

Java Stream Programming

Lambda and Stream

장민창

mcjang@huccloud.co.kr

Java Stream Programming

1. Lambda Expression
2. Stream
3. Filtering & Slicing
4. Mapping
5. Find & Matching
6. Reducing
7. Primitive Stream
8. Generate Stream
9. Collectors

1. Lambda Expression

1. Lambda Expression

- 람다(Lambda)란?
 - 람다 미적분학 학계에서 개발할 시스템에서 유래함.
 - 메소드에게 전달할 수 있는 익명 클래스를 함수로 단순화 시킨 것.
- 람다의 특징
 - 익명
 - 보통의 메소드와 달리 이름이 없다.
 - 함수
 - 특정 클래스에 종속되지 않아 함수라 부른다.
 - 전달
 - 람다 표현식을 메서드의 인자로 전달하거나 변수로 저장할 수 있다.
 - 간결성
 - 익명 클래스처럼 자질구레한 코드를 구현할 필요가 없다.

1. Lambda Expression

- 람다의 적용

```
employees.sort( new Comparator<Employees>(){  
    @Override  
    public int compare(Employees emp1, Employees emp2) {  
        return emp2.getEmployeeId() - emp1.getEmployeeId();  
    }  
} );
```



```
employees.sort( (emp1, emp2) -> emp2.getEmployeeId() - emp1.getEmployeeId() );
```

1. Lambda Expression

- 람다의 구성

화살표

(emp1, emp2) -> emp2.getEmployeeId() - emp1.getEmployeeId();

람다 파라미터 람다 바디

- 람다 파라미터
 - Comparator의 compare 메소드의 파라미터 (두 명의 사원)
- 화살표 (->)
 - 람다의 파라미터 리스트와 바디를 구분함.
- 람다 바디
 - 두 명의 사원 번호를 구분함. 람다의 반환값에 해당하는 표현식

1. Lambda Expression

- Java 8에서 유효한 다섯가지 람다 표현식
 - String 파라미터 하나를 가지며 int를 반환.
람다표현식에는 return 이 함축되어 있으므로 명시적으로 사용하지 않아도 된다.

(String s) -> s.length()

- Employees 형식의 파라미터 하나를 가지며 boolean을 반환한다.

(Employees emp) -> emp.getSalary() > 5000

- int 형식의 파라미터 두개를 가지며 리턴 값이 없다(void 리턴).
람다 표현식을 여러 행의 문장을 포함할 수 있다.

```
(int x, int y) -> {  
    System.out.println("Result : ");  
    System.out.println(x + y);  
}
```

- 파라미터가 없으며 int를 반환한다.

() -> 42

- Employees 형식의 파라미터 두개를 가지며, int를 반환한다.

(Employees emp1, Employees emp2) -> emp1.getSalary() - emp2.getSalary()

*{ } 중괄호가 있는 람다 표현식에서 리턴이 필요한 경우
return을 명시적으로 작성해야 한다.*

1. Lambda Expression

- 함수형 인터페이스
 - 랴다는 함수형 인터페이스를 파라미터로 받는 메소드에게만 사용할 수 있다.
 - 함수형 인터페이스는 오직 하나의 추상메소드만 지정되어 있는 인터페이스를 말한다.
 - Comparator 인터페이스의 추상메소드는 `int compare(T o1, T o2);` 하나.
- 함수형 인터페이스의 종류
 - Predicate<T> : 파라미터 하나를 전달받아, boolean을 반환하는 함수형 인터페이스
 - BiPredicate<L, R> : 파라미터 L과 R을 전달받아 boolean을 반환하는 함수형 인터페이스
 - Consumer<T> : 파라미터 하나를 전달받아, void를 반환하는 함수형 인터페이스
 - BiConsumer<T, U> 파라미터 T와 U를 전달받아 void를 반환하는 함수형 인터페이스
 - Function<T, R> : 파라미터 T를 전달받아 R을 반환하는 함수형 인터페이스
 - BiFunction<T, U, R> : 파라미터 T와 U를 전달받아 R을 반환하는 함수형 인터페이스


1. Lambda Expression

- 메소드 레퍼런스

- 메소드 레퍼런스를 이용하면, 기존의 메소드를 파라미터로 전달 할 수 있다.
- 메소드의 파라미터에 함수형 인터페이스가 정의되어 있을 때, 메소드 레퍼런스를 인자로 전달할 수 있다.

```
List<Employees> subordinates = filterEmployees(employees, Employees::isManagerIs100);
```

```
public static List<Employees> filterEmployees(  
    List<Employees> employees, Predicate<Employees> p ) {  
    List<Employees> result = new ArrayList<>();  
    for (Employees employee : employees) {  
        if ( p.test(employee) ) {  
            result.add(employee);  
        }  
    }  
    return result;  
}
```



1. Lambda Expression

- 람다와 메소드 레퍼런스 단축 표현 예제

람다	메서드 레퍼런스 단축 표현
(Apple e) -> e.getWeight()	Apple::getWight
() -> Thread.currentThread().dumpStack()	Thread.currentThread::dumpStack
(str, i) -> str.substring(i)	String::substring
(String s) -> System.out.println(s)	System.out::println

- 메소드 레퍼런스 만드는 세 가지 방법

- 1. 정적 메소드 레퍼런스

- Integer.parseInt() 메소드는 Integer::parseInt 로 표현함.

- 2. 다양항 형식의 인스턴스 메소드 레퍼런스

- String.length() 메소드는 String::length로 표현함.

- 3. 기존 객체의 인스턴스 메소드 레퍼런스

- Employees 객체를 할당받은 employee 라는 변수가 있고,
Employees 에는 getSalary() 라는 메소드가 있다면, employee::getSalary 로 표현함.

1. Lambda Expression

- 세 가지 종류의 람다 표현식을 메소드 레퍼런스로 바꾸는 방법

1

람다

`(args) -> ClassName.staticMethod(args)`

메서드 레퍼런스

`ClassName::staticMethod`

2

람다

`(arg0, rest) -> arg0.instanceMethod(rest)`
arg0은 ClassName 형식

메서드 레퍼런스

`ClassName::instanceMethod`

3

람다

`(args) -> expr.instanceMethod(args)`

메서드 레퍼런스

`expr::instanceMethod`

2. Stream

2. Stream

- 스트림이란?
 - 병렬처리를 지원하는 내부 반복자.
 - 컬렉션이나 배열을 스트림으로 변환하여 여러가지 형태의 처리를 할 수 있다.
 - 반복문을 이용하던 기존의 컬렉션을 스트림으로 처리할 수 있다.
 - 반복문을 사용하지 않고 컬렉션을 제어할 수 있다

```
for (Employees subordinate : subordinates) {  
    System.out.println(subordinate.getEmployeeId());  
}
```



```
subordinates  
    .stream()  
    .forEach((emp) -> System.out.println(emp.getEmployeeId()));
```

2. Stream

- 스트림을 사용하는 이유?
 - 1. 정렬, 필터링, 맵핑 등을 통해 지저분하고 중복되던 반복문 코드가 사라짐.
 - 2. 데이터를 제어하는 주체가 RDB 에서 JAVA로 옮겨짐.
 - 이유
 - 1. 데이터가 늘어남에 따라 데이터를 정제하는 RDB 능력의 한계가 드러남.
 - 2. 비정형 데이터를 RDB 에서 정제할 수 없음.
- RDB의 역할을 Java의 Stream이 대신 처리함.

2. Stream

- 기존 컬렉션의 사용 예제
 - 저칼로리 음식만 필터링 한 후 칼로리 순으로 정렬, 출력하는 예제
 - 코드가 장황하다.

```
List<Dish> lowCaloricDishes = new ArrayList<>();  
for (Dish d: menu) {  
    if (d.getCalories() < 400) {  
        lowCaloricDishes.add(d);  
    }  
}
```

누적자로
요소 필터링

익명 클래스로
요리 정렬

```
Collections.sort(lowCaloricDishes, new Comparator<Dish>() {  
    public int compare(Dish d1, Dish d2) {  
        return Integer.compare(d1.getCalories(), d2.getCalories());  
    }  
});
```

```
List<String> lowCalroricDishesName = new ArrayList<>();  
for (Dish d: lowCaloricDishes) {  
    lowCalroricDishesName.add(d.getName());  
}
```

정렬된 리스트 처리하면서
요리 이름 선택

2. Stream

- 기존의 코드를 스트림으로 변경

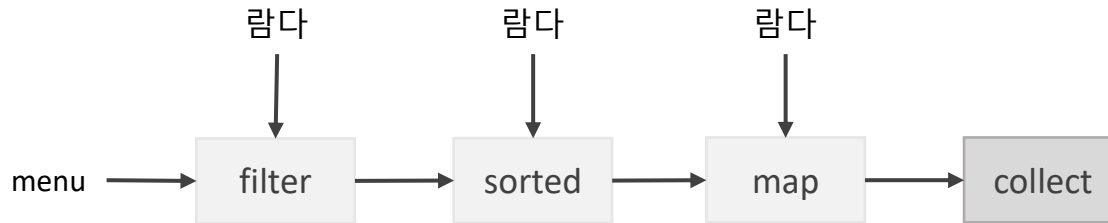
```
List<String> lowCaloricDishesName =  
    menu.stream()  
        // 400 칼로리 이하의 요리 선택  
        .filter(d -> d.getCalories() < 400)  
        // 칼로리로 요리 정렬  
        .sorted(Comparator.comparing(Dish::getCalories))  
        // 요리명 추출  
        .map(Dish::getName)  
        // 모든 요리명을 리스트에 저장  
        .collect(Collectors.toList());
```

- 병렬처리로 가능.

```
List<String> lowCaloricDishesName =  
    menu.parallelStream()  
        // 400 칼로리 이하의 요리 선택  
        .filter(d -> d.getCalories() < 400)  
        // 칼로리로 요리 정렬  
        .sorted(Comparator.comparing(Dish::getCalories))  
        // 요리명 추출  
        .map(Dish::getName)  
        // 모든 요리명을 리스트에 저장  
        .collect(Collectors.toList());
```


2. Stream

- 스트림은 코드를 블록 단위로 관리 / 처리한다.

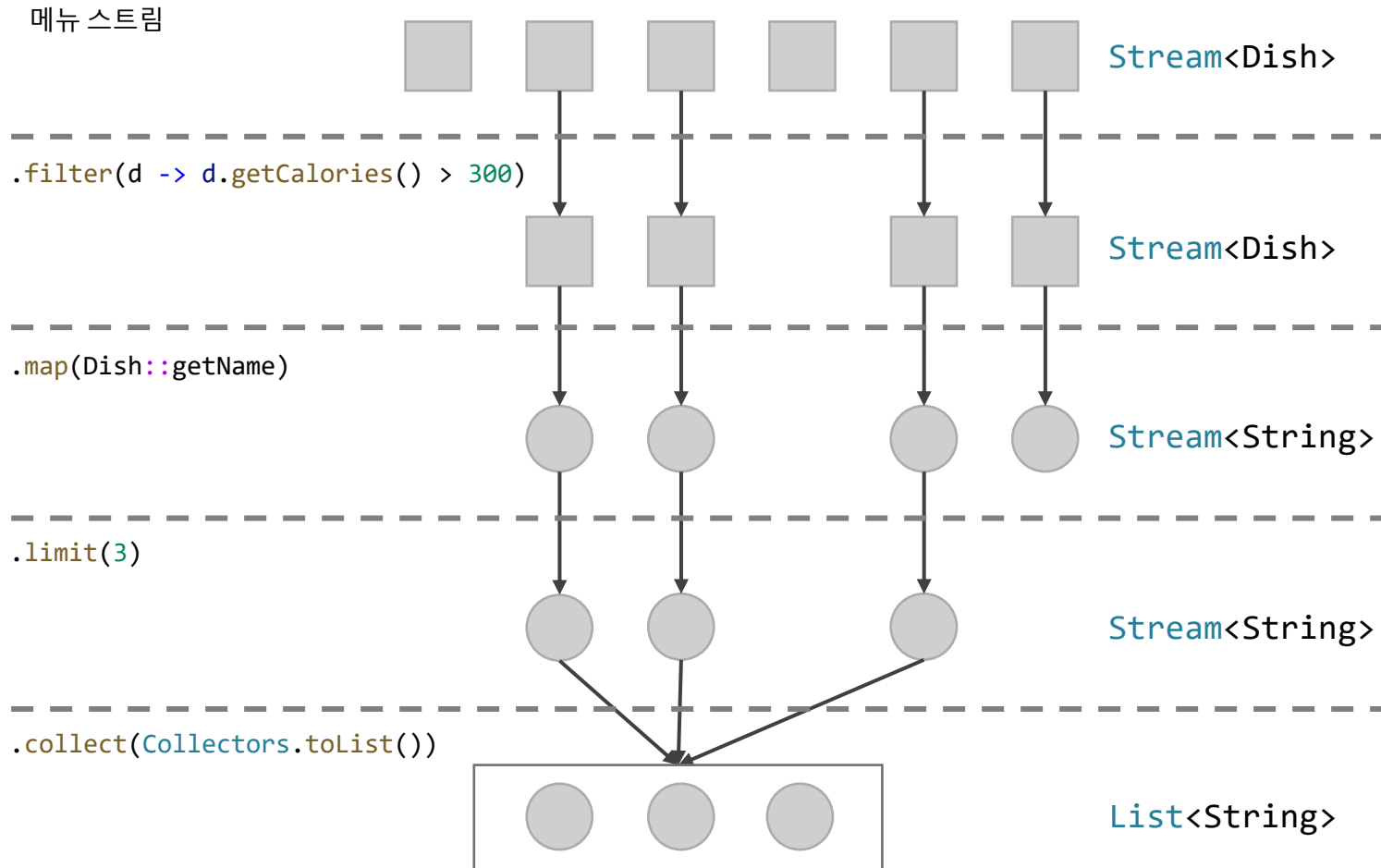


- 아래와 같은 코드가 있다면

```
List<String> threeHighCaloricDishNames =  
    menu.stream()  
        // 300 칼로리 이상의 요리 선택  
        .filter(d -> d.getCalories() > 300)  
        // 요리명 추출  
        .map(Dish::getName)  
        // 선착순 3개만 선택  
        .limit(3)  
        // 모든 요리명을 리스트에 저장  
        .collect(Collectors.toList());
```

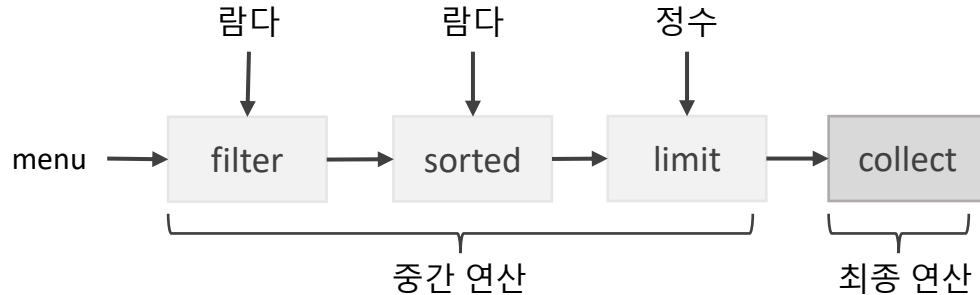
2. Stream

- 스트림은 이렇게 처리한다.



2. Stream

- 스트림의 연산 종류



- 아래에서 중간 연산과 최종 연산은 무엇일까?

```
List<String> names =  
    menu.stream()  
        .filter(d -> d.getCalories() > 300) // 중간연산  
        .map(Dish::getName) // 중간연산  
        .limit(3) // 중간연산  
        .collect(Collectors.toList()); // 최종연산
```

2. Stream

- 중간 연산과 최종 연산의 종류
 - 중간 연산

연산	형식	반환형식	연산의 인수	함수 디스크립터
filter	중간 연산	Stream<T>	Predicate<T>	T -> boolean
map	중간 연산	Stream<T>	Function<T, R>	T -> R
limit	중간 연산	Stream<T>		
sorted	중간 연산	Stream<T>	Comparator<T>	(T, T) -> int
distinct	중간 연산	Stream<T>		

- 최종 연산

연산	형식	목적
forEach	최종 연산	스트림의 각 요소를 소비하면서 람다를 적용한다. void를 반환한다.
count	최종 연산	스트림의 요소 개수를 반환한다. long을 반환한다.
collect	최종 연산	스트림을 리듀스해서 리스트, 맵, 정수 형식의 컬렉션을 만든다.

3. Filtering & Slicing

3. Filtering & Slicing

- 필터링과 슬라이싱
 - 필터링

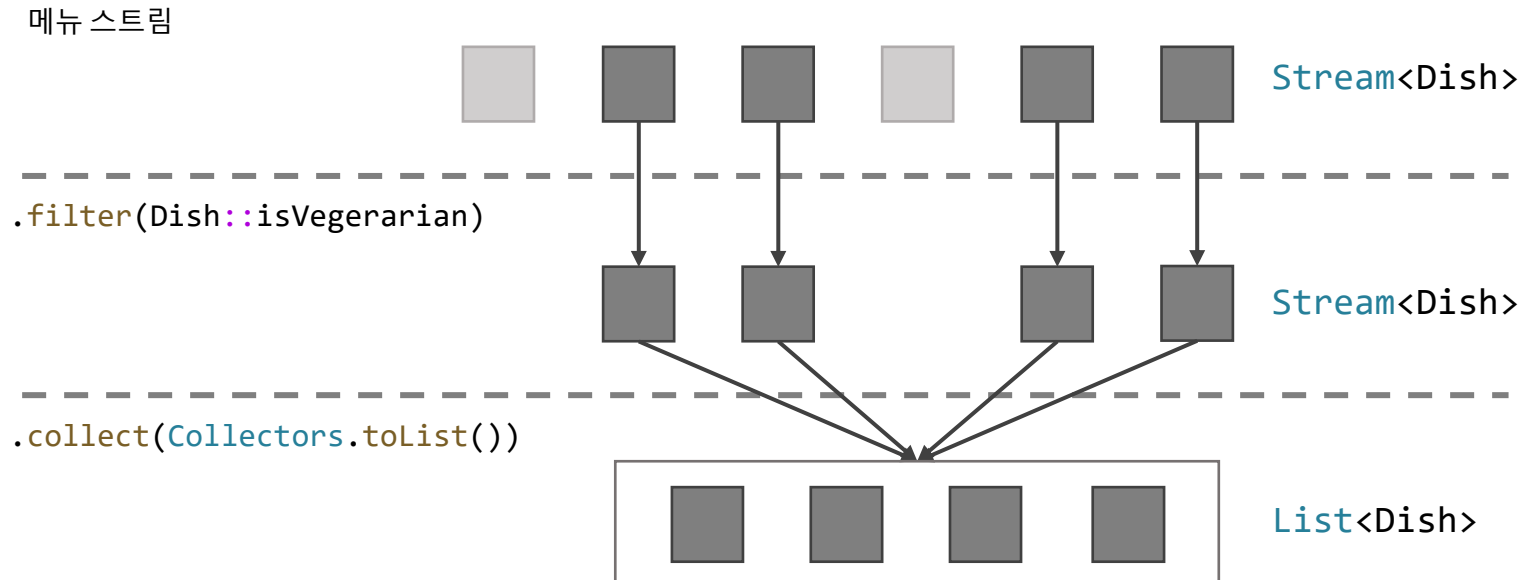
```
List<Dish> vegetarianMenu =  
    menu.stream()  
        // 채식 요리인지 확인하는 메서드 레퍼런스  
        .filter(Dish::isVegerarian)  
        .collect(Collectors.toList());
```

- 고유 요소 필터링

```
List<Integer> numbers = Arrays.asList(1, 2, 1, 3, 3, 2, 4);  
numbers.stream()  
    .filter(i -> i % 2 == 0)  
    .distinct()  
    .forEach(System.out::println);
```

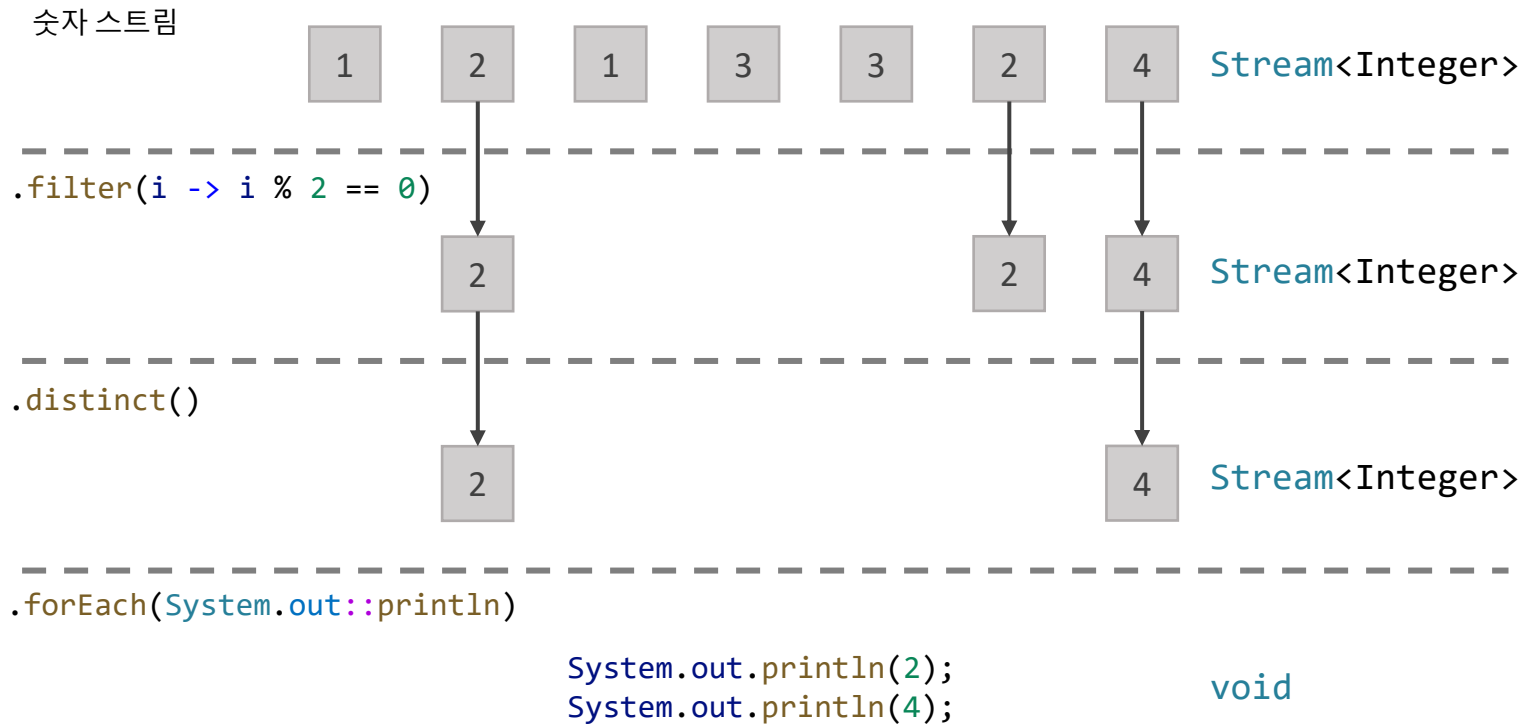
3. Filtering & Slicing

- 필터링과 슬라이싱
 - 동작 원리 – filter



3. Filtering & Slicing

- 필터링과 슬라이싱
 - 동작 원리 – distinct



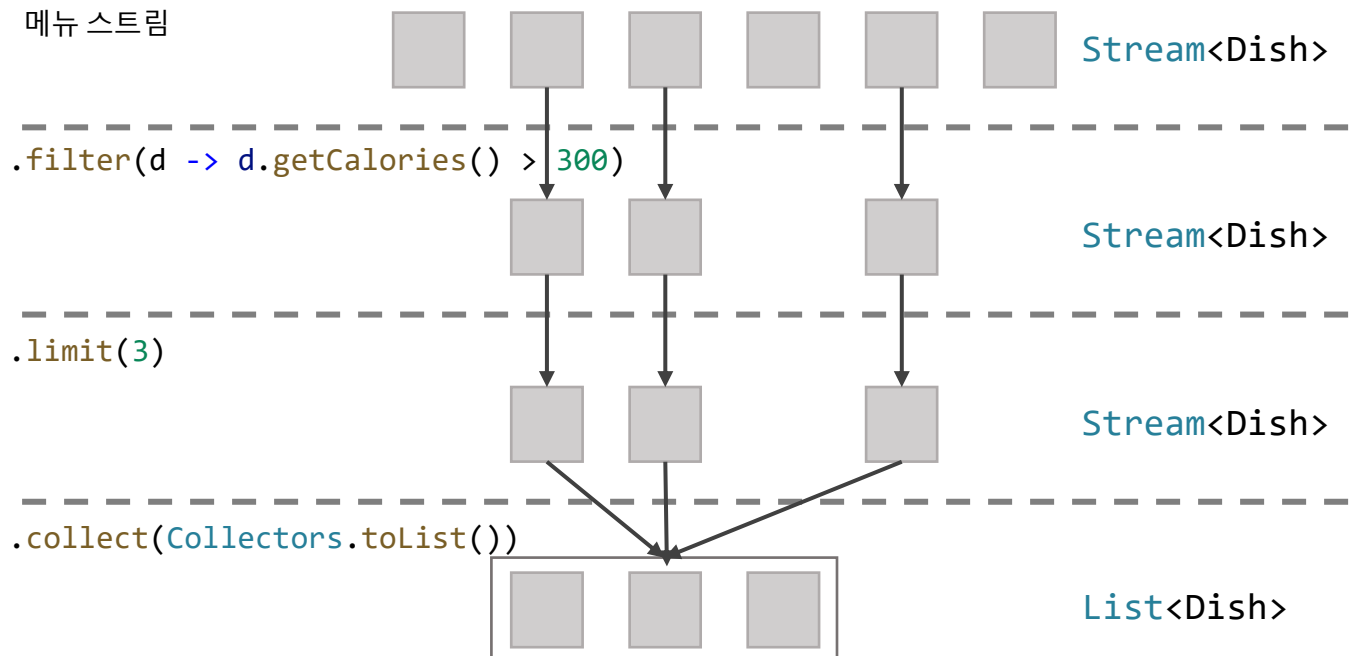
3. Filtering & Slicing

- 필터링과 슬라이싱

- 스트림 축소

```
List<Dish> dishes =  
    menu.stream()  
        .filter(d -> d.getCalories() > 300)  
        .limit(3)  
        .collect(Collectors.toList());
```

- 동작 원리



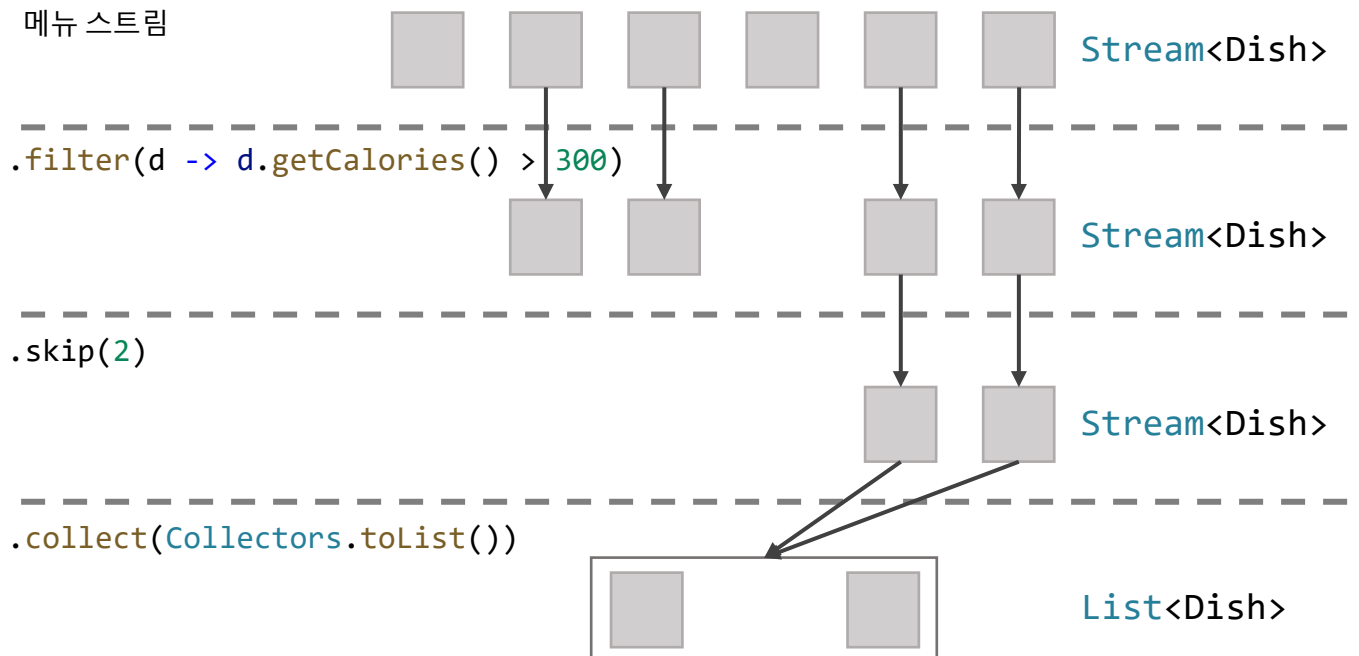
3. Filtering & Slicing

- 필터링과 슬라이싱

- 요소 건너뛰기

```
List<Dish> dishes =  
    menu.stream()  
        .filter(d -> d.getCalories() > 300)  
        .skip(2)  
        .collect(Collectors.toList());
```

- 동작 원리



4. Mapping

4. Mapping

- 맵핑이란?
 - 필터링이 SQL의 WHERE와 같다면, 맵핑은 SELECT 컬럼과 같음.
(단, 하나의 요소만 맵핑할 수 있다)
 - 원본 스트림을 변환하고자 할 때 사용
 - 스트림의 원하는 요소만 추출해 새로운 스트림으로 생성함.
- 예>
- Java8, Lambdas, In, Action가 들어있는 단어 리스트를 이용해
각 글자들의 길이를 갖는 숫자 리스트를 만들.

```
List<String> words = Arrays.asList("Java8", "Lambdas", "In", "Action");  
List<Integer> wordLengths = words.stream()  
                                .map(String::length)  
                                .collect(Collectors.toList());
```

4. Mapping

- flatMap

- 맵과 동일한 동작을 수행함.

```
List<String> uniqueCharacters =  
    words.stream()  
        // 각 단어를 개별문자를 포함하는 배열로 변환  
        .map(w -> w.split(""))  
        // 생성된 스트림을 하나의 스트림으로 평명화  
        .flatMap(Arrays::stream)  
        .distinct()  
        .collect(Collectors.toList());
```

- map 과 flatMap의 차이

- 반환 결과가 stream 일 경우, 차이가 발생함.
 - map : 맵핑된 결과를 Stream 컬렉션에 넣어 반환 함. → Stream<Stream<Object>>
 - flatMap : 맵핑된 결과를 Stream으로 반환 함. → Stream<Object>

5. Find & Matching

5. Find & Matching

- 특정 속성이 데이터 집합에 포함되어있는지 여부를 검색하는 데이터 처리
 - `allMatch`, `anyMatch`, `noneMatch`, `findFirst`, `findAny` 등의 메소드 제공

- 적어도 한 요소와 일치하는지 확인하기.
 - 주어진 스트림 내에서 적어도 한 요소와 일치하는지 확인 후 불린으로 반환
 - `anyMatch` 사용.
 - 메뉴 스트림에 채식요리가 있는지 확인하기.

```
if (menu.stream().anyMatch(Dish::isVegerarian)) {  
    System.out.println("The menu is (somewhat) vegetarian friendly!!");  
}
```

- 모든 요소가 일치하는지 확인
 - 모든 요리가 1000칼로리 이하인지 확인하기

```
boolean isHealthy = menu.stream()  
    .allMatch(d -> d.getCalories() < 1000);
```

5. Find & Matching

- 일치하는 요소가 하나도 없는지 확인
 - allMatch 와 반대 처리

```
boolean isHealthy = menu.stream()  
    .noneMatch(d -> d.getCalories() >= 1000);
```

- 검색하기
 - 스트림 내에서 임의요소 가져오기

```
Optional<Dish> dish = menu.stream()  
    .filter(Dish::isVegetarian)  
    .findAny();
```


5. Find & Matching

- 검색하기

- 스트림 내에서 첫 번째 요소 가져오기

```
List<Integer> someNumbers = Arrays.asList(1, 2, 3, 4, 5);
Optional<Integer> firstSquarDivisibleByThree =
    someNumbers.stream()
                .map(x -> x * x)
                .filter(x -> x % 3 == 0)
                .findFirst();
```

- Optional 이란?

- Null 요소를 유연하게 처리할 수 있는 방법
- 네 가지 기능 제공
 - isPresent() : Optional이 값을 포함하면 true, 아니면 false를 반환
 - isPresent(Consumer<T> block) : Optional에 값이 있을 때만 Consumer를 실행
 - T get() : Optional에 값이 있을 경우 값을 반환,
그렇지 않을 경우 “NoSuchElementException” 발생
 - T orElse(T other) : Optional에 값이 있을 경우 값을 반환. 없을 경우 other 값을 반환

5. Find & Matching

- `findAny`와 `findFirst`의 차이
 - `findAny` : 임의의 요소를 가져옴. 주로 병렬 스트림에서 사용함.
 - 병렬스트림에서 첫 번째 요소의 의미가 존재하지 않음.
 - `findFirst` : 첫 번째 요소를 가져옴. 일반 스트림에서 사용함.

6. Reducing

6. Reducing

- 스트림의 요소로 집계함.
 - SQL 에서 집계함수의 역할 수행함.

- 요소의 합 구하기

- 일반 Java에서 합계를 구하는 코드

```
int sum = 0;
for (int x : numbers) {
    sum += x;
}
```

- 리듀싱을 활용한 코드

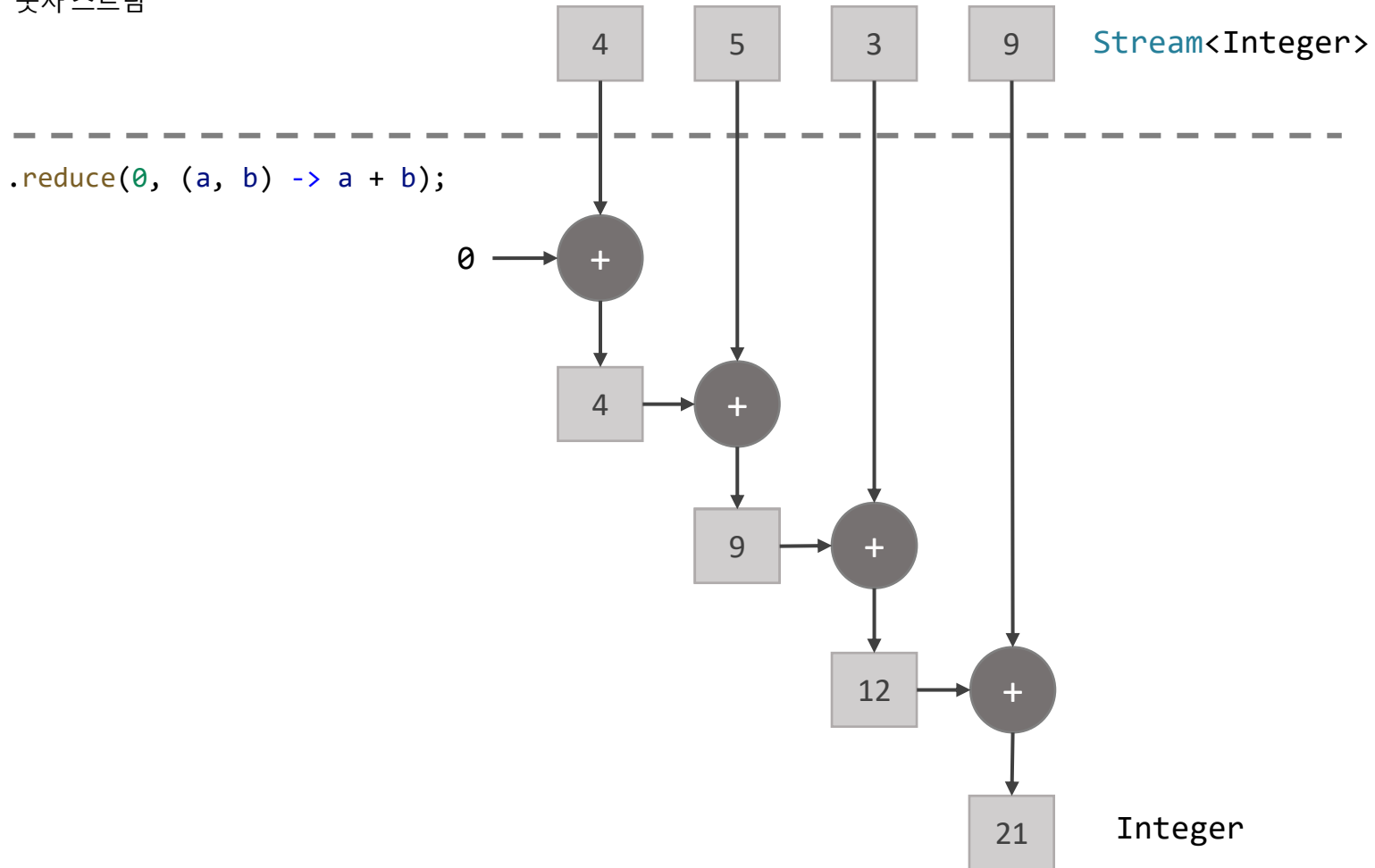
```
int sum = numbers.stream().reduce(0, (a, b) -> a + b);
```

- a: 집계된 값
 - b: 스트림 요소의 값

6. Reducing

- 리듀싱 처리 과정

숫자 스트림



6. Reducing

- 최대값과 최소값

- Integer::max를 이용해 최대값 구하기

```
Optional<Integer> max = numbers.stream().reduce(Integer::max);
```

- Integer::min을 이용해 최소값 구하기

```
Optional<Integer> min = numbers.stream().reduce(Integer::min);
```

- 리듀스에서 기본값이 없을 경우 Optional이 반환됨.

6. Reducing

- 스트림의 중간 연산과 최종 연산

연산	형식	반환 형식	사용된 함수형 인터페이스 형식	함수 디스크립터
filter	중간 연산	Stream<T>	Predicate<T>	T-> boolean
distinct	중간 연산	Stream<T>		
skip	중간 연산	Stream<T>	Long	
limit	중간 연산	Stream<T>	Long	
map	중간 연산	Stream<T>	Function<T, R>	T -> R
flatMap	중간 연산	Stream<T>	Function<T, Stream<R>>	T -> Stream<R>
sorted	중간 연산	Stream<T>	Comparator<T>	(T, T) -> int
anyMatch	최종 연산	boolean	Predicate<T>	T -> boolean
noneMatch	최종 연산	boolean	Predicate<T>	T -> boolean
allMatch	최종 연산	boolean	Predicate<T>	T -> boolean
findAny	최종 연산	Optional<T>		
findFirst	최종 연산	Optional<T>		
forEach	최종 연산	void	Consumer<T>	T -> void
collect	최종 연산	R	Collector<T, A, R>	
reduce	최종 연산	Optional<T>	BinaryOperator<T>	(T, T) -> T
count	최종 연산	long		

7. Primitive Stream

7. Primitive Stream

- int, double, long 특화 스트림 제공
 - 리듀싱 관련 메소드 제공
 - 기본형 특화 스트림 → 일반 스트림 변환 메소드 제공

- 숫자 스트림으로 맵핑하기

```
int calories = menu.stream() // Stream<Dish> 반환
                  .mapToInt(Dish::getCalories) // IntStream 반환
                  .sum();
```

- mapToInt(), mapToDouble(), mapToLong()을 많이 사용함.

- 객체 스트림으로 복원하기

```
// 스트림을 숫자 스트림으로 변환
IntStream intStream = menu.stream().mapToInt(Dish::getCalories);
// 숫자 스트림을 일반 스트림으로 변환
Stream<Integer> stream = intStream.boxed();
```

7. Primitive Stream

- 기본값 처리

- OptionalInt, OptionalDouble, OptionalLong 제공
- 스트림에 요소가 없는 경우 최대값을 구하는 상황

```
OptionalInt maxCalories = menu.stream()  
                                .mapToInt(Dish::getCalories)  
                                .max();
```

- Optional 의 orElse 를 사용할 수도 있다.

```
// 값이 없을 때 기본 최대값을 명시적으로 설정  
int max = maxCalories.orElse(1);
```

8. Generate Stream

8. Generate Stream

- 값으로 스트림 만들기

```
Stream<String> stream = Stream.of("Java8", "Lambdas", "In", "Action");  
stream.map(String::toUpperCase).forEach(System.out::println);
```

- 스트림 비우기

```
Stream<String> emptyStream = Stream.empty();
```

- 배열로 스트림 만들기

```
int[] numbers = {2, 3, 5, 7, 11, 13};  
int sum = Arrays.stream(numbers).sum(); // 41
```

8. Generate Stream

- 파일로 스트림 만들기

```
long uniqueWords = 0;
Path filePath = Paths.get("data.txt");
Charset utf8 = Charset.defaultCharset();
// 스트림은 자원을 자동으로 Close 할 수 있는 AutoClosable이다.
try (Stream<String> lines = Files.lines(filePath, utf8)) {
    // 단어 스트림 생성
    uniqueWords = lines.flatMap(line -> Arrays.stream(line.split(" ")))
        .distinct() // 중복제거
        .count(); // 고유 단어 수 계산
}
catch (IOException ioe) {
    // 파일을 열다가 에러가 발생시 처리
}
```

9. Collectors

9. 데이터 집계

9. 데이터 집계

- 스트림으로 집계하기
 - Java 8 이 제공하는 “컬렉터(Collectors 클래스)”를 통해 아래 기능들을 사용할 수 있음.
 - 1. 스트림 요소를 하나의 값으로 리듀스하고 요약하기
 - 2. 스트림 요소 그룹화
 - 3. 스트림 요소 분할

- 스트림 요소 카운팅하기

```
long howManyDishes = menu.stream().collect(Collectors.counting());
```

- 아래처럼 스트림에서 바로 사용할 수도 있다.

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


9. 데이터 집계

- 스트림 요소에서 최대값과 최소값 검색하기

- maxBy와 minBy 를 사용할 수 있음.

```
Comparator<Dish> dishCaloriesComparator =  
    Comparator.comparingInt(Dish::getCalories);  
  
Optional<Dish> mostCalorieDish =  
    menu.stream()  
        .collect(Collectors.maxBy(dishCaloriesComparator));
```

- 총 합 구하기

- summingInt(), summingDouble(), summingLong() 제공

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

- 총 평균 구하기

- averagingInt(), averagingDouble(), averagingLong() 제공

```
double avgCalories = menu.stream()  
    .collect(Collectors.averagingInt(Dish::getCalories));
```

9. 데이터 집계

- 모든 집계 구하기
 - 스트림의 요소수, 주어진 조건의 합계, 평균, 최대값, 최소값을 한번에 찾아줌
 - summarizingInt(), summarizingDouble(), summarizingLong() 제공

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

- 결과

```
IntSummaryStatistics {  
    count=0, sum=4300, min=120, average=477.777778, max=800  
}
```

9. 문자열 연결

9. 문자열 연결

- 스트림 내의 문자열을 모두 연결하는 `joining()`

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

- 아래처럼 할 수도 있다.

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

9. 그룹화

9. 그룹화

- 스트림 요소를 하나 이상의 특성으로 분류해 그룹화 함.
 - SQL의 GROUP BY와 같은 연산
 - `Collectors.groupingBy()` 사용

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

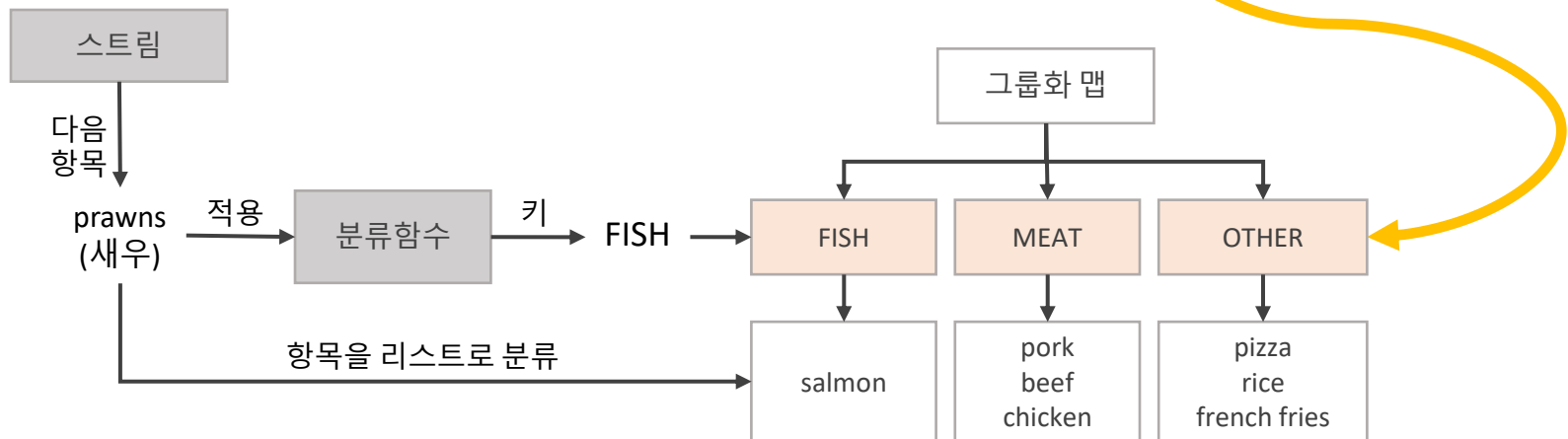
- 결과

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

9. 그룹화

- 그룹화가 동작하는 방식

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



9. 그룹화

- 그룹 키 커스텀하기

```
public enum CaloricLevel { DIET, NORMAL, FAT }
```

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


9. 다수준 그룹화

9. 다수준 그룹화

- Collectors.groupingBy() 를 이용한 다수준 그룹화

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

- 결과

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

9. 서브그룹 그룹화

9. 서브그룹 그룹화

- 메뉴 별 요리의 개수 구하기

```
Map<Dish.Type, Long> typesCount =  
    menu.stream()  
        .collect(Collectors.groupingBy(Dish::getType,  
            Collectors.counting()));
```

- 결과

```
{MEAT=3, FISH=2, OTHER=4}
```

- 가장 높은 칼로리를 가진 메뉴 구하기

```
Map<Dish.Type, Optional<Dish>> mostCaloricByType =  
    menu.stream()  
        .collect(Collectors.groupingBy(Dish::getType,  
            Collectors.maxBy(Comparator.comparingInt(Dish::getCalories))));
```

데이터가 존재하는지 확실히 알지 못하기 때문에,
Optional을 리턴한다.

- 결과

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

9. 서브그룹 그룹화

- 가장 높은 칼로리를 가진 메뉴 구하기 – Optional 제거

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

분류함수

감싸인 컬렉터

변환 함수

- 결과

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

9. 분할

9. 분할

- 스트림을 분할

```
Map<Boolean, List<Dish>> partitionedMenu =  
    // Dish의 isVegetarian 을 기준으로 분할  
    menu.stream().collect(Collectors.partitioningBy(Dish::isVegetarian));
```

- 결과

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

- 결과로 채식 요리만 가져옴.

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

- 필터(filter())로도 같은 기능을 수행할 수 있다.

감사합니다.

Java Stream Programming

장민창

mcjang@hucloud.co.kr