

# 모던 자바 인 액션 스터디 - 3주차

chanE

# 목차

1. 블로그에서 정보를 얻는 방법

2. 람다 연습문제 토론해보기

3. 모던 자바 인 액션 3장 + 4장

4. Q&A

# 블로그에서 정보를 얻는 방법

## 과연 신뢰할 만한 글인가?

모르는 개념이나 에러를 만났을 때, 우리는 보통 구글에 관련 키워드를 검색합니다.

Google

effectively final

동영상 이미지 뉴스 Java 도서 쇼핑 지도 항공편 금융

검색결과 약 2,240,000,000개 (0.33초)

수정된 검색어에 대한 결과: **effectively final**  
다음 검색어로 대신 검색: **effectively final**

**자바의 effectively final**

2021. 3. 9. — 자바에서 **final**로 선언되지 않았지만 초기화된 이후 참조가 변경되지 않아 **final**처럼 동작하는.

**codechacha.com**  
**Java의 Effectively final이란 무엇인가? - codechacha**

Java8에서 final이 붙지 않은 변수의 값이 변경되지 않는다면, 그 변수를 **Effectively final**이라고 ...  
**Effectively final**은 익명 클래스 내부에서 접근할 수 있습니다.

**velog**  
**[Java] Effectively Final이란?**

2022. 6. 13. — 남다른 람다 정리를 넘어 **Effectively Final**에 대하여 정리해 보았습니다. 그리고 람다도 함께 정리해 보았습니다.  
**Effectively Final?** 🔥 · 왜 이렇게 사용할까? 🤔

**vagabond95.me**  
https://vagabond95.me › posts › lambda-with-final

# 블로그에서 정보를 얻는 방법

## 과연 신뢰할 만한 글인가?

2주차 과제였던 첫 번째 글에서, **effectively final**에 대한 정보를 얻을 수 있었습니다.  
그렇다면, 해당 글의 내용은 전부 맞는 내용일까요 ?

블로그 내용 중 예제로 사용된 소스 코드를 살펴보겠습니다.

```
List<Person> personList = List.of(new Person(2), new Person(3));
for (Person p : personList) {
    p.setId(2);
    personList.removeIf(o -> o.getId() == p.getId());
}
```

이 코드는 정상적으로 동작할까요?

# 블로그에서 정보를 얻는 방법

## 과연 신뢰할 만한 글인가?

2주차 과제였던 첫 번째 글에서, **effectively final**에 대한 정보를 얻을 수 있었습니다.  
그렇다면, 해당 글의 내용은 전부 맞는 내용일까요 ?

블로그 내용 중 예제로 사용된 소스 코드를 살펴보겠습니다.

```
List<Person> personList = List.of(new Person(2), new Person(3));
for (Person p : personList) {
    p.setId(2);
    personList.removeIf(o -> o.getId() == p.getId());
}
```

List.of로 만든 List의 removeIf 를 호출하고 있으므로, **UnsupportedOperationException**이 발생할 것입니다.

List.of 에 대해서 정확히 알고 있지 않더라도, 코드를 실행해 보았다면 런타임 예외(언체크 예외)가 발생한 것을 확인했을 것입니다.

# 블로그에서 정보를 얻는 방법

## 믿을만한 정보인지 판단하는 방법

그렇다면 이런 생각을 할 수 있습니다.

유명하고 뛰어난 실력을 가진 개발자(ex: 향로님, 오명운님 등)께서 쓰신 글은 믿을 만 하다. 이런 글을 제외하고는 내가 직접 검증을 해보자.

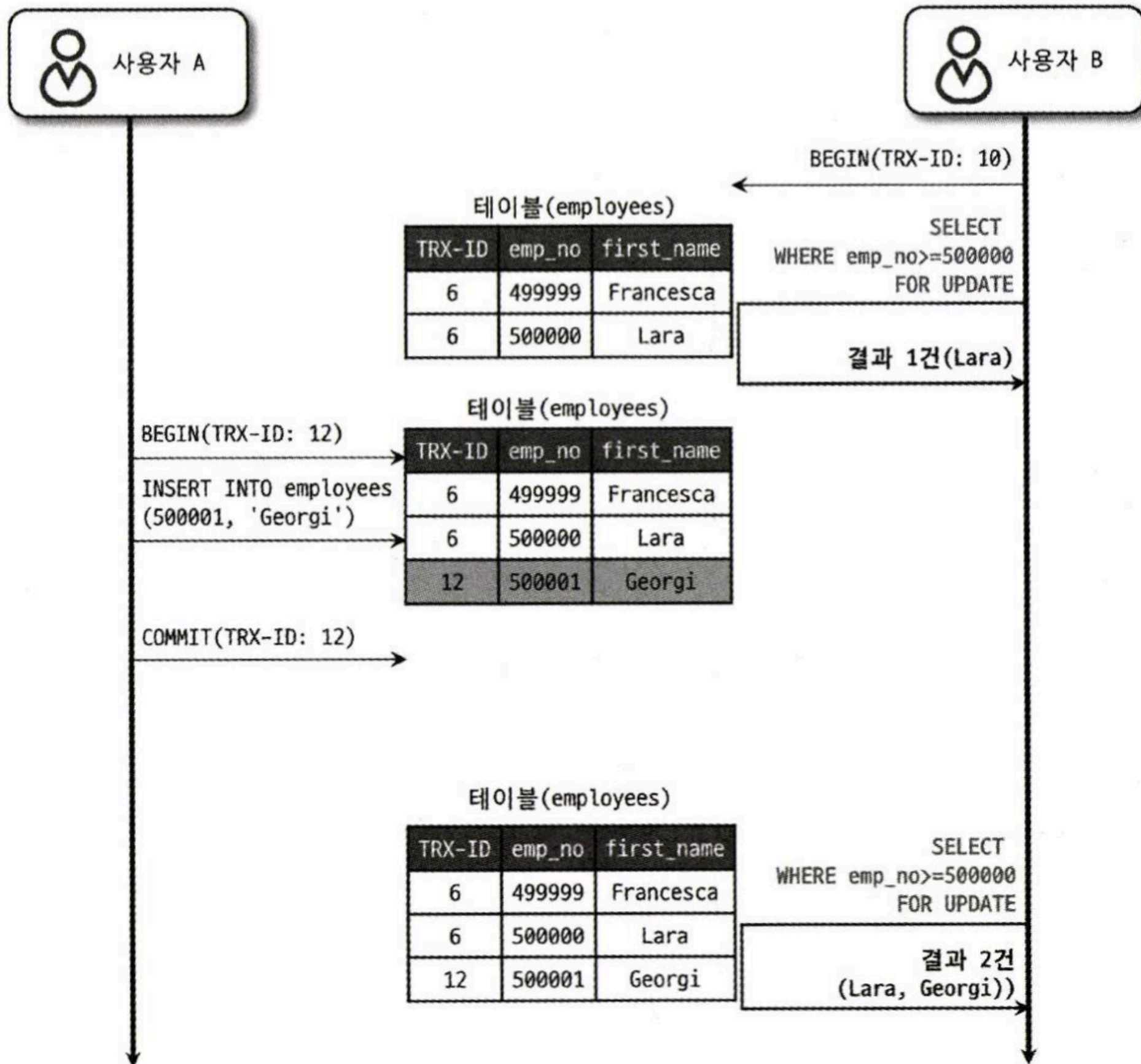
과연 그럴까요.. ?

# 블로그에서 정보를 얻는 방법

## 믿을만한 정보인지 판단하는 방법

책은 믿을 수 있을까요 ? **Real MySQL 8.0** I 권을 잠깐 살펴봅시다.

MySQL의 InnoDB 스토리지 엔진에서 기본으로 사용하는 격리 수준인 REPEATABLE READ에서 PHANTOM READ(다른 트랜잭션에서 변경한 작업에 의해 레코드가 보였다 안 보였다 하는 현상)가 발생하는 현상에 대해서 그림과 함께 설명하고 있습니다.



터미널을 두 개 열어서 확인해보겠습니다.



# 블로그에서 정보를 얻는 방법

## 믿을만한 정보인지 판단하는 방법

이상하게도 예제 코드가 작동하지 않습니다.

메일을 보내볼까 하다가.. 마지막으로 **정오표**를 확인합니다.



사용자 A



사용자 B

BEGIN (TRX-ID : 10)

테이블 (employees)

TRX-ID	emp_no	first_name
6	499999	Francesca
6	500000	Lara

SELECT  
WHERE emp\_no>=500000

..??

결과 1건 (Lara)

BEGIN (TRX-ID : 12)

INSERT INTO employees  
(500001, 'Georgi')

테이블 (employees)

TRX-ID	emp_no	first_name
6	499999	Francesca
6	500000	Lara
12	500001	Georgi

COMMIT (TRX-ID : 12)

그다음 다른트랜잭션에서

오후 6:10

INSERT 치는순간 50초의 타임아웃이 걸려버려

오후 6:10

김동호

inser

오후 6:10

김동호

못친다는거지

오후 6:11

테이블 (employees)

TRX-ID	emp_no	first_name
6	499999	Francesca
6	500000	Lara
12	500001	Georgi

SELECT  
WHERE emp\_no>=500000  
FOR UPDATE

결과 2건  
(Lara, Georgi)

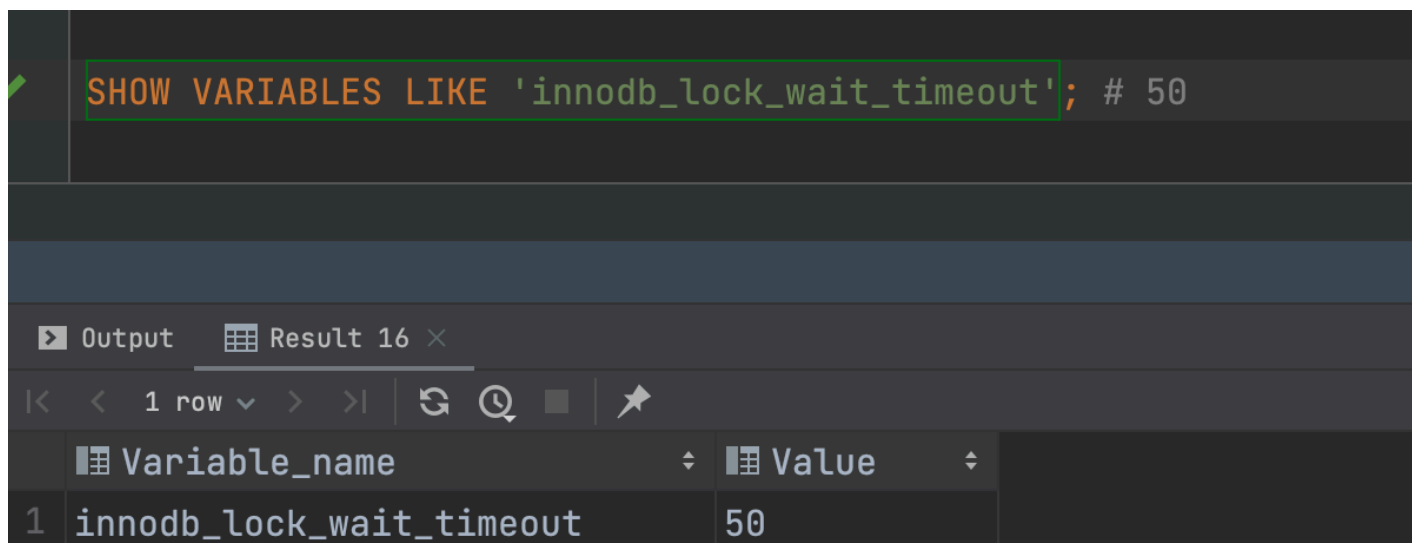
# 블로그에서 정보를 얻는 방법

## 왜 50초의 타임아웃이 걸렸을까?

코드를 검증하는 것만으로도, 또 다른 지식을 얻을 수 있습니다.

왜 50초의 타임아웃이 발생했을까 하는 의문이 다른 지식의 습득으로 이어집니다.

그 이유는 **innodb\_lock\_wait\_timeout**의 기본값 때문입니다.



```
SHOW VARIABLES LIKE 'innodb_lock_wait_timeout'; # 50
```

Variable_name	Value
innodb_lock_wait_timeout	50

InnoDB 스토리지 엔진은 내부적으로 잠금이 교착 상태에 빠지지 않았는지 체크하기 위해 잠금 대기 목록을 그래프 (Wait-for List) 형태로 관리한다. InnoDB 스토리지 엔진은 데드락 감지 스레드를 가지고 있어서 데드락 감지 스레드가 주기적으로 잠금 대기 그래프를 검사해 교착 상태에 빠진 트랜잭션들을 찾아서 그중 하나를 강제 종료한다.

...

**innodb\_lock\_wait\_timeout**은 초 단위로 설정할 수 있으며, 잠금을 설정한 시간동안 획득하지 못하면 쿼리는 실패하고 에러를 반환한다.

# 블로그에서 정보를 얻는 방법

믿을만한 정보인지 판단하는 방법

그렇다면.. 공식 **API** 문서는 믿을 수 있을까요.. ?

# 블로그에서 정보를 얻는 방법

## 믿을만한 정보인지 판단하는 방법

모든 글에 대해서 스스로 검증할 수 있다면 가장 좋겠지만, 우리에게 시간은 제한되어 있습니다.

그렇다면 어떤 기준을 세울 수 있을까요?

저 나름으로는 **다음과 같은 규칙**을 만들고 지키려고 노력합니다.

- 1 공식 API 문서의 경우(ex: 오라클 등) 검증하지 않고 이해하기
- 2 그 이외의 모든 글은 검증하기

### 검증하는 방법

- 모든 예제 코드를 실행시켜보기
- 저자의 설명이 맞는지 확인하기(실제로 구현 코드를 확인하고 비교하기)
- 출처 확인하기

검증을 통해서, **추가적인 정보와 이해력**을 높일 수 있습니다.

# 람다 연습문제 토론해보기

<https://github.com/GDSC-Hongik/2023-2-OC-Java-Study/pulls>

# 모던 자바 인 액션 3장

## 메서드 참조

메서드 참조는 **특정 메서드만을 호출하는 람다의 축약형**이라고 생각할 수 있습니다. 명시적으로 메서드명을 참조함으로써 **가독성을 높일 수** 있습니다.

메서드 참조는 메서드명 앞에 구분자(::)를 붙이는 방식으로 메서드 참조를 활용할 수 있습니다. `Apple::getWeight` 람다 표현식 (`Apple a`) -> `a.getWeight()`를 축약한 것입니다.

**Table 3.4** Examples of lambdas and method reference equivalents

Lambda	Method reference equivalent
<code>(Apple apple) -&gt; apple.getWeight()</code>	<code>Apple::getWeight</code>
<code>() -&gt; Thread.currentThread().dumpStack()</code>	<code>Thread.currentThread()::dumpStack</code>
<code>(str, i) -&gt; str.substring(i)</code>	<code>String::substring</code>
<code>(String s) -&gt; System.out.println(s)</code> <code>(String s) -&gt; this.isValidName(s)</code>	<code>System.out::println</code> <code>this::isValidName</code>

# 모던 자바 인 액션 3장

## 람다 표현식을 조합할 수 있는 유용한 메서드

여러 개의 람다 표현식을 조합해서 복잡한 람다 표현식을 만들 수 있습니다.

예를 들어 두 프레디케이트를 조합해서 두 프레디케이트의 or 연산을 수행하는 커다란 프레디케이트를 만들 수 있습니다. 여기서 등장하는 것이 바로 **디폴트 메서드**입니다.

```
@Contract(pure = true) @NotNull
default Predicate<T> or( @NotNull Predicate<? super T> other) {
    Objects.requireNonNull(other);
    return (t) → test(t) || other.test(t);
}
```



# 모던 자바 인 액션 3장

## Comparator 조합

사과의 무게를 내림차순으로 정렬하고 싶다면? Comparator에 정의된 reversed 메서드를 사용하면 됩니다.

```
Comparator<Apple> c = Comparator.comparing(Apple::getWeight);  
// 역정렬  
inventory.sort(Comparator.comparing(Apple::getWeight).reversed());
```

그렇다면 무게가 같은 두 사과가 존재할 때 어떻게 해야할까요?

이럴 땐 비교 결과를 더 다듬을 수 있는 두 번째 Comparator를 만들 수 있습니다. 예를 들어 무게로 두 사과를 비교한 다음에 무게가 같다면 원산지 국가별로 사과를 정렬할 수 있습니다.

이럴 때는 thenComparing 메서드를 활용할 수 있습니다.

```
inventory.sort(Comparator.comparing(Apple::getWeight)  
    .reversed()  
    .thenComparing(Apple::getColor));
```

# 모던 자바 인 액션 3장

## Predicate 조합

Predicate 인터페이스는 복잡한 프레디케이트를 만들 수 있도록 negate, and, or 세 가지 메서드를 제공합니다. 예를 들어 ‘빨간색이 아닌 사과’ 처럼 특정 프레디케이트를 반전시킬 때 negate 메서드를 사용할 수 있습니다.

```
Predicate<Apple> redApple = (Apple a) → Color.RED == a.getColor();  
Predicate<Apple> notRedApple = redApple.negate();
```

또한 and 메서드를 이용해서 빨간색이면서 무거운 사과를 선택하도록 두 람다를 조합할 수 있습니다.

```
Predicate<Apple> redAndHeavyApple =  
    redApple.and((Apple a) → a.getWeight() > 150);
```

그뿐만 아니라 or을 이용해서 ‘빨간색이면서 무거운(150그램 이상) 사과 또는 그냥 녹색 사과’ 등 다양한 조건을 만들 수 있습니다.

```
Predicate<Apple> redAndHeavyAppleOrGreen = redApple.and(apple → apple.getWeight() > 150)  
    .or(apple → Color.GREEN == apple.getColor());
```

여기서 소개한 and, or 등은 왼쪽에서 오른쪽으로 연결됩니다. 즉 a.or(b).and(c)는 (a || b) && c와 같습니다.

# 모던 자바 인 액션 3장

## Function 조합

Function 인터페이스는 Function 인스턴스를 반환하는 `andThen`, `compose` 두 가지 디폴트 메서드를 제공합니다.

`andThen` 메서드는 주어진 함수를 먼저 적용한 결과를 다른 함수의 입력으로 전달하는 결과를 반환합니다.

```
Function<Integer, Integer> f = x → x + 1;
Function<Integer, Integer> g = x → x * 2;
Function<Integer, Integer> h = f.andThen(g); // g(f(x)), f를 먼저 적용하고 g를 적용한다.
int result = h.apply(t: 1); // 4
System.out.println(result);
```

`compose` 메서드는 인수로 주어진 함수를 먼저 실행한 다음에 그 결과를 외부 함수의 인수로 제공합니다.

```
Function<Integer, Integer> f = x → x + 1;
Function<Integer, Integer> g = x → x * 2;
Function<Integer, Integer> h = f.compose(g); // f(g(x)), 인수로 주어진 함수를 먼저 실행한다
int result = h.apply(t: 1); // 3
```

# 모던 자바 인 액션 3장

```
public class Letter { Complexity is 5 Everything is cool!
```

```
    public static String addHeader(final String text) { 1 usage  
        return "From Raoul, Mario and Alan: " + text;  
    }
```

Complexity is 3 Ev

```
    public static String addFooter(final String text) { 1 usage  
        return text + " Kind regards";  
    }
```

Complexity is 3 Ev

```
    public static String checkSpelling(final String text) { 1 usage  
        return text.replaceAll(regex: "labda", replacement: "lambda");  
    }
```

```
    public static void main(String[] args) { Complexity is 4 Everything is cool!  
        Function<String, String> addHeader = Letter::addHeader; // 헤더를 추가한 다음에  
        Function<String, String> transformationPipeline =  
            addHeader.andThen(Letter::checkSpelling) // 철자 검사를 하고  
            .andThen(Letter::addFooter); // 마지막에 푸터를 추가할 수 있다.  
    }  
}
```

# 모던 자바 인 액션 3장

`Objects.requireNonNull`

<https://hudi.blog/java-requirenonnull/>

# 모던 자바 인 액션 4장

## 선언형 연산 표현

```
SELECT name FROM dishes WHERE calorie < 400;
```

칼로리가 낮은 요리명을 선택하라는 SQL 질의에서, 요리의 속성을 이용하여 어떻게 필터링할 것인지는 구현할 필요가 없습니다. (예를 들어 자바처럼 반복자, 누적자 등을 이용할 필요가 없습니다)

SQL에서는 질의를 어떻게 구현해야 할지 명시할 필요가 없으며 구현은 **자동으로** 제공됩니다.

컬렉션으로도 이와 비슷한 기능을 만들 수 있지 않을까요?

이 질문의 답은 **스트림**입니다.

# 모던 자바 인 액션 4장

## 스트림이란 무엇인가?

자바 8 API에 새로 추가된 기능입니다.

스트림을 이용하면 선언형(데이터를 처리하는 임시 구현 코드 대신 **질의**로 표현)으로 컬렉션 데이터를 처리할 수 있습니다.

예제를 통해 살펴보겠습니다. 다음 예제는

1. 저칼로리의 요리명을 반환하고
2. 칼로리를 기준으로 요리를 정렬합니다.

**Java 7**을 기준으로 이 요구사항을 구현해보겠습니다.

# Java 7

```
public class LowCalories { Complexity is 9 It's time to do something...
}
    /**
     * 1. 저칼로리의 요리명을 반환하고,
     * 2. 칼로리를 기준으로 요리를 정렬한다.
     */
    public static void main(String[] args) { Complexity is 8 It's time to do something...
        /**
         * Java7
         */
        List<Dish> lowCaloricDishes = new ArrayList<>(); 가비지 변수
        for (Dish dish : menu) {
            if (dish.getCalories() < 400) {
                lowCaloricDishes.add(dish);
            }
        }
        Collections.sort(lowCaloricDishes, new Comparator<Dish>() { Complexity is 3 Ever
            @Override
            public int compare(final Dish dish1, final Dish dish2) {
                return Integer.compare(dish1.getCalories(), dish2.getCalories());
            }
        });
        List<String> lowCaloricDishesName = new ArrayList<>();
        for (Dish dish : lowCaloricDishes) {
            lowCaloricDishesName.add(dish.getName());
        }
    }
}
```



```
/**
 * Java8
 */
List<String> lowCaloricDishesName = menu.stream() Stream<Dish>
    .filter(dish → dish.getCalories() < 400)
    .sorted(Comparator.comparing(Dish::getCalories))
    .map(Dish::getName) Stream<String>
    .collect(Collectors.toList());
```

다음은 Java 8의 스트림을 이용해서 구현한 코드입니다. 지금까지 배웠던 동작 파라미터화와 선언형 코드를 활용하면 변하는 요구사항에 쉽게 대응할 수 있습니다.

filter, sorted, map, collect 같은 여러 빌딩 블록 연산을 연결해서 복잡한 데이터 처리 파이프라인을 만들 수 있습니다. filter 메서드의 결과는 sorted로, 다시 sorted 결과는 map 메서드로, map 메서드의 결과는 collect로 연결됩니다.

어떻게 가능할까요 ?

# 모던 자바 인 액션 4장

## 스트림이란 무엇인가?

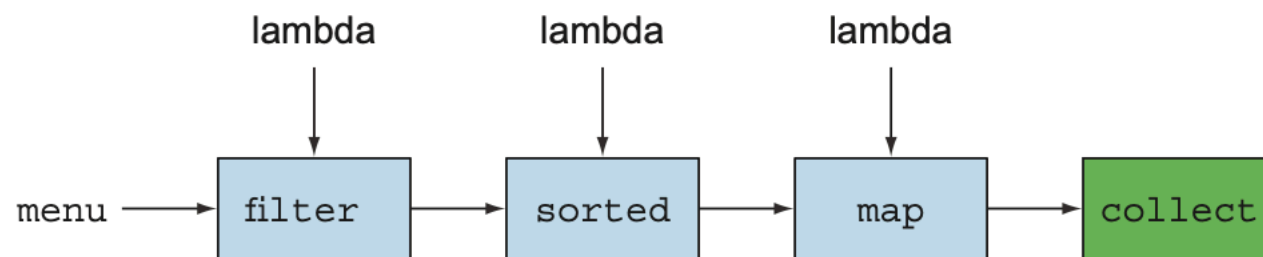


Figure 4.1 Chaining stream operations forming a stream pipeline

filter (또는 sorted, map, collect)와 같은 연산은 **고수준 빌딩 블록**으로 이루어져 있으므로 특정 스레딩 모델에 제한되지 않고 자유롭게 어떤 상황에서도 사용할 수 있습니다.

또한 이들은 내부적으로 단일 스레드 모델에 사용할 수 있지만 멀티코어 아키텍처를 최대한 투명하게 활용할 수 있게 구현되어 있습니다.

결과적으로 데이터 처리 과정을 병렬화하면서 **스레드**와 **락**을 걱정할 필요가 없습니다.

스트림의 특징

- 선언형: 더 간결하고 가독성이 좋아진다.
- 조립할 수 있음: 유연성이 좋아진다.
- 병렬화: 성능이 좋아진다.

# 모던 자바 인 액션 4장

## 스트림 시작하기

자바 8 컬렉션에는 스트림을 반환하는 `stream` 메서드가 추가되었습니다.

Returns a sequential Stream with this collection as its source.

This method should be overridden when the `splitterator()` method cannot return a splitterator that is IMMUTABLE, CONCURRENT, or *late-binding*. (See `splitterator()` for details.)

Returns: a sequential Stream over the elements in this collection

Implementation The default implementation creates a sequential Stream from the

Requirements: collection's Splitterator.

Since: 1.8

```
@Contract(pure = true)
```

```
default Stream<E> stream() {  
    return StreamSupport.stream(splitterator(), parallel: false);  
}
```

Returns a possibly parallel Stream with this collection as its source. It is allowable for this method to return a sequential stream.

This method should be overridden when the `splitterator()` method cannot return a splitterator that is IMMUTABLE, CONCURRENT, or *late-binding*. (See `splitterator()` for details.)

Returns: a possibly parallel Stream over the elements in this collection

Implementation The default implementation creates a parallel Stream from the

Requirements: collection's Splitterator.

Since: 1.8

```
@Contract(pure = true)
```

```
default Stream<E> parallelStream() {  
    return StreamSupport.stream(splitterator(), parallel: true);  
}
```

# 모던 자바 인 액션 4장

## 스트림 시작하기

스트림이란 ‘데이터 처리 연산을 지원하도록 소스에서 추출된 연속된 요소(sequence of elements)’로 정의할 수 있습니다.

**연속된 요소:** 컬렉션과 마찬가지로 스트림은 특정 요소 형식으로 이루어진 연속된 값 집합의 인터페이스를 제공합니다. 컬렉션의 주제는 데이터이고, 스트림의 주제는 계산입니다.

**데이터 처리 연산:** 스트림은 함수형 프로그래밍 언어에서 일반적으로 지원하는 연산과 데이터베이스와 비슷한 연산을 지원합니다. 예를 들어 filter, map, reduce, find, match, sort 등으로 데이터를 조작할 수 있습니다.

# 모던 자바 인 액션 4장

## 스트림 시작하기

**소스:** 컬렉션, 배열, I/O 자원 등의 데이터 제공 소스로부터 데이터를 소비합니다. 정렬된 컬렉션으로 스트림을 생성하면 정렬이 그대로 유지됩니다. 즉, 리스트로 스트림을 만들면 스트림의 요소는 리스트의 요소와 같은 순서를 유지합니다.

// 1. 컬렉션

```
List<String> lowCaloricDishesName = menu.stream() Stream<Dish>
    .filter(dish -> dish.getCalories() < 400)
    .sorted(Comparator.comparing(Dish::getCalories))
    .map(Dish::getName) Stream<String>
    .collect(Collectors.toList());
```

// 2. 배열

```
Integer[] numbers = {1, 2, 3, 4, 5};
Arrays.stream(numbers)
    .forEach(System.out::println);
```

// 3. I/O 자원

```
String filePath = "src/main/java/me/euichan/java8/chap04/Source.java";
try (BufferedReader reader = new BufferedReader(new FileReader(filePath))) {
    reader.lines() // return Stream
        .forEach(System.out::println);
} catch (Exception e) {
    e.printStackTrace();
}
```

# 모던 자바 인 액션 4장

## 스트림 시작하기

또한 스트림에는 다음과 같은 두 가지 중요 특징이 있습니다.

**파이프라이닝:** 대부분의 스트림 연산은 스트림 연산끼리 연결해서 커다란 파이프라인을 구성할 수 있도록 스트림 자신을 반환합니다.

**내부 반복:** 반복자를 이용해서 명시적으로 반복하는 컬렉션과 달리 스트림은 **내부 반복**을 지원합니다.

```
List<String> threeHighCaloricDishNames =  
    menu.stream()  
        .filter(dish -> dish.getCalories() > 300)  
        .map(Dish::getName)  
        .limit(3)  
        .collect(Collectors.toList());  
System.out.println(threeHighCaloricDishNames); // [pork, beef, chicken]
```

문제: 각각의 메서드를 수행한 후 어떤 반환 형식을 가지는지 말해주세요. 데이터 소스는 List<Dish> menu입니다.

# 모던 자바 인 액션 4장

## 스트림 시작하기

```
List<String> threeHighCaloricDishNames =  
    menu.stream()  
        .filter(dish -> dish.getCalories() > 300)  
        .map(Dish::getName)  
        .limit(3)  
        .collect(Collectors.toList());  
System.out.println(threeHighCaloricDishNames); // [pork, beef, chicken]
```

데이터 소스:

데이터 소스는 **연속된 요소**를 스트림에 제공합니다.

다음으로 스트림에 filter, map, limit 으로 이어지는 일련의 **데이터 처리 연산**을 적용합니다.

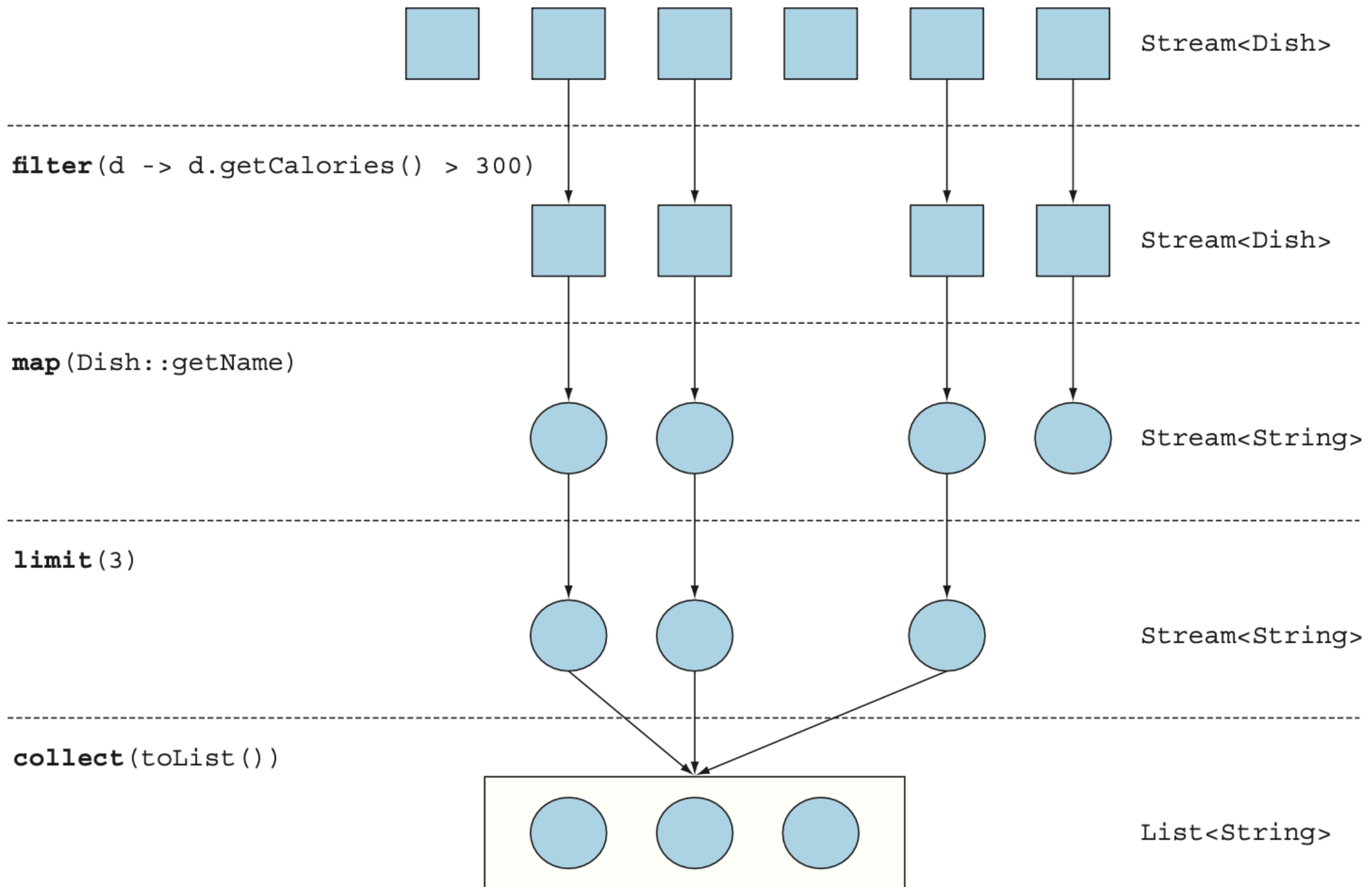
collect를 제외한 모든 연산은 서로 **파이프라인**을 형성할 수 있도록 스트림을 반환합니다.

마지막으로 collect 연산으로 파이프라인을 처리해서 결과를 반환합니다.

마지막에 collect를 호출하기 전까지는 menu에서 무엇도 선택되지 않았으며 출력 결과도 없습니다.

즉, collect가 호출되기 전까지 메서드 호출이 저장되는 효과가 있습니다.

Menu stream



**Figure 4.2** Filtering a menu using a stream to find out three high-calorie dish names



# 모던 자바 인 액션 4장

## 스트림과 컬렉션

데이터를 언제 계산하느냐가 컬렉션과 스트림의 가장 큰 차이이다.

컬렉션 - 컬렉션의 모든 요소는 컬렉션에 추가하기 전에 계산되어야 한다

스트림 - 이론적으로 요청할 때만 요소를 계산하는 고정된 자료구조다(스트림에 요소를 추가하거나 스트림에서 요소를 제거할 수 없다).

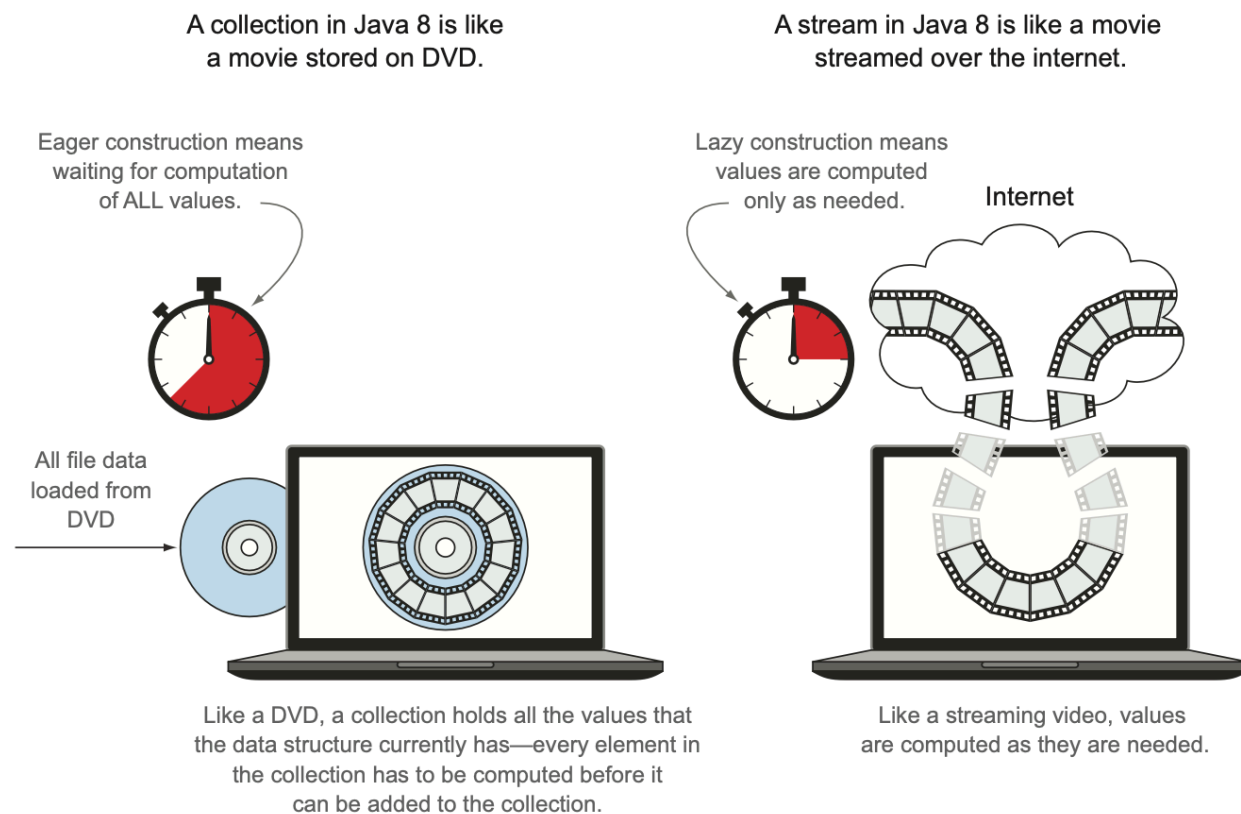


Figure 4.3 Streams versus collections

**DVD** 처럼 컬렉션은 현재 자료구조에 포함된 모든 값을 계산한 다음에 컬렉션에 추가할 수 있다.

**스트리밍 비디오**처럼 스트림은 필요할 때 값을 계산한다.

# 모던 자바 인 액션 4장

## 스트림과 컬렉션

스트림은 단 한번만 소비할 수 있습니다. 두 번 탐색하려고 하면 다음과 같은 예러가 발생합니다.

```
java.lang.IllegalStateException: stream has already been operated upon or closed
```

**한 스트림에서 여러 결과를 얻어야 하는 상황이라면 ?**

만약 메뉴 데이터 모델의 요리 스트림을 탐색하면서 다양한 정보를 얻고 싶다면 ?

그러려면 한 번에 한 개 이상의 람다를 스트림으로 적용해야 합니다.

즉, `fork` 같은 메서드를 이용해서 스트림을 포크(분기)시키고 포크된 스트림에 다양한 함수를 적용해야 합니다.

그러나 자바8의 스트림에서는 이 기능을 제공하지 않습니다.

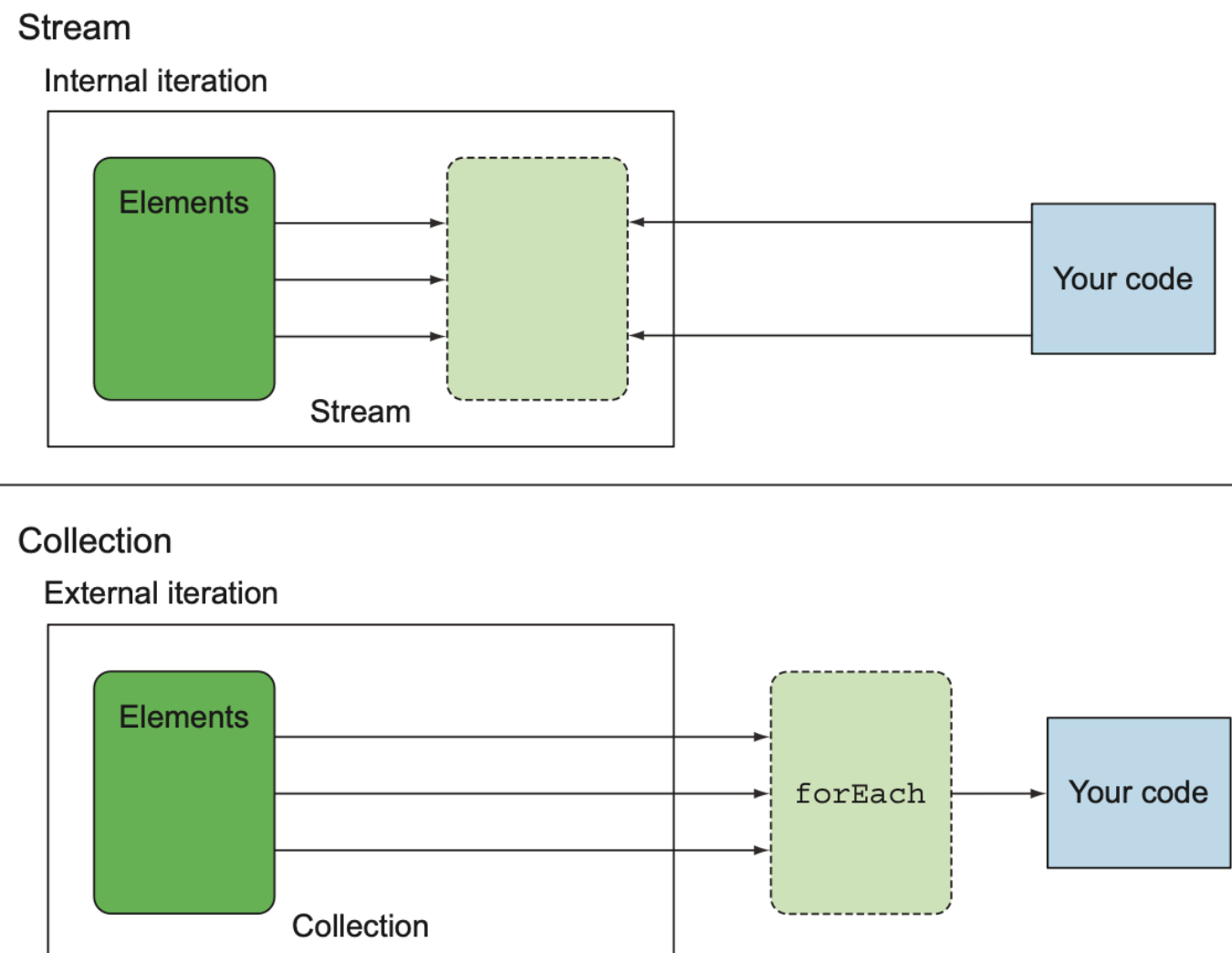
`Spilterator`, `BlockingQueue`, `Future`를 이용해서 자바8 에서 제공하지 않는 기능을 API로 만들 수 있습니다.

컬렉션 인터페이스를 사용하려면 사용자가 직접 요소를 반복해야 합니다(예를 들면 for-each). 이를 외부 반복(external iteration)이라고 합니다.

반면 스트림 라이브러리는 **내부 반복**(internal iteration, 반복을 알아서 처리하고 결과 스트림값을 어딘가에 저장해주는)을 사용합니다.

내부 반복을 이용하면 **작업을 투명하게 병렬로 처리하거나 더 최적화된 다양한 순서로 처리할 수** 있습니다. 스트림 라이브러리의 내부 반복은 데이터 표현과 하드웨어를 활용한 병렬성 구현을 자동으로 선택합니다.

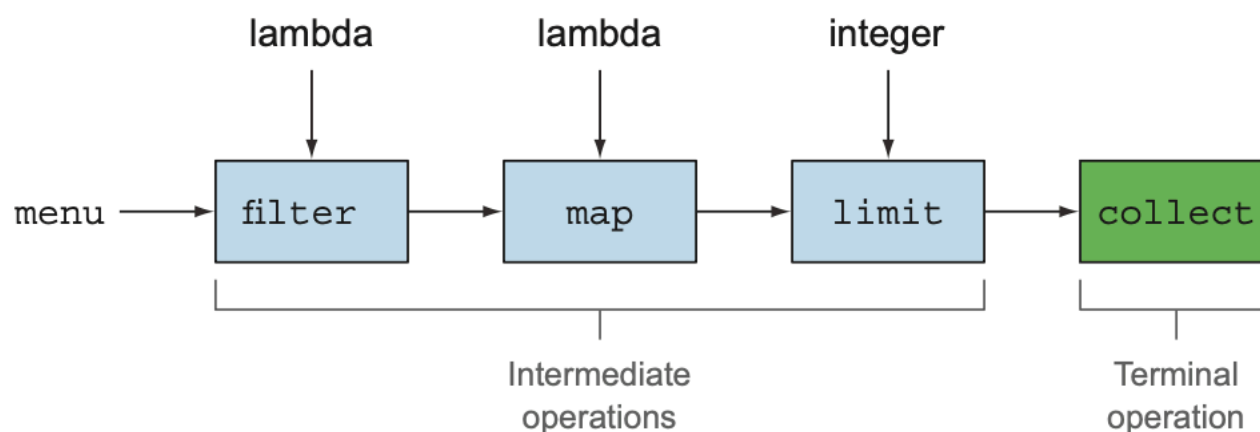
for-each 를 이용하는 외부 반복에서는 병렬성을 스스로 관리해야 합니다.



# 모던 자바 인 액션 4장

## 스트림 연산

```
List<String> threeHighCaloricDishNames = menu.stream()
    .filter(dish → dish.getCalories() > 300) // 중간 연산
    .map(Dish::getName) // 중간 연산
    .limit(maxSize: 3) // 중간 연산 intermediate operation
    .collect(Collectors.toList()); // 최종(종단) 연산 terminal operation
```



중간 연산의 중요한 특징은 단말(최종) 연산을 스트림 파이프라인에 실행하기 전까지는 아무 연산도 수행하지 않는다는 것, 즉 게으르다(lazy)는 것이다.

중간 연산을 합친 다음에 합쳐진 중간 연산을 최종 연산으로 한 번에 처리하기 때문이다.

Figure 4.5 Intermediate versus terminal operations

# 모던 자바 인 액션 4장

## 스트림 연산

```
public static void main(String[] args) { Complexity is 8 It's time to do something...
    List<String> names = menu.stream() Stream<Dish>
        .filter(dish → {
            System.out.println("filtering = " + dish.getName());
            return dish.getCalories() > 300;
        })
        .map(dish → {
            System.out.println("mapping:" + dish.getName());
            return dish.getName();
        }) Stream<String>
        .limit(maxSize: 3) //쇼트 서킷
        .collect(Collectors.toList());
    System.out.println(names);
}
```

limit 연산 그리고 쇼트서킷이라는  
기법 때문에 오직 처음 3개만 선택

filter와 map은 서로 다른 연산이지만  
한 과정으로 병합(루프 퓨전)

```
IntermediateOperation x
/Library/Java/JavaVirtualMachines/jdk-11.0.14.jdk/Contents/Home/bin/java ...
filtering = pork
mapping:pork
filtering = beef
mapping:beef
filtering = chicken
mapping:chicken
[pork, beef, chicken]
```

# 모던 자바 인 액션 4장

## 최종 연산

최종 연산은 스트림 파이프라인에서 결과를 도출합니다. 보통 List, Integer, void 등 스트림 이외의 결과가 반환됩니다.

**Table 4.1** Intermediate operations

Operation	Type	Return type	Argument of the operation	Function descriptor
filter	Intermediate	Stream<T>	Predicate<T>	T -> boolean
map	Intermediate	Stream<R>	Function<T, R>	T -> R
limit	Intermediate	Stream<T>		
sorted	Intermediate	Stream<T>	Comparator<T>	(T, T) -> int
distinct	Intermediate	Stream<T>		

**Table 4.2** Terminal operations

Operation	Type	Return type	Purpose
forEach	Terminal	void	Consumes each element from a stream and applies a lambda to each of them.
count	Terminal	long	Returns the number of elements in a stream.
collect	Terminal	(generic)	Reduces the stream to create a collection such as a List, a Map, or even an Integer. See chapter 6 for more detail.

# 출처

모던 자바 인 액션 3장, 4장

Real MySQL 8.0 1권, 5장 트랜잭션과 잠금

<https://madplay.github.io/post/effectively-final-in-java>

<https://hudi.blog/java-requirenonnull/>

**감사합니다**