Effective Java Item 14 - Comparable을 구현할지 고려하라

Comparable, Comparator 에 대한 간단 요약

모두 인터페이스입니다.

Comparable - compareTo(T o)

Comparator - compare(T o1, T o2)

차이점은,

Comparable은 **자기 자신과 매개변수 객체를 비교**하는 것이고,

Comparator는 **두 매개변수 객체를 비교**한다는 것입니다.

CompareTo는 단순 동치성 비교에 더해 순서까지 비교할 수 있으며, 제네릭합니다.

Comparable을 구현했다는 것은(Ex: String 클래스) 그 클래스의 인스턴스들에는 자연적인 순서(natural order)가 있음을 뜻합니다. 그래서 Comparable을 구현한 `객체들의 배열`은 다음처럼 손쉽게 정렬이 가능합니다.

Arrays.sort(a);

```
public static void main(String[] args) {
  String[] s = new String[]{"hello", "my", "name", "is", "euichan"};
  Arrays.sort(s);
  for (int i = 0 ; i < s.length; i++) {
    System.out.println(s[i]);
  }

}
```

다음 프로그램은 명령줄 인수들을 (중복을 제거하고) 알파벳순으로 출력합니다. String이 Comparable을 구현한 덕분입니다.

```
Set<String> set = new TreeSet<>();
Collections.addAll(set, args);
System.out.println(set);
```

사실상 자바 플랫폼 라이브러리의 모든 값 클래스와 열거 타입이 Comparable을 구현했습니다.
알파벳, 숫자, 연대 같이 순서가 명확한 값 클래스를 작성한다면 Comparable 인터페이스를 구현하는게 좋습니다.

```
public interface Comparable<T> {
        int compareTo(T t); // T -> int
}
```

compareTo 메서드의 일반 규약

이 객체와 주어진 객체의 순서를 비교합니다.

이 객체가 주어진 객체보다 작으면 `음의 정수를`
같으면 0을,
크면 `양의 정수를` 반환합니다.

이 객체와 비교할 수 없는 타입의 객체가 주어지면 `ClassCastException`을 던집니다.

4장 요약

스트림(Stream) 을 이용하면 선언형(즉, 데이터를 처리하는 임시 구현 코드 대신 **질의**로 표현할 수 있다)으로 컬렉션 데이터를 처리할 수 있습니다.

스트림을 이용하면 멀티스레드 코드를 구현하지 않아도 데이터를 **투명하게** 병렬로 처리할 수 있습니다.

Java7과 Java8 코드를 비교해서 차이점이 무엇인지 살펴봅니다.
`저칼로리의 요리명을 반환하고, 칼로리를 기준으로 요리를 정렬`

Java7 코드

```
 */
List<Dish> lowCaloricDishes = new ArrayList<>(); //'가비지 변수'
for (Dish dish : menu) { //filtering
  if (dish.getCalories() < 400) {
    lowCaloricDishes.add(dish);
  }
}
Collections.sort(lowCaloricDishes, new Comparator<Dish>() { //익명 클래스로 요리 정렬
  @Override
  public int compare(Dish dish1, Dish dish2) {
    return Integer.compare(dish1.getCalories(), dish2.getCalories());
  }
});
List<String> lowCaloricDishesName = new ArrayList<>();
for (Dish dish : lowCaloricDishes) {
  lowCaloricDishesName.add(dish.getName());
}
```

위 코드에서는 `lowCaloricDishes` 라는 가비지 변수를 사용했습니다. 즉, lowCaloricDishes는 컨테이너 역할만 하는 중간 변수입니다. Java8에서 이러한 세부 구현은 라이브러리 내에서 모두 처리합니다.

Java8 코드

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

stream()을 parallelStream()으로 바꾸면 이 코드를 멀티코어 아키텍처에서 병렬로 실행할 수 있습니다.

스트림을 사용할 때의 이점

- **선언형**으로 코드를 구현할 수 있습니다. 즉, 루프와 if 조건문 등의 제어 블록을 사용해서 어떻게 동작을 구현할지 지정할 필요 없이 '저칼로리의 요리만 선택하라' 같은 동작의 수행을 지정할 수 있습니다.
- filter, sorted, map, collect 같은 여러 빌딩 블록 연산을 연결해서 복잡한 데이터 처리 파이프라인을 만들 수 있습니다. filter 메서드의 결과는 sorted 메서드로, sorted 결과는 map 메서드로, map 메서드의 결과는 collect로 연결됩니다.

filter, sorted, map, collect 같은 연산은 **고수준 빌딩 블록**으로 이루어져 있으므로 특정 스레딩 모델에 제한되지 않고 자유롭게 어떤 상황에서든 사용할 수 있습니다. (또한 이들은 내부적으로 단일 스레드 모델에 사용할 수 있지만 멀티코어 아키텍처를 최대한 투명하게 활용할 수 있게 구현되어 있습니다) 결과적으로 우리는 데이터 처리 과정을 병렬화하면서 스레드와 락을 걱정할 필요가 없습니다.

Java8 Stream API의 특징을 다음과 같이 요약할 수 있습니다.

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

Java8 컬렉션에는 스트림을 반환하는 stream 메서드가 추가됐습니다.

```
@Contract(pure = true)
default Stream<E> stream() {
    return StreamSupport.stream(spliterator(), 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 `spliterator()` method cannot return a spliterator that is `IMMUTABLE`, `CONCURRENT`, or *late-binding*. (See `spliterator()` for details.)

Returns:	a possibly parallel `Stream` over the elements in this collection
Implementation Requirements:	The default implementation creates a parallel `Stream` from the collection's `Spliterator`.
Since:	1.8

16 usages 9 overrides

```
@Contract(pure = true)
default Stream<E> parallelStream() {
    return StreamSupport.stream(spliterator(), parallel: true);
}
```

스트림이란?

데이터 처리 연산을 지원하도록 소스에서 추출된 연속된 요소 로 정의할 수 있습니다.

- 연속된 요소 : 컬렉션과 마찬가지로 스트림은 특정 요소 형식으로 이루어진 연속된 값 집합의 인터페이스를 제공합니다. 컬렉션의 주제는 **데이터** 이고 스트림의 주제는 **계산** 입니다.
- 소스 : 스트림은 컬렉션, 배열, I/O 자원 등의 데이터 제공 소스로부터 데이터를 `소비` 합니다. 정렬된 컬렉션으로 스트림을 생성하면 정렬이 그대로 유지됩니다. 즉, 리스트로 스트림을 만들면 스트림의 요소는 리스트

의 요소와 같은 순서를 유지합니다.

- 데이터 처리 연산 : 스트림은 함수형 프로그래밍 언어에서 일반적으로 지원하는 연산과 데이터베이스와 비슷한 연산을 지원합니다. `filter, map, reduce, find, match, sort` 등으로 데이터를 조작할 수 있습니다. 스트림 연산은 순차적으로 또는 병렬로 실행할 수 있습니다.

또한 스트림에는 다음과 같은 두 가지 중요 특징이 있습니다.

- 파이프라이닝 : 대부분의 스트림 연산은 스트림 연산끼리 연결해서 커다란 파이프라인을 구성할 수 있도록 스트림 자신을 반환합니다. 그 덕분에 **게으름(laziness)**, **쇼트서킷** 같은 최적화도 얻을 수 있습니다. 연산 파이프라인은 데이터 소스에 적용하는 데이터베이스 질의와 비슷합니다.
- 내부 반복 : 반복자를 이용해서 명시적으로 반복하는 컬렉션과 달리 스트림은 내부 반복을 지원합니다.

```
public class HighCaloriesNames {

  public static void main(String[] args) {
   List<String> threeHighCaloricDishNames =
       menu.stream() //메뉴(요리 리스트)에서 스트림을 얻는다.
           .filter(dish -> dish.getCalories() > 300) //파이프라인 연산 만들기. 첫 번째로 고칼로리 요리 필터링
           .map(Dish::getName) //요리명 추출
           .limit( maxSize: 3)//선착순 세 개만 선택
           .collect(toList()); //결과를 다른 리스트로 저장
    System.out.println(threeHighCaloricDishNames); // [pork, beef, chicken]
  }

}
```

우선 요리 리스트를 포함하는 menu에 stream 메서드를 호출해서 스트림을 얻었습니다.
`Stream<Dish>`

여기서 **데이터 소스**는 요리 리스트(메뉴) 입니다.
데이터 소스는 **연속된 요소**를 스트림에 제공합니다. 다음으로 스트림에 filter, map, limit, collect로 이어지는 일련의 데이터 처리 연산을 적용합니다.

collect를 제외한 모든 연산은 서로 **파이프라인**을 형성할 수 있도록 스트림을 반환합니다.
파이프라인은 소스에 적용하는 질의 같은 존재입니다.

마지막으로 collect 연산으로 **파이프라인을 처리** 해서 결과를 반환합니다. (collect는 스트림이 아니라 List를 반환합니다)

마지막으로 collect를 호출하기 전까지는 menu에서 무엇도 선택되지 않으며 출력 결과도 없습니다.
즉, collect가 호출되기 전까지 메서드 호출이 저장되는 효과가 있습니다.

- filter : **람다를 인수로 받아** 스트림에서 특정 요소를 제외시킵니다.
- map : **람다를 이용해서** 한 요소를 다른 요소로 변환하거나 정보를 추출합니다.
- limit : 정해진 개수 이상의 요소가 스트림에 저장되지 못하게 스트림 크기를 축소합니다
- collect : 스트림을 다른 형식으로 변환합니다.
  Stream API 를 이용해 좀 더 선언형으로 데이터를 처리할 수 있었습니다.
  ('고칼로리 요리 3개를 찾아라'와 같이)

컬렉션과 스트림의 차이 = 데이터를 언제 계산하느냐

컬렉션과 스트림 모두 **연속된** 요소 형식의 값을 저장하는 자료구조의 인터페이스를 제공합니다.
연속된 = 순서와 상관없이 아무 값에나 접속하는 것이 아니라 순차적으로 값에 접근합니다.

컬렉션과 스트림의 가장 큰 차이는 데이터를 언제 계산하느냐 입니다.

컬렉션은 현재 자료구조가 포함하는 **모든** 값을 메모리에 저장하는 자료구조입니다.
즉, 컬렉션의 모든 요소는 컬렉션에 추가하기 전에 계산되어야 합니다.
(컬렉션에 요소를 추가하거나 컬렉션의 요소를 삭제할 수 있습니다. 이런 연산을 수행할 때마다 컬렉션의 모든 요소를 메모리에 저장해야 하며 컬렉션에 추가하려는 요소는 **미리 계산되어야 합니다.**)

반면 스트림은 이론적으로 **요청할 때만 요소를 계산**하는 **고정된 자료구조**입니다.

스트림에 요소를 추가하거나 스트림에서 요소를 제거할 수 없습니다.

결과적으로 스트림은 게으르게 만들어지는 컬렉션과 같습니다. 즉, 사용자가 데이터를 요청할 때만 값을 계산합니다.

반면, 컬렉션은 적극적으로 생성합니다.

적극적 생성 = 모든 값을 계산할 때까지 기다립니다(**컬렉션**)
게으른 생성 = 필요할 때만 값을 계산합니다(**스트림**)

딱 한번만 탐색할 수 있다

반복자와 마찬가지로 스트림도 한 번만 탐색할 수 있습니다. 즉, 탐색된 스트림의 요소는 소비됩니다.한 번 탐색한 요소를 다시 탐색하려면 초기 데이터 소스에서 새로운 스트림을 만들어야 합니다.

스트림은 단 한 번만 소비할 수 있다는 점을 명심하자!

컬렉션과 스트림의 또 다른 차이점은 외부 반복과 내부 반복입니다.

외부 반복과 내부 반복

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

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

```
public static void main(String[] args) {
        List<String> names = new ArrayList<>();
                for (Dish dish : menu) {
                names.add(dish.getName());
        }
}
```

for-each 구문은 반복자를 사용하는 불편함을 어느 정도 해결해줍니다. for-each를 이용하면 Iterator 객체를 이용하는 것보다 더 쉽게 컬렉션을 반복할 수 있습니다.

```
List<String> names = new ArrayList<>();
for (Dish dish : menu) {
        names.add(dish.getName());
}
```

Iterator 객체를 이용하는 것보다 더 쉽게 컬렉션을 반복할 수 있습니다.

```
List<String> names = new ArrayList<>();
Iterator<Dish> iterator = menu.iterator();
while (iterator.hasNext()) {
        Dish dish = iterator.next();
        names.add(dish.getName());
}
```

다음은 스트림을 이용한 내부 반복입니다.

```
List<String> names = menu.stream()
.map(Dish::getName)//map 메서드를 getName 메서드로 파라미터화해서 요리명을 추출합니다.
.collect(Collectors.toList());
```

컬렉션은 외부적으로 반복, 즉 **명시적으로 컬렉션 항목을 하나씩 가져와서 소비**합니다.

내부 반복을 이용하면 작업을 투명하게 병렬로 처리하거나 더 최적화된 다양한 순서로 처리할 수 있습니다.

스트림 라이브러리의 내부 반복은 `데이터 표현`과 `하드웨어를 활용한 병렬성 구현`을 자동으로 선택합니다.

반면 for-each를 이용하는 외부 반복에서는 병렬성을 스스로 관리해야 합니다.(병렬성을 포기 or Synchronized 이용)

스트림은 내부 반복을 사용하므로 반복 과정을 우리가 신경 쓰지 않아도 됩니다.

하지만 이와 같은 이점을 누리려면 (filter나 map 같이) 반복을 숨겨주는 연산 리스트가 미리 정의되어 있어야 합니다.

반복을 숨겨주는 대부분의 연산은 람다 표현식을 인수로 받으므로 3장에서 배운 동작 파라미터화를 활용할 수 있습니다.

스트림 연산

스트림 인터페이스의 연산을 크게 두 가지로 구분할 수 있습니다.

```
List<String> threeHighCaloricDishNames =
    menu.stream() //메뉴(요리 리스트)에서 스트림을 얻는다.
        .filter(dish -> dish.getCalories() > 300) //파이프라인 연산 만들기. 첫 번째로 고칼로리 요리 필터링
        .map(Dish::getName) //요리명 추출
        .limit( maxSize: 3)//선착순 세 개만 선택
        .collect(toList()); //결과를 다른 리스트로 저장
```

filter, map, limit은 서로 연결되어 파이프라인을 형성합니다.
collect로 파이프라인을 실행한 다음에 **닫습니다.**

연결할 수 있는 스트림 연산을 **중간 연산(intermediate operation)** 이라고 하며, 스트림을 닫는 연산을 **최종 연산(external operation)** 이라고 합니다.

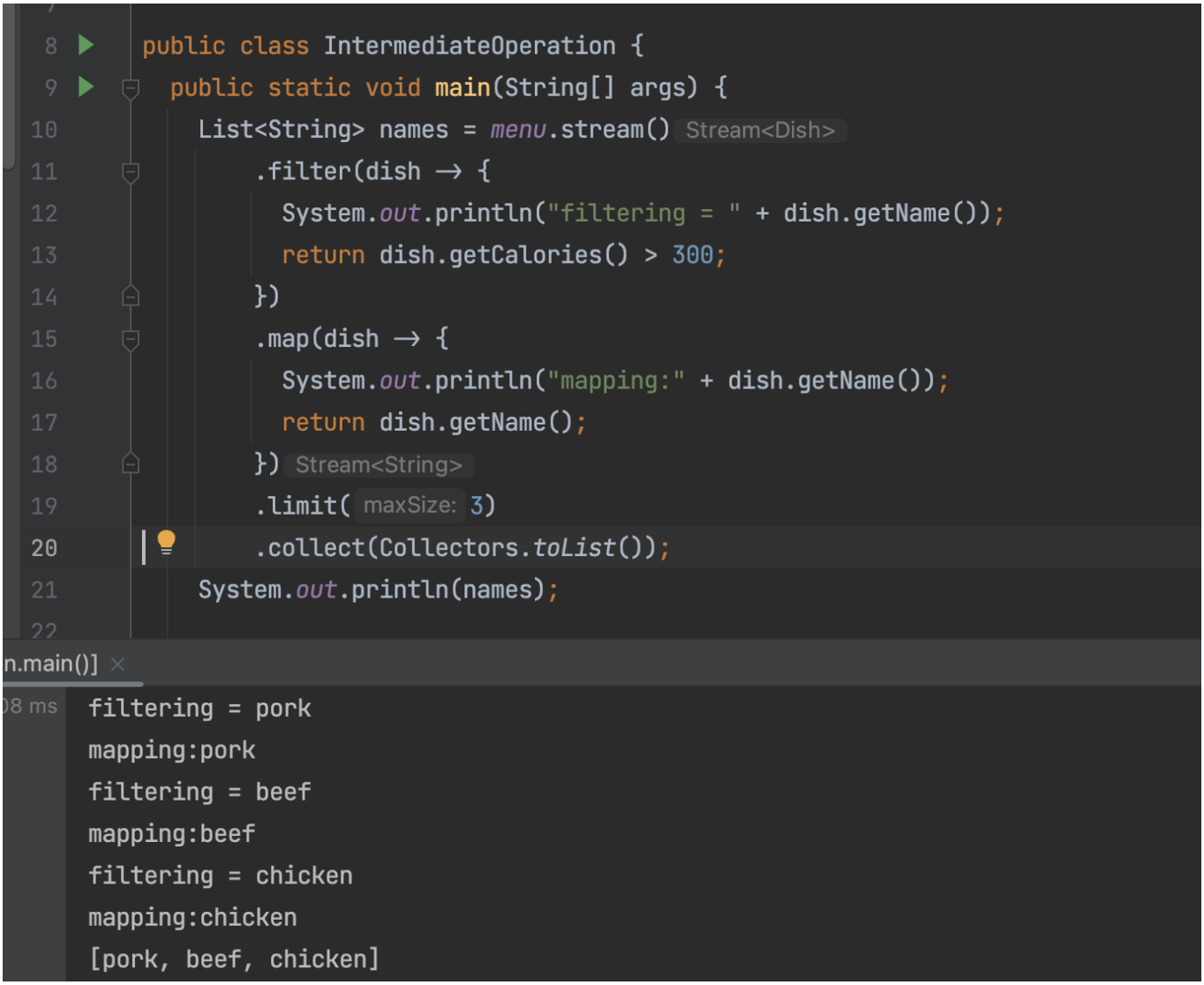
중간 연산

`filter`나 `sorted` 같은 중간 연산은 다른 스트림을 반환합니다. 따라서 여러 중간 연산을 연결해서 질의를 만들 수 있습니다.

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

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

스트림 파이프라인에서 어떤 일이 일어나는지 쉽게 확인할 수 있도록 람다가 현재 처리 중인 요리를 출력해봅니다.



```
public class IntermediateOperation {
    public static void main(String[] args) {
        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)
                .collect(Collectors.toList());
        System.out.println(names);
```

```
n.main()]
08 ms  filtering = pork
       mapping:pork
       filtering = beef
       mapping:beef
       filtering = chicken
       mapping:chicken
       [pork, beef, chicken]
```

300칼로리가 넘는 요리는 여러 개지만 오직 처음 3개만 선택되었습니다. 이는 limit 연산 그리고 **쇼트서킷**이라 불리는 기법 덕분입니다.

둘째, filter와 map은 서로 다른 연산이지만 한 과정으로 병합되었습니다. (이 기법을 **루프 퓨전** 이라고 합니다.)

최종 연산

최종 연산은 스트림 파이프라인에서 결과를 도출합니다.

보통 최종 연산에 의해 List, Integer, void 등 스트림 이외의 결과가 반환됩니다. 예를 들어 다음 파이프라인에서 forEach는 소스에 각 요리에

람다를 적용한 다음에 void를 반환하는 최종 연산입니다.

```
menu.stream().forEach(System.out::println);
```

System.out.println을 forEach에 넘겨주면 menu에서 만든 스트림의 모든 요리를 출력합니다.

스트림 이용하기

스트림 이용 과정은 다음과 같이 세 가지로 요약할 수 있습니다.

- 질의를 수행할 (컬렉션 같은) 데이터 소스
- 스트림 **파이프라인을 구성할 중간 연산 연결**
- 스트림 파이프라인을 실행하고 결과를 만들 **최종 연산**

4장 추가 내용

1. Stream의 forEach는 loop가 아니다.

https://www.popit.kr/java8-stream%EC%9D%80-loop%EA%B0%80-%EC%95%84%EB%8B%88%EB%8B%A4/

2. Stream API를 사용하며 실수하기 쉬운 것들

https://hamait.tistory.com/547

3. Lazy Evaluation

https://dororongju.tistory.com/137

4. 컬렉션에서 원소 삭제하기

https://www.daleseo.com/how-to-remove-from-list-in-java/

5장 요약

데이터를 어떻게 처리할지는 스트림 API가 관리하므로 편리하게 데이터 관련 작업을 할 수 있습니다.
스트림 API는 내부 반복 뿐 아니라 **코드를 병렬로 실행할지 여부도 결정할 수 있습니다.**

이러한 일은 순차적인 반복을 단일 스레드로 구현하는 외부 반복으로는 달성할 수 없습니다.

필터링 : 프레디케이트 필터링

```
List<Dish> vegetarianMenu = menu.stream()

.filter(Dish::isVegetarian)//채식 요리인지 확인하는 method reference
                                                    .collect(toList());
```

filter 메서드는 프레디케이트(불리언을 반환하는 함수)를 인수로 받아서 프레디케이트와 일치하는 모든 요소를 포함하는 스트림을 반환합니다.

필터링 : 고유 요소 필터링

스트림은 고유 요소로 이루어진 스트림을 반환하는 **distinct** 메서드도 지원합니다.
(고유 여부는 hashCode, equals)

다음 코드는 리스트의 모든 짝수를 선택하고 중복을 필터링합니다.

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

스트림 슬라이싱(Java 9부터)

5.2절에서는 스트림의 요소를 선택하거나 스킵하는 다양한 방법을 설명합니다.

프레디케이트를 이용하는 방법, 스트림의 처음 몇 개의 요소를 무시하는 방법, 특정 크기로 스트림을 줄이는 방법 등 다양한 방법을 이용할 수 있습니다.

프레디케이트를 이용한 슬라이싱

Java9는 스트림의 요소를 효과적으로 선택할 수 있도록 takeWhile, dropWhile 두 가지 새로운 메서드를 지원합니다.

320 칼로리 이하의 요리를 선택하는 방법은 다음과 같이 filter를 이용하는 방법을 생각해볼 수 있습니다.

```
List<Dish> filteredMenu = specialMenu.stream()
        .filter(dish -> dish.getCalories() < 320)
        .collect(Collectors.toList());
```

위 리스트는 이미 칼로리 순으로 정렬되어 있습니다.filter 연산을 이용하면 전체 스트림을 반복하면서 각 요소에 프레디케이트를 적용하게 됩니다.

따라서 리스트가 이미 정렬되어 있다는 사실을 이용해 320칼로리보다 크거나 같은 요리가 나왔을 때 (칼로리 >= 320) 반복 작업을 중단할 수 있습니다.

작은 리스트에는 이와 같은 동작이 별거 아닌 것처럼 보일 수 있지만, 아주 많은 요소를 포함하는 큰 스트림에서는 상당한 차이가 될 수 있습니다.

`takeWhile` 연산을 이용하면 이를 간단하게 처리할 수 있습니다.

`takeWhile`을 이용하면 무한스트림을 포함한 모든 스트림에 프레디케이트를 적용해 스트림을 슬라이스할 수 있습니다.

```
List<Dish> slicedMenu1 = specialMenu.stream()
        .takeWhile(dish -> dish.getCalories() < 320)
        .collect(Collectors.toList());
```

`filter`는 조건에 대해 다 검사하며 참인것만 다음으로 넘어가지만 `takeWhile`은 조건에 대해 참이 아닐경우

바로 거기서 멈추게 됩니다.

나머지 요소를 선택하려면 `dropWhile`을 이용하면 됩니다.

```
List<Dish> slicedMenu2 = specialMenu.stream()
        .dropWhile(dish -> dish.getCalories() < 320)
        .collect(Collectors.toList());
```

`dropWhile`은 `takeWhile`과 정반대의 작업을 수행합니다. dropWhile은 프레디케이트가 처음으로 거짓이 되는 지점까지 발견된 요소를 버립니다.

프레디케이트가 거짓이 되면 그 지점에서 작업을 중단하고 남은 모든 요소를 반환합니다.

dropWhile은 무한한 남은 요소를 가진 무한 스트림에서도 동작합니다.

스트림 축소

스트림은 주어진 값 이하의 크기를 갖는 새로운 스트림을 반환하는 `limit(n)` 메서드를 지원합니다.

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

예를 들어 300칼로리 이상의 세 요리를 선택해서 리스트를 만들 수 있습니다.

filter와 limit을 조합한 모습으로, 프레디케이트와 일치하는 처음 세 요소를 선택한 다음에 즉시 결과를 반환합니다.

정렬되지 않은 스트림(예를 들면 소스가 Set)에도 limit을 사용할 수 있습니다. 소스가 정렬되어 있지 않았다면 limit의 결과도 정렬되지 않은 상태로 반환된다.

요소 건너뛰기

스트림은 처음 n개 요소를 제외한 스트림을 반환하는 skip(n) 메서드를 지원합니다. n개 이하의 요소를 포함하는 스트림에 skip(n)을 호출하면 빈 스트림이 반환됩니다.

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

예를 들어 다음 코드는 300칼로리 이상의 처음 두 요리를 건너뛴 다음에 300칼로리가 넘는 나머지 요리를 반환합니다.

매핑

특정 객체에서 특정 데이터를 선택하는 작업은 데이터 처리 과정에서 자주 수행되는 연산입니다.

예를 들어 SQL의 테이블에서 특정 열만 선택할 수 있습니다.

스트림 API의 map과 flatMap 메서드는 `특정 데이터를 선택하는 기능`을 제공합니다.

스트림의 각 요소에 함수 적용하기

스트림은 함수를 인수로 받는 map 메서드를 지원합니다. 인수로 제공된 함수는 각 요소에 적용되며 함수를 적용한 결과가 새로운 요소로 매핑됩니다.

이 과정은 기존의 값을 고친다라는 개념보다는 새로운 버전을 만든다 라는 개념에 가까우므로 변환(transforming)에 가까운 매핑(mapping)이라는 단어를 사용합니다.

```
List<String> dishNames = menu.stream()
        .map(Dish::getName)//인수로 제공된 함수는 각 요소에 적용
        .collect(Collectors.toList());
```

getName은 문자열을 반환하므로 map 메서드의 출력 스트림은 `Stream<String>` 형식을 가집니다.

단어 리스트가 주어졌을 때 각 단어가 포함하는 글자 수의 리스트를 반환한다고 가정합니다.

리스트의 각 요소에 함수를 적용하면 가능합니다. 이전 예제에서 확인했던 것처럼 map을 이용할 수 있습니다.

`String::length`를 map에 전달해서 문제를 해결할 수 있습니다.

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

각 요리명의 길이를 알고 싶다면 다른 map 메서드를 연결할 수 있습니다.

```
List<Integer> dishNameLengths = menu.stream()
        .map(Dish::getName)
        .map(String::length)
        .collect(Collectors.toList());
```

스트림 평면화

리스트에서 **고유 문자**로 이루어진 리스트를 반환해보겠습니다.

```
stringList.stream()
        .map(word -> word.split(""))
        .distinct()
        .collect(Collectors.toList());
```

위 코드에서 map으로 전달한 람다는 각 단어의 String[](문자열 배열)을 반환한다는 점이 문제입니다.

따라서 map 메서드가 반환한 스트림의 형식은 `Stream<String[]>`입니다. 원하는 것은 `Stream<String>`입니다.

다행히 flatMap이라는 메서드를 이용해서 이 문제를 해결할 수 있습니다.

map과 Arrays.stream 활용

우선 배열 스트림 대신 문자열 스트림이 필요합니다. 다음 코드에서 보여주는 것처럼 문자열을 받아 스트림을 만드는 `Arrays.stream()` 메서드가 있습니다.

```
String[] arrayOfWords = {"Goodbye", "World"};
Stream<String> streamOfWords = Arrays.stream(arrayOfWords);
```

위 예제의 파이프라인에 Arrays.stream() 메서드를 적용해봅시다.

```
List<Stream<String>> collect = words.stream()
        .map(word -> word.split(""))
        .map(Arrays::stream)
        .distinct()
        .collect(Collectors.toList());
```

결국 스트림 리스트 `List<Stream<String>>`이 만들어지면서 문제가 해결되지 않았습니다. 문제를 해결하려면

먼저 각 단어를 개별 문자열로 이루어진 배열로 만든 다음에 각 배열을 별도의 스트림으로 만들어야 합니다.

flatMap 사용

```
List<String> flatMap = words.stream()
        .map(word -> word.split(""))
        .flatMap(Arrays::stream)
        .distinct()
        .collect(Collectors.toList());
```

flatMap은 각 배열을 스트림이 아니라 스트림의 콘텐츠로 매핑합니다. 즉, map(Arrays::stream)과 달리 flatMap은 하나의 평면화된 스트림을 반환합니다.

요약하자면, flatMap 메서드는 각 값을 다른 스트림으로 만든 다음에 모든 스트림을 하나의 스트림으로 연결하는 기능을 수행합니다.

flatMap 메서드는 스트림의 형태가 배열과 같을 때, 모든 원소를 단일 원소 스트림으로 반환할 수 있습니다.

스트림의 형태가 배열인 경우 또는 입력된 값을 또 다시 스트림의 형태로 반환하고자 할 때는 flatMap이 유용합니다.

5장 추가 내용 : map vs flatMap

https://madplay.github.io/post/difference-between-map-and-flatmap-methods-in-java

검색과 매핑

`특정 속성이 데이터 집합에 있는지 여부를 검색`하는 데이터 처리도 자주 사용됩니다.

스트림 API는 **allMatch, anyMatch, noneMatch, findFirst, findAny**등 다양한 유틸리티 메서드를 제공합니다.

프레디케이트가 적어도 한 요소와 일치하는지 확인

프레디케이트가 주어진 스트림에서 적어도 한 요소와 일치하는지 확인할 때 anyMatch 메서드를 이용합니다.

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

anyMatch는 불리언을 반환하므로 최종 연산입니다.

프레디케이트가 모든 요소와 일치하는지 검사

allMatch 메서드는 anyMatch와는 달리 스트림의 모든 요소가 주어진 프레디케이트와 일치하는 지 검사합니다.

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

NONEMATCH

noneMatch는 allMatch와 반대 연산을 수행합니다. 즉, nonMatch는 주어진 프레디케이트와 일치하는 요소가 없는지 확인합니다.

예를 들어 이전 예제를 다음처럼 noneMatch로 다시 구현할 수 있습니다.

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

anyMatch, allMatch, noneMatch 세 머서드는 스트림 쇼트서킷 기법(Java의 &&, ||)와 같은 연산을 활용합니다.

마찬가지로 스트림의 모든 요소를 처리할 필요 없이 주어진 크기의 스트림을 생성하는 limit도 쇼트서킷 연산입니다.

요소 검색

findAny 메서드는 현재 스트림에서 **임의의 요소**를 반환합니다. findAny 메서드를 다른 스트림연산과 연결해서 사용할 수 있습니다.

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

스트림 파이프라인은 내부적으로 단일 과정으로 실행할 수 있도록 최적화됩니다.

즉, 쇼트서킷을 이용해서 결과를 찾는 즉시 실행을 종료합니다.

Optional이란?

`java.util.Optional`. `Optional<T>` 클래스는 값의 존재나 부재 여부를 표현하는 컨테이너 클래스입니다.

Optional은 값이 존재하는지 확인하고 값이 없을 때 어떻게 처리할지 강제하는 기능을 제공합니다.

- isPresent()는 Optional이 값을 포함하면 참(true)을 반환하고, 값을 포함하지 않으면 거짓(false)을 반환합니다.
- `ifPresent(Consumer<T> block)` 은 값이 있으면 주어진 블록을 실행합니다. Consumer : T -> void
- T get()은 값이 존재하면 값을 반환하고, 값이 없으면 NoSuchElementException을 일으킵니다.
- T orElse(T other)는 값이 있으면 값을 반환하고, 값이 없으면 기본값을 반환합니다.

첫 번째 요소 찾기

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

System.out.println(firstSquareDivisibleByThree.orElseThrow(NoSuchElementExce
ption::new));
```

병렬성 때문에 findFirst와 findAny 모두 필요하다. 병렬 실행에서는 첫 번째 요소를 찾기 어려우므로 반환 순서가 상관없다면

병렬 스트림에서는 제약이 적은 findAny를 사용한다.

리듀싱

스트림 요소를 조합해서 더 복잡한 질의를 표현하는 방법을 설명합니다.

예를 들어, 메뉴의 모든 칼로리의 합계를 구하시오, 메뉴에서 칼로리가 가장 높은 요리는? 같은 질의.

이러한 질의를 수행하려면 Integer 같은 결과가 나올 때까지 스트림의 모든 요소를 반복적으로 처리해야 합니다.

이런 질의를 **리듀싱 연산(모든 스트림 요소를 처리해서 값으로 도출하는)** 이라고 합니다.

요소의 합

for-each 루프를 사용해서 리스트의 숫자 요소를 더하는 코드는 다음과 같습니다.

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

reduce를 이용하면 다음처럼 스트림의 모든 요소를 더할 수 있습니다.

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

reduce로 다른 람다를 넘겨주면 모든 요소에 곱셈을 적용할 수 있습니다.

```
int product = numbers.stream()
        .reduce(1, (a, b) -> a * b);
```

메서드 참조를 이용하면 이 코드를 좀 더 간결하게 만들 수 있습니다.

```
int sum = numbers.stream()
        .reduce(0, Integer::sum);
```

초깃값 없음

초깃값을 받지 않도록 오버로드된 reduce도 있습니다. 그러나 이 reduce는 Optional 객체를 반환합니다.

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

스트림에 아무 요소도 없는 상황이라면, 초기값이 없으므로 reduce는 합계를 반환할 수 없다. 따라서 합계가 없음을 가리킬 수 있도록

Optional 객체로 감싼 결과를 반환합니다.

최댓값과 최솟값

```
//최댓값
Optional<Integer> max = numbers.stream()
        .reduce(Integer::max);
//최솟값
Optional<Integer> min = numbers.stream()
        .reduce(Integer::min);
```