

## 5. 스트림 활용 - 스트림과 친구 되어보기

### 5. 스트림 활용

💡 Java 8의 Stream API를 활용한 데이터 처리 기법을 상세히 알아본다.

#### 5.1 필터링: Stream에서 특정 조건을 만족하는 요소만 선택해 새로운 스트림 생성

필터링은 스트림에서 원하는 요소만 선택하는 핵심 기능이다. 두 가지 주요 방식이 있다:

- 프레디케이트 필터링
- 고유 요소 필터링

##### 5.1.1 프레디케이트로 필터링: 논리 조건 기반의 식별

스트림 인터페이스는 `filter` 메소드를 지원한다. `filter()` 는 프레디케이트(불리언을 반환하는 함수)를 인수로 받아 프레디케이트와 일치하는 모든 요소를 포함하는 스트림을 반환한다.

예제 1: 채식 메뉴 필터링

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

예제 2: 특정 문자로 시작하는 이름 필터링

```
List<String> filtered = names.stream()
    .filter(name -> name.startsWith("K"))
    .collect(toList());
```

##### 5.1.2 고유 요소 필터링: 중복 제거해 유일한 요소 반환

`distinct()` 는 중복 요소를 제거하고 고유한 요소를 포함하는 스트림을 반환한다. 고유 여부는 스트림에서 만든 객체의 `hashCode` 와 `equals` 로 결정된다.

예제: 짝수 중복 제거

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 1, 1, 4, 5, 2, 4);
numbers.stream()
    .filter(i -> i % 2 == 0)
    .distinct()
    .forEach(System.out::println); // 2, 4 출력
```

## 5.2 스트림 슬라이싱: Java 9의 새로운 기능

✦ 요소를 선택하거나 스킵하는 다양한 방법이 존재한다. 프레디케이트 이용, 처음 몇 개 요소 무시, 특정 크기로 스트림 줄이기 등이 가능하다.

### 5.2.1 프레디케이트를 이용한 슬라이싱

#### 1) takeWhile: 조건이 true인 동안 요소를 가져온다

정렬된 리스트에서 특정 조건을 만족하는 요소만 효율적으로 선택할 수 있다.

```
List<Dish> specialMenu = Arrays.asList(
    new Dish("seasonal fruit", true, 120, Dish.Type.OTHER),
    new Dish("prawns", false, 300, Dish.Type.FISH),
    new Dish("rice", true, 350, Dish.Type.OTHER),
    new Dish("chicken", false, 400, Dish.Type.MEAT)
);

// 320 칼로리 이하 요리 선택 (정렬된 리스트 가정)
List<Dish> slicedMenu1 = specialMenu.stream()
    .takeWhile(dish -> dish.getCalories() < 320)
    .collect(toList());
```

💡 **filter vs takeWhile**: filter는 전체 스트림을 반복하지만, takeWhile은 조건이 false가 되면 즉시 중단한다.

#### 2) dropWhile: 조건이 true인 동안 건너뛰고 이후의 요소를 가져온다

dropWhile 은 takeWhile 과 정반대 작업을 수행한다. 처음으로 거짓이 되는 지점까지 발견한 요소를 버린다. 프레디케이트가 거짓이 되면 그 지점에서 작업을 중단하고 남은 요소를 반환한다.

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

### 5.2.2 스트림 축소: 앞에서 n개만 선택

스트림이 정렬되어 있으면 최대 요소 n개를 반환할 수 있다.

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

### 5.2.3 요소 건너뛰기

처음 n개 요소를 제외한 스트림을 반환하는 skip(n) 을 지원한다.


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

## 5.3 매핑

특정 객체에서 특정 데이터를 선택하는 작업은 자주 수행되는 연산이다. Stream API의 `map` 과 `flatMap` 은 특정 데이터를 선택하는 기능을 제공한다.

### 5.3.1 스트림의 각 요소에 함수 적용

함수를 인수로 받는 `map` 메소드를 지원한다. 인수로 제공된 함수는 각 요소에 적용되며 함수를 적용한 결과가 새로운 요소로 매핑된다.

 "새로운 버전을 만든다"라는 개념에 더 가깝다.

예제: 요리명 추출

```
List<String> dishNames = menu.stream()
    .map(Dish::getName)
    .collect(toList());
```

`getName` 은 문자열을 반환하므로 `map` 메소드의 출력 스트림은 `Stream<String>` 형식을 갖는다.

### 5.3.2 map vs flatMap

**map: 요소 하나를 다른 하나로 변환**

```
List<String> words = List.of("Java", "Stream");
List<String[]> result = words.stream()
    .map(word -> word.split("")) // String -> String[]
    .collect(Collectors.toList());

// 결과: [["J", "a", "v", "a"], ["S", "t", "r", "e", "a", "m"]]
// 중첩 구조가 생성됨!
```

중첩 구조가 생성되면 중복 제거가 어렵고 한 글자씩 작업이 어렵다. 이는 `flatMap` 메소드를 이용해 문제를 해결할 수 있다.

**Arrays.stream: 배열을 스트림으로 변환**

```
String[] letters = {"a", "b", "c"};
Stream<String> stream = Arrays.stream(letters);
```

배열을 `Stream<T>` 로 바꿔줘야 `flatMap`에서 합치기가 가능하다.

## flatMap: 납작하게 펴기

map() 사용 시 결과가 Stream<String[]> 처럼 중첩 구조를 생성한다. 전체에서 글자 단위로 한 줄의 스트림으로 작업하고 싶을 때가 있다.

**flatMap**: 내부에 있는 스트림을 납작하게 평탄화시켜준다.

```
List<String> words = List.of("Java", "Stream");
List<String> result = words.stream()
    .flatMap(word -> Arrays.stream(word.split("")))
    .distinct()
    .collect(toList());

// 결과: ["J", "a", "v", "S", "t", "r", "e", "m"]
// Stream<String>으로 펼쳐짐!
```

## 5.4 검색과 매칭

### 5.4.1 조건에 하나라도 일치: anyMatch

anyMatch 메소드를 사용하면 프레디케이트가 적어도 한 요소와 일치하는지 확인할 수 있다. 불리언을 반환한다.

```
if(menu.stream().anyMatch(Dish::isVegetarian)) {
    System.out.println("채식 메뉴가 있습니다!");
}
```

### 5.4.2 모든 요소가 조건을 만족: allMatch

allMatch 는 모든 요소가 주어진 프레디케이트와 일치하는지 검사한다.

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

### 5.4.3 모든 요소가 조건을 만족하지 않는 경우: noneMatch

noneMatch 는 allMatch 와 반대 연산을 수행한다. 일치하는 요소가 없는지 확인한다.

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

#### ⚡ 쇼트서킷(Short-circuit)

- Stream이나 조건문 평가에서 불필요한 연산을 생략한다
- 여러 조건을 순서대로 평가할 때 결과가 결정되면 나머지 조건을 평가하지 않고 생략한다

- 예시:
  - `&&` : 앞이 false면 전체가 무조건 false
  - `||` : 앞이 true면 전체가 무조건 true

#### 5.4.4 요소 검색: findAny

`findAny` 는 현재 스트림에서 임의의 요소를 반환한다.

```
Optional<Dish> dish = menu.stream()
    .filter(Dish::isVegetarian)
    .findAny(); // 스트림에서 아무 요소 하나를 찾아 반환
```

#### 5.4.5 첫 번째 요소 찾기: findFirst


일부 스트림에는 논리적인 순서가 정해져 있을 수 있다. 이런 스트림에서 첫 번째 요소를 어떻게 찾을까?

`findFirst()` : 항상 첫 번째 요소를 반환한다 (순서 보장).

#### findAny vs findFirst 언제 사용?

메소드	특징	사용 시기
<code>findFirst()</code>	<ul style="list-style-type: none"> <li>• 순서를 보장</li> <li>• 병렬 처리 시에도 원래 순서 유지</li> <li>• 성능 비용이 더 높음</li> </ul>	순서가 중요한 경우
<code>findAny()</code>	<ul style="list-style-type: none"> <li>• 아무 요소나 빠르게 반환</li> <li>• 병렬 처리 시 더 효율적</li> <li>• 순서를 신경쓰지 않음</li> </ul>	순서가 중요하지 않은 경우

### 5.5 리듀싱

 스트림의 요소들을 하나로 결합하거나 누적해서 하나의 결과 값으로 표현한다.

예: 합, 평균, 최댓값, 최솟값 등

#### 5.5.1 요소의 합

기존 방식:

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

`numbers`의 각 요소는 결과에 반복적으로 더해진다. 리스트에서 하나의 숫자가 남을 때까지 `reduce` 과정을 반복한다. 이런 상황에서 `reduce` 를 사용하면 애플리케이션의 반복된 패턴을 추상화할 수 있다.

## reduce를 사용한 방식:

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

- $(a, b) \rightarrow a + b$  를 넘겨주면 모든 요소에 덧셈이 가능하다
- a: 누적 중간 결과
- b: 스트림의 현재 요소

Java 8에서 Integer 클래스에 두 숫자를 더하는 정적 sum 메소드를 제공한다:

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

초기값을 받지 않도록 오버로드된 reduce도 있다. 이는 Optional을 반환한다. 아무 요소도 없을 수 있으므로 Optional로 감싼 결과를 반환한다.

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

## 5.5.2 최댓값과 최솟값

최댓값과 최솟값을 찾을 때도 reduce를 활용할 수 있다. 이 때도 마찬가지로 정적 메소드가 있다.

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

---

## 5.6 실전 연습

 <https://github.com/dev-jm-1024/modern-java>

---

## 5.7 숫자형 스트림

### 기본형 특화 스트림의 필요성

```
int calories = menu.stream()  
    .map(Dish::getCalories)  
    .reduce(0, Integer::sum);
```

내부적으로 합계를 계산하기 전에 Integer를 기본형으로 언박싱해야 한다. Stream<T> 는 제네릭 타입을 사용하지만, 기본형(int, long 등)은 박싱이 필요해 성능 저하가 발생한다.

왜 sum() 메소드를 직접 호출할 수 없을까?

```
int calories = menu.stream()
    .map(Dish::getCalories)
    .sum(); // 컴파일 에러! 불가능
```

map 메소드가 Stream<T>를 생성하기 때문이다. 스트림 요소 형식은 Integer지만 인터페이스에는 sum 메소드가 없다. Stream<T>에는 sum()이라는 메소드가 존재하지 않는다.

⚠ Integer.sum(a, b)는 단순히 두 int를 더해주는 정적 메소드일 뿐, Stream의 sum()과는 관계가 없다.

## 5.7.1 기본형 특화 스트림

Stream API는 박싱 비용을 피할 수 있게 해준다:

특화 스트림	요소 타입	주요 메소드
IntStream	int	sum(), max(), min(), average()
DoubleStream	double	sum(), max(), min(), average()
LongStream	long	sum(), max(), min(), average()

💡 특화 스트림은 오직 박싱 과정에서 일어나는 효율성과 관련이 있다. 스트림에 추가 기능은 없다.

### 숫자 스트림으로 매핑

스트림을 특화 스트림으로 변환할 때 mapToInt, mapToDouble, mapToLong 세 가지 메소드를 가장 많이 사용한다.

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

mapToInt 메소드는 각 요리에서 모든 칼로리(Integer 형식)를 추출한 다음 IntStream을 반환한다. IntStream 인터페이스에서 제공하는 sum 메소드를 이용해 합계를 구한다. 스트림이 비어있으면 sum은 기본값 0을 반환한다.

### 객체 스트림으로 복원하기

숫자 스트림을 만든 다음, 원상태인 특화되지 않은 스트림으로 복원할 수 있다.

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

### 왜 복원이 필요한가?

- 기본형 특화 Stream은 성능 최적화를 위해 제한된 연산만 지원한다

- 더 복잡한 연산이 필요한 경우 일반 스트림으로 변환이 필요하다

## 기본값: OptionalInt

IntStream에서 최댓값을 찾을 때는 0이라는 기본값 때문에 잘못된 결과를 도출할 수 있다. 값이 존재하는지 여부를 가리킬 수 있는 컨테이너 클래스 Optional이 있다.

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

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

타입	특징
Optional<Integer>	오토 박싱/언박싱 필요
OptionalInt	기본형 특화, 성능 손실 없음

## 5.7.2 숫자 범위

특정 범위의 숫자를 이용해야 하는 상황이 발생한다. 예를 들어 1에서 100 사이의 숫자를 생성하려고 한다고 가정하자.

IntStream과 LongStream에서는 range 와 rangeClosed 메소드를 제공한다:

메소드	동작	예시
range()	시작값 포함, 종료값 제외	range(1, 100) → [1, 99]
rangeClosed()	시작값 포함, 종료값 포함	rangeClosed(1, 100) → [1, 100]

```
IntStream evenNumbers = IntStream.rangeClosed(1, 100) // [1, 100] 범위
    .filter(n -> n % 2 == 0); // 1부터 100까지의 짝수 스트림

System.out.println(evenNumbers.count()); // 50 출력
```

## 5.8 스트림 만들기

### 5.8.1 값으로 스트림 만들기

임의의 수를 인수로 받는 정적 메소드 Stream.of 를 이용해서 스트림을 만들 수 있다.

```
Stream<String> stream = Stream.of("Modern", "Java", "In", "Action");
stream.map(String::toUpperCase)
```



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

```
// 빈 스트림 생성
```

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

## 5.8.2 null이 될 수 있는 객체로 스트림 만들기

때로는 null이 될 수 있는 객체를 스트림으로 만들어야 할 수 있다.

기존 방식:

```
String homeValue = System.getProperty("home");
Stream<String> homeValueStream =
    homeValue == null ? Stream.empty() : Stream.of(homeValue);
```

Stream.ofNullable 사용:

```
Stream<String> homeValueStream = Stream.ofNullable(System.getProperty("home"));
```

### 🔴 System.getProperty()

- JVM, OS 환경 정보를 확인한다
- 설정 파일 없이 동작 환경에 따른 분기 처리가 가능하다
- 디버깅 시 유용하다
- 제공된 키에 대응하는 속성이 없으면 null을 반환한다

## 5.8.3 배열로 스트림 만들기

배열을 인수로 받는 정적 메소드 `Arrays.stream` 을 이용해 스트림을 만들 수 있다.

```
int[] numbers = {1, 2, 3, 4, 5, 6, 7};
int sum = Arrays.stream(numbers).sum(); // IntStream이 생성되어 sum() 사용 가능
```

## 🎯 핵심 정리

1. **필터링**: `filter` 와 `distinct` 로 원하는 요소만 선택한다
2. **슬라이싱**: `takeWhile`, `dropWhile`, `limit`, `skip` 으로 스트림을 자른다
3. **매핑**: `map` 과 `flatMap` 으로 요소를 변환한다
4. **검색과 매칭**: `anyMatch`, `allMatch`, `noneMatch`, `findAny`, `findFirst` 로 요소를 찾는다
5. **리듀싱**: `reduce` 로 모든 요소를 하나의 값으로 결합한다
6. **숫자형 스트림**: `IntStream`, `DoubleStream`, `LongStream` 으로 박싱 비용을 피한다
7. **스트림 생성**: 다양한 방법으로 스트림을 만들 수 있다