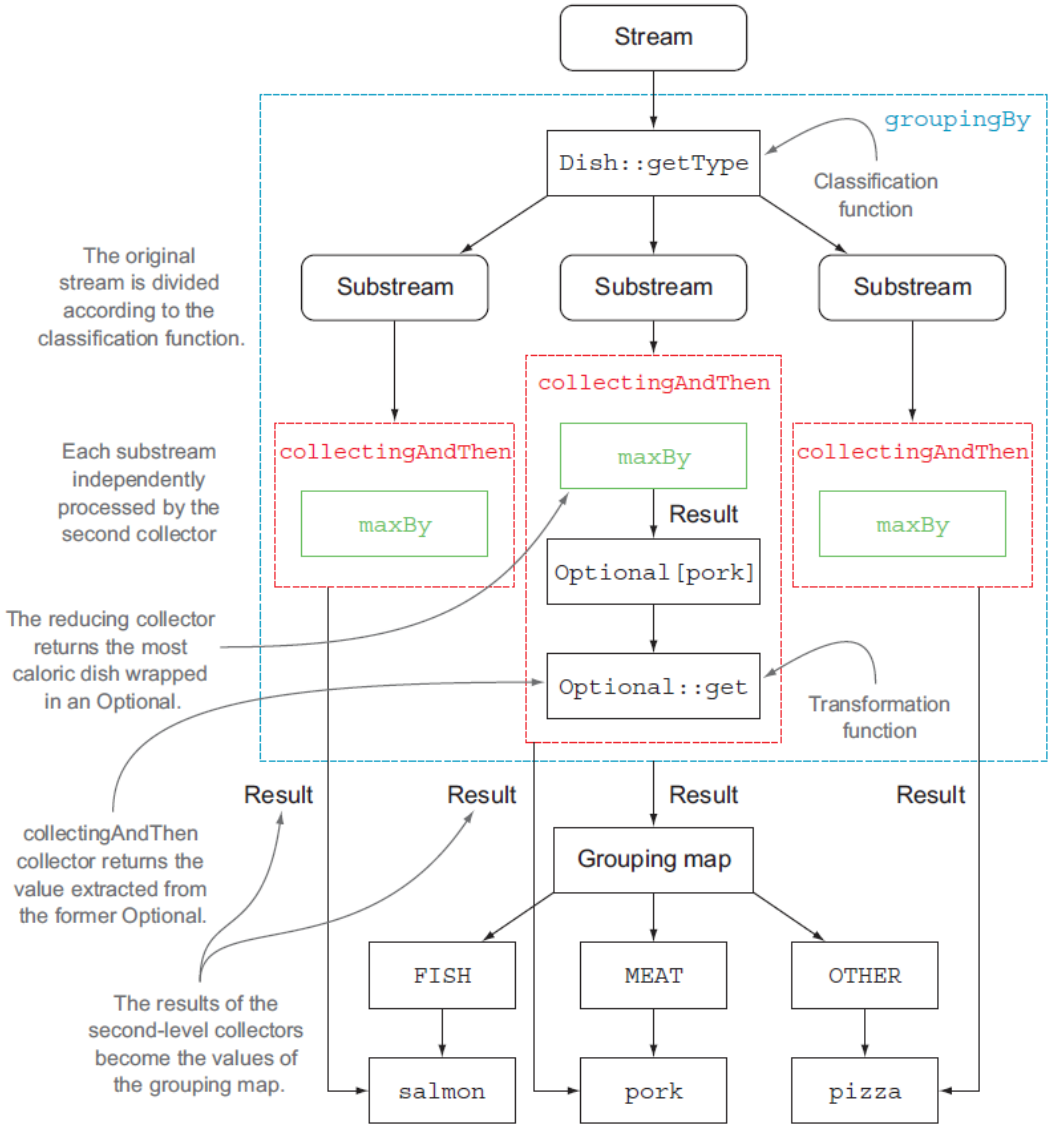
```
1  Map<Dish.Type, Dish> mostCaloricByType =
2    menu.stream()
3    .collect(groupingBy(Dish::getType,
4    collectingAndThen(maxBy(comparingInt(Dish::getCalories)), Optional::get)));
```

{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    }
```

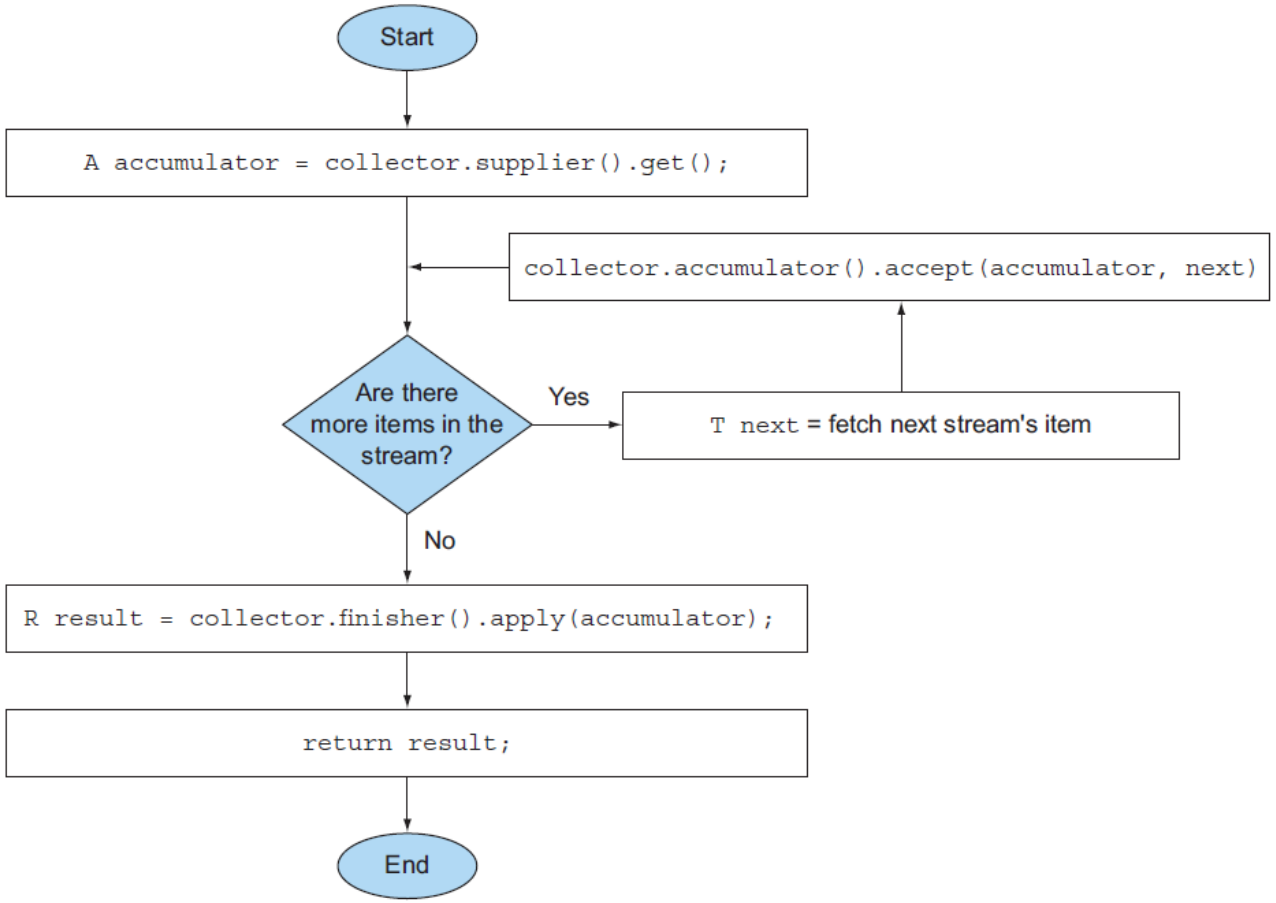
Funtion.identity() 사용법

```
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

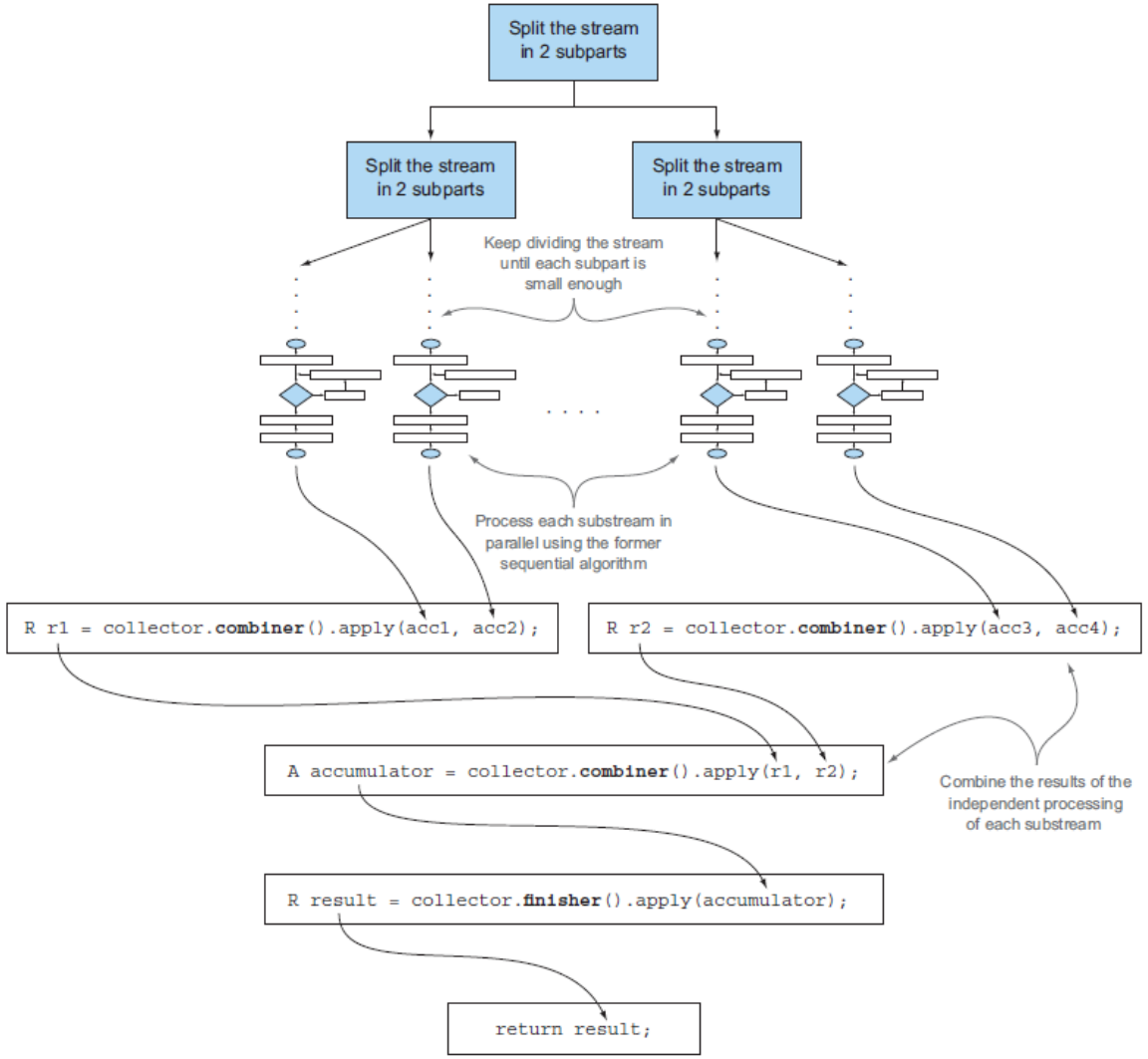
combiner 메서드 : 두 결과 컨테이너 병합

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