8. 컬렉션 API - 드디어 UnsupportedOperationException과 이별할 시간

8. 컬렉션 API 개선

> 자바 8에서 도입된 컬렉션 API의 주요 개선사항들을 살펴보자. 컬렉션 팩토리, 새로운 처리 메소드, 그리고 ConcurrentHashMap의 개선된 기능들을 다룬다.

8.1 컬렉션 팩토리

자바에서 적은 요소를 포함하는 리스트를 어떻게 만들까?

```
List<String> list = new ArrayList<>();
list.add("item1");
list.add("item2");
// ...
```

새 문자열을 저장하는데 많은 코드가 필요할 수 있다.
이때 `Arrays.asList()` 팩토리 메소드를 이용할 수 있다.

```
List<String> list = Arrays.asList("pen", "pencil", "book");
```

- 고정 크기의 리스트를 만들어서 요소를 갱신할 수 있다
- 새 요소 추가/삭제는 불가능하다

만약 추가를 시도한다면:

```
list.add("book"); // UnsupportedOperationException 발생
```

내부적으로 고정된 크기의 변환할 수 있는 배열로 구현되었기 때문이다.

집합으로 시도해보자:

`Arrays.asSet()` 는 없으므로 HashSet 생성자를 사용한다.

```
Set<String> result = new HashSet<>(Arrays.asList("pen", "pencil", "book"));
```

매끄럽지 못하며 내부적으로 불필요한 객체를 할당한다.

8.1.1 리스트 팩토리

`List.of` 팩토리 메소드를 이용해 간단하게 리스트를 만들 수 있다.

```
List<String> list = List.of("pen", "book", "pencil");
```

- 요소를 추가하면 예외가 발생한다 → 변경할 수 없는 리스트
- `set()`으로 변경하려고 해도 비슷한 예외가 발생한다

📊 List.of와 Arrays.asList 차이점

둘 다 `List<T>`를 만들어주는 팩토리 도우미이다.

구분	Arrays.asList	List.of
변경 가능성	크기 고정, `set()` 가능, `add()` / `remove()` 불가	크기 고정, 모든 변경 불가
배킹(Backing)	원본 배열의 뷰, 상호 영향	독립적인 불변 컬렉션
null 허용	null 허용	null 요소 금지

변경 가능성

- `Arrays.asList`는 크기가 고정적이며 `set()`이 가능하다. 하지만 `add()` / `remove()`는 불가능하다 → `UnsupportedOperationException` 발생
- `List.of`도 크기가 고정적이며 `add` / `remove` / `set` 모두 불가능하다

배킹(Backing)

- `Arrays.asList`는 원본 배열을 그대로 뷰로 감싸며, 배열 값이 바뀌면 리스트도 바뀌고 리스트에서 `set()` 하면 배열도 바뀐다
- `List.of`는 독립적인 불변 컬렉션이며 원본과 연결되지 않는다

null 허용

- `Arrays.asList`는 null이 허용되는 반면, `List.of`는 null 요소를 금지한다

8.1.2 집합 팩토리

`List.of`와 비슷한 방법으로 바꿀 수 없는 집합을 만들 수 있다.

```
Set<String> result = Set.of("book", "pen", "pencil");
```

중복된 요소를 제공해 집합을 생성하려고 하면 요소가 중복되어 있다는 설명과 함께 `IllegalArgumentException`이 발생한다.
집합은 오직 고유한 요소만 포함한다.

8.1.3 맵 팩토리

맵은 집합/리스트에 비해 조금 복잡하다.
키(key)와 값(value)이 필요하기 때문이다.

Map.of

```
Map<String, Integer> m1 = Map.of(
    "one", 1,
    "two", 2,
    "three", 3
);
```

10개 이하의 키와 값을 가진 작은 맵을 만들 때 유용하다.
최대 10개의 키/값 쌍을 지원한다.

그 이상일 경우 `Map.Entry<K, V>`를 사용한다:

```
Map<String, Integer> m2 = Map.ofEntries(
    entry("one", 1),
    entry("two", 2),
    entry("three", 3)
);
```

개수 제한이 없다.

8.2 리스트와 집합 처리

자바 8에서는 List, Set 인터페이스에 다음과 같은 메소드가 추가되었다.

🛠️ 새로운 메소드들

1) removeIf

- 프레디케이트를 만족하는 요소를 제거한다
- List나 Set을 구현하거나 그 구현을 상속받은 모든 클래스에서 이용 가능하다

2) replaceAll

- 리스트에서 사용 가능하다
- `UnaryOperator` 함수를 이용해 요소를 바꾼다

3) sort

- List 인터페이스에서 제공한다
- 리스트를 정렬한다

8.2.1 removeIf

다음과 같은 코드는 문제가 있다:

```
for(Transaction transaction : transactions) {
    // Iterator<Transaction> iterator = transactions.iterator(); 와 같다
    // Transaction transaction = iterator.next();
```

```
    if(Character.isDigit(transaction.getReferenceCode().charAt(0))) {
        transactions.remove(transaction); // 반복하면서 별도의 두 객체를 통해 컬렉션 변경
    }
}
```

❌ **문제점들:**

- `ConcurrentModificationException` 발생
  - 컬렉션을 반복(iterator) 중에 구조를 변경하면 자바의 fail-fast 이터레이터가 던지는 예외
- for-each 루프는 Iterator 객체를 사용한다
  - 반복 중에는 Iterator가 제공하는 메소드만 사용해야 한다
- 두 개의 개별 객체가 컬렉션을 관리한다
  - Iterator 객체: `next()`, `hasNext()`를 이용해 소스를 질의한다
  - Collection 객체 자체: `remove()`를 호출해 요소를 삭제한다

코드가 복잡해졌으며, 이 패턴은 자바 8의 `removeIf` 메소드로 변경 가능하다.
코드가 단순해지고 버그를 예방할 수 있다.

`removeIf`는 삭제할 요소를 가리키는 프레디케이트를 인수로 받는다:

```
transactions.removeIf(
    transaction -> Character.isDigit(transaction.getReferenceCode().charAt(0))
);
```

하지만 제거가 아닌 변경이 필요한 상황도 존재한다.
이때 `replaceAll`을 사용한다.

8.2.2 replaceAll 메소드

List 인터페이스의 `replaceAll`을 이용해 리스트의 각 요소를 새로운 요소로 바꿀 수 있다.

1) Stream 사용

```
result.stream()
    .map(code -> Character.toUpperCase(code.charAt(0)) + code.substring(1))
    .collect(Collectors.toList())
    .forEach(System.out::println);
```

하지만 이 코드는 새 문자열 컬렉션을 만든다.
우리가 원하는 것은 **기존 컬렉션을 바꾸는 것**이다.

2) ListIterator 사용

```
for(ListIterator<String> iterator = referenceCodes.listIterator();
    iterator.hasNext(); ) {
    String code = iterator.next();
```

```
        iterator.set(Character.toUpperCase(code.charAt(0)) + code.substring(1));
    }
```

컬렉션 객체를 Iterator 객체와 혼용하면 반복과 컬렉션 변경이 동시에 이루어지면서 쉽게 문제를 일으킨다.

✅ **개선된 방법:**

```
referenceCodes.replaceAll(
    code -> Character.toUpperCase(code.charAt(0)) + code.substring(1)
);
```

8.3 맵 처리

자바 8에서는 Map 인터페이스에 몇 가지 디폴트 메소드를 추가했다.

8.3.1 forEach 메소드

맵에서 키와 값을 반복하며 확인하는 작업은 번거롭다.
`Map.Entry<K, V>`의 반복자를 이용해 맵의 항목 집합을 반복할 수 있다.

```
for(Map.Entry<String, Integer> entry : ageOfFriends.entrySet()) {
    String friend = entry.getKey();
    Integer age = entry.getValue();
    System.out.println(friend + " is " + age + " years old");
}
```

자바 8부터 Map 인터페이스는 `BiConsumer`(키와 값을 인수로 받음)를 인수로 받는 `forEach` 메소드를 지원하므로 코드를 조금 더 간단하게 구현할 수 있다.

```
ageOfFriends.forEach((friend, age) ->
    System.out.println(friend + " is " + age + " years old")
);
```

8.3.2 정렬 메소드

맵의 항목을 값 또는 키를 기준으로 정렬할 수 있다.

- `Entry.comparingByValue`
- `Entry.comparingByKey`

```
result.entrySet().stream()
    .sorted(Entry.comparingByKey())
    .forEachOrdered(System.out::println);
```

이제 요청한 키가 Map에 존재하지 않을 경우 어떻게 처리하느냐가 문제다.
`getOrDefault` 메소드를 사용한다.

8.3.3 getOrDefault 메소드

기존에 찾으려는 키가 존재하지 않으면 null이 반환되었다.
`NullPointerException`을 방지하려면 요청 결과가 null인지 확인해야 했다.

이때 `getOrDefault` 메소드를 이용하면 쉽게 이 문제를 해결할 수 있다.
첫 번째 인수로 키, 두 번째 인수로 기본 값을 받는다.
키가 존재하지 않으면 두 번째 인수로 받은 **기본 값**을 반환한다.

```
Map<String, String> movies = Map.ofEntries(
    entry("Raphael", "Star Wars"),
    entry("Olivia", "James Bond")
);

System.out.println(movies.getOrDefault("Olivia", "Matrix"));   // James Bond 출력
System.out.println(movies.getOrDefault("Thibaut", "Matrix"));  // Matrix 출력
```

키가 존재하더라도 값이 null인 상황에서는 `getOrDefault`가 null을 반환한다.
즉, 키가 존재하느냐의 여부에 따라서 두 번째 인수가 반환될지 결정한다.

8.3.4 계산 패턴

맵에 키가 존재하는지 여부에 따라 어떤 동작을 실행하고 결과를 저장해야 하는 상황이 필요할 때가 있다.

🔧 계산 메소드들

1) computeIfAbsent

- 제공된 키에 해당하는 값이 없으면(값이 없거나 null), 키를 이용해 새 값을 계산하고 맵에 추가한다

2) computeIfPresent

- 제공된 키가 존재하면 새 값을 계산하고 맵에 추가한다

3) compute

- 제공된 키로 새 값을 계산하고 맵에 저장한다

```
lines.forEach(line ->
    dataToHash.computeIfAbsent(line,       // line은 맵에서 찾을 키
            this::calculateDigest));       // 키가 존재하지 않으면 실행

private byte[] calculateDigest(String key) {
    return messageDigest.digest(key.getBytes(StandardCharsets.UTF_8));
}
```

`Map<K, List<V>>`에 요소를 추가하려면 항목이 초기화되었는지 확인해야 한다.

기존 방식:

```
String friend = "Raphael";
List<String> movies = friendsToMovies.get(friend);

if(movies == null) {
    movies = new ArrayList<>();
    friendsToMovies.put(friend, movies);
}
movies.add("Star Wars");
```

개선된 방식:

```
friendsToMovies.computeIfAbsent(
    "Raphael", name -> new ArrayList<>()
).add("Star Wars");
```

현재 키와 관련된 값이 맵에 존재하며 null이 아닐 때만 새 값을 계산한다.

8.3.5 삭제 패턴

자바 8에서는 키가 특정 값과 연관되어 있을 때만 항목을 제거하는 오버로드된 버전의 메소드를 제공한다.

기존 방식:

```
String key = "Raphael";
String value = "Jack Reacher 2";

if(favoriteMovies.containsKey(key) &&
   Objects.equals(favoriteMovies.get(key), value)) {
    favoriteMovies.remove(key);
    return true;
} else {
    return false;
}
```

개선된 방식:

```
favoriteMovies.remove(key, value);
```

키가 특정한 값과 연관되었을 때만 항목을 제거한다.

8.3.6 교체 패턴

맵의 항목을 바꾸는데 사용할 수 있는 두 개의 메소드가 맵에 추가되었다.

1) replaceAll

- `BiFunction`을 적용한 결과로 각 항목의 값을 교체한다

- 이전에 살펴본 List의 `replaceAll`과 비슷한 동작을 수행한다

2) replace

- 키가 존재하면 Map의 값을 바꾼다
- 키가 특정 값으로 매핑되었을 때만 값을 교체하는 오버로드 버전이 있다

```
favoriteMovies.replaceAll((friend, movie) -> movie.toUpperCase());
```

BiFunction 이해하기:

```
BiFunction<K, V, V> function = (key, value) -> {
    // key: 현재 처리 중인 키
    // value: 현재 처리 중인 값
    // return: 새로운 값

    return newValue;
};
```

8.3.7 합침: merge(K key, V value, BinaryOperator function)

두 개의 맵을 합친다고 가정하자.

```
Map<String, String> family = Map.ofEntries(
 entry(“Teo”, “Star wars”),
 entry(“Cristina”, “James Bond”)
);

Map<String, String> friends = Map.ofEntries(
 entry(“Raphel”, “Star wars”),
);

Map<String, String> everyone = new HashMap<>(family);
everyone.putAll(friends);

// family Map을 기반으로 새로운 HashMap 만든다
// everyone 안에는 처음에 family의 값이 들어있다
// friends Map의 모든 항목을 everyone에 추가
```

중복된 키가 없다면 위 코드는 잘 동작한다.
값을 좀 더 유연하게 합쳐야 한다면 새로운 `merge` 메소드를 이용할 수 있다.
중복된 키를 어떻게 합칠지 결정하는 `BiFunction` 을 인수로 받는다.

```
Map<String, String> family = Map.ofEntries(
 entry(“Teo”, “Star wars”),
 entry(“Cristina”, “James Bond”)
);

Map<String, String> friends = Map.ofEntries(
 entry(“Raphel”, “Star wars”),
 enry(“Cristina”, “Matrix”)
);

Map<String, String> everyone = new HashMap<>(family);
friends.forEach( (k,v) →
 everyone.merge( k, v, (movie1, movie2) → movie + “ & “ + movie2 )
);

System.out.println(everyone);
```

작동 흐름

1)

k = Raphel , v = Star Wars

everyone.get(“Raphel”)
: Key가 존재하지 않음 → 실행하지 않는다

2)

k = Cristina v = Matrix

everyone.get(“Raphel”)
: “James Bond” 존재 → 결과 O
: remappingFunction 실행 : James Bond & Matrix

지정된 키와 연관된 값이 없거나 값이 null 이면
merge는 key를 null 이 아닌 값과 연결한다

아니면 merge는 연결된 값을 주어진 매핑함수의 결과 값으로 대치하거나
결과가 null 이면 항목 제거한다

8.4 개선된 ConcurrentHashMap

`ConcurrentHashMap` 클래스는 동시성 친화적이다.
Java에서 멀티스레드 환경에서 안전하게 동작하는 HashMap 구현체이다.

흔히 HashMap은 동기화(synchronization)가 전혀 없기 때문에 여러 Thread가 동시에 접근하거나 수정하면 데이터 불일치, 무한루프, `ConcurrentModificationException` 같은 문제가 발생한다.

내부 자료구조의 특정 부분만 잠궈 동시 추가, 갱신 작업을 허용한다.

8.4.1 리듀스와 검색

`ConcurrentHashMap`은 스트림에서 봤던 것처럼 비슷한 종류의 세 가지 새로운 연산을 지원한다.

🔄 세 가지 연산

forEach

- 각 키, 값 쌍에 주어진 액션을 실행한다

reduce

- 모든 키/값 쌍을 제공된 리듀스 함수를 이용해 결과로 합친다

search

- null이 아닌 값을 반환할 때까지 각 키/값 쌍에 함수를 적용한다

키에 함수 받기, 값, `Map.Entry`, `(key, value)` 인수를 이용한 연산 형태도 지원한다.

📋 연산 형태들

- **키, 값으로 연산**: `forEach`, `reduce`, `search`
- **키로 연산**: `forEachKey`, `reduceKeys`, `searchKeys`
- **값으로 연산**: `forEachValue`, `reduceValues`, `searchValues`
- **Map.Entry 객체로 연산**: `forEachEntry`, `reduceEntries`, `searchEntries`

⚠️ 주의사항

이들 연산은 `ConcurrentHashMap`의 상태를 잠그지 않고 연산을 수행한다.
따라서 연산에 제공된 함수는 계산이 진행되는 동안 바뀔 수 있는 객체, 값, 순서 등에 의존하지 않아야 한다.

🎯 병렬성 기준

이들 연산에 병렬성 기준 값을 지정해야 한다.

- 맵의 크기가 주어진 기준 값보다 작으면 순차적 연산을 진행한다
- 기준 값을 **1**로 지정하면 공통 스레드 풀을 이용해 병렬성을 극대화한다
- **Long.MAX_VALUE**를 기준 값으로 설정하면 1개의 스레드로 연산을 실행한다

📚 정리

자바 8의 컬렉션 API 개선사항들은 다음과 같다:

- **컬렉션 팩토리**: `List.of`, `Set.of`, `Map.of`로 간편한 불변 컬렉션 생성
- **새로운 처리 메소드**: `removeIf`, `replaceAll`, `sort`로 더 안전하고 간결한 컬렉션 조작
- **맵 처리 개선**: `forEach`, 정렬, `getOrDefault`, 계산/삭제/교체 패턴, `merge`로 풍부한 맵 조작

- **ConcurrentHashMap 개선**: 스레드 안전한 리듀스, 검색, forEach 연산 지원

이러한 개선사항들은 코드의 가독성을 높이고 버그를 줄이며, 성능까지 향상시키는 효과를 가져다준다.