8장. 컬렉션 API 개선

우리의 개발 라이프에 생기를 더해주는 것 중 하나가 컬렉션이다. 바퀴를 재 발명하지 않고 리스트, 집합, 맵 등의 완성도 있는 자료구조를 객체 선언해서 사용함으로써 개발 속도를 높이면서도 신뢰성 있는 코드를 작성할 수 있다.

컬렉션 API는 JDK 8 전, 후로도 많은 사용성이 개선되었는데, 이번 장에서는 이에 대한 내용을 주로 다룬다.

컬렉션 팩토리

정수를 담는 리스트를 선언하는 방법. 근데 아래 방법으로 선언을 하는 경우에는 엘리먼트의 변경이 불가하다!

```
// 팩토리 메소드 미사용
List<Integer> list = new ArrayList<>();
list.add(1);
list.add(2);
list.add(3);
list.add(4);
list.add(5);


// 팩토리 메소드 사용
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);
```

팩토리 메소드로 초기화시 특징은?

1. 엘리먼트 변경 불가

고정된 크기의 상수 배열로 구현된 탓에 add, remove 등 사이즈 변경을 일으키는 모든 메소드의 사용이 불가하고, 사용시 UncheckedException인 `UnsupportedOperationException`이 발생한다.

2. null을 허용하지 않음.

내부적으로 `Objects.requireNonNull` 을 사용해 엘리먼트의 null 여부를 체크 함.

(`java.util.ArrayList`는 nullable)

`Arrays.asList` 변경이 불가한 이유는?

asList에서 사용하는 ArrayList는 java.util.ArrayList가 아닌 Arrays 내부의 정적인 클래스인 ArrayList를 사용하고 있고 엘리먼트가 상수로 초기화 되기 때문 입니다.

다만 set, replace 등의 메소드는 지원이 되므로 세부 항목에 대한 변경은 가능합니다.

(상속한 클래스는 AbstractList로 동일한 추상 클래스 입니다!)

```
public class Arrays {

...

    @SafeVarargs
    @SuppressWarnings("varargs")
    public static <T> List<T> asList(T... a) {
        return new ArrayList<>(a);
    }

...

    private static class ArrayList<E> extends AbstractList<E>
        implements RandomAccess, java.io.Serializable
    {
        private static final long serialVersionUID = -2764017481108945198L;
        private final E[] a;

        ArrayList(E[] array) {
            a = Objects.requireNonNull(array);
        }
        ...
    }

...

}
```

`Array.asList` 와 달리 사이즈 변경이 가능한 컬렉션을 생성하는 방법은?

```
List<Integer> reCreatedList = new ArrayList(Arrays.asList(1,2,3,4,5));
Set<Integer> reCreatedList2 = new HashSet(Arrays.asList(1,2,3,4,5));
List<Integer> reCreatedList3 = IntStream.of(1, 2, 3).boxed()
                .collect(Collectors.toList());
```

Java 9부터 지원하는 리스트 팩토리 메소드

`Arrays.asList` 처럼 고정 크기의 엘리먼트 배열을 사용하며, 내부적으로 변경을 일으키는 모든 연산이 사용이 불가합니다. 사용시 `UnsupportedOperationException`이 발생합니다.

이 점에 있어 set, replace 까지는 허용 되었던 Arrays.asList와는 차이를 보입니다.

```
List<String> niniz = List.of("스카피", "죠르디", "앙몬드");
```

`Arrays.asList`와의 차이점은?

`List.of` 메소드가 10개 까지 10개의 인수를 받을 수 있는 오버로드 메소드를 제공한다.

인수가 1~2개 까지는 `ImmutableCollections.List12` 스태틱 클래스를 사용하고 생성자의 인수가 고정 되어 있다. 반면에 3~10개 부터는 가변 인수(varargs)로 된 생성자를 가지는 `ImmutableCollections.ListN` 스태틱 클래스를 사용한다.

```
public static List<T> of(String t1, String t2) {
}
```

가변 인수의 특징은 인수로 넘긴 엘리먼트를 배열로 할당하고 초기화 하므로 가비지 컬렉션 비용을 부채로 가지게 된다.

Java 9부터 지원하는 세트 팩토리 메소드

```
Set<String> niniz = Set.of("스카피", "죠르디", "앙몬드");
```

Java 9부터 지원하는 맵 팩토리 메소드

```
Map<String, String> niniz
  = Map.of("스카피", "토끼", "죠르디", "공룡", "앙몬드", "하프물범");

Map<String, String> map = Map.ofEntries(
                entry("죠르디", "공룡"),
                entry("스카피", "토끼"),
                entry("앙몬드", "하프물법")
        );
```

리스트와 집합 처리

자바 8도 지원

`Predicate`, `UnaryOperator`를 사용해서 리스트의 요소를 제어할 수 있는 API가 제공된다.

```
// removeIf: Predicate에 만족하는 요소를 제거한다.
List<Integer> onetoHundredList = IntStream.rangeClosed(1, 100).boxed()
          .collect(Collectors.toList());

onetoHundredList.removeIf(num -> num < 30);

// 1 ~ 100 => 30 ~ 100
System.out.println(onetoHundredList);
```

```
// replaceAll: UnaryOperator를 사용해 각 요소를 변경한다.

List<Integer> onetoHundredList = IntStream.rangeClosed(1, 100).boxed()
                .collect(Collectors.toList());

onetoHundredList.replaceAll(num -> num + 100);

// 1 ~ 100 => 101 ~ 200
System.out.println(onetoHundredList);
```

맵 처리

forEach 메서드

```
Map<String, String> niniz
  = Map.of("스카피", "토끼", "죠르디", "공룡", "앙몬드", "하프물범");
niniz.forEach(element -> {
    // something.
});
```

정렬 메서드

맵을 키와 값을 기준으로 정렬 할 수 있다.

```
Map niniz = Map.of("스카피", "토끼", "죠르디", "공룡", "앙몬드", "하프물범");
```

```
niniz.entrySet()
        .stream()
        .sorted(Map.Entry.comparingByKey())
        .forEach(key -> {
            System.out.println(key);
        });

niniz.entrySet()
        .stream()
        .sorted(Map.Entry.comparingByValue())
        .forEach(key -> {
            System.out.println(key);
        });
```

HashMap 성능

최초에 초기화할 때는 LinkedList 구조로 된 Node를 사용하지만 버킷 내에 요소의 수가 증가하면 내부적으로 TreeNode로 전환된다.

NAVER D2

https://d2.naver.com/helloworld/831311

`gerOrDefault` 메소드

키가 존재하지 않을 때 get을 하면 null이 반환되어, NPE을 유발하게 되는 계기가 된다. 이를 방지하기 위해 특정 키에 값이 없을 경우 기본적인 값을 보장해주는 메소드다.

```
Map<String, String> map = new HashMap<>();
map.put("죠르디", "공룡");

System.out.println(map.getOrDefault("죠르디", "토끼"));
System.out.println(map.getOrDefault("헬로우", "방가방가"));
```

계산 패턴

```
// computeIfAbsente: 제공된 키에 해당하는 값이 없으면 (혹은 null), 키를 이용해 새 값을
//                   계산하고 맵에 추가한다.
Map<String, String> map = new HashMap<>(Map.ofEntries(
                entry("죠르디", "공룡"),
```

```
                entry("스카피", "토끼"),
                entry("앙몬드", "하프물법")
        ));

System.out.println(map);

// nothing work
map.computeIfAbsent("죠르디", (key) -> {
    System.out.println(key);
    System.out.println("----");

    return "강한 공룡";
});

// 새로운 값 추가
map.computeIfAbsent("라이언", (key) -> {
    System.out.println(key);
    System.out.println("----");

    return "사자";
});

System.out.println(map);


// computeIfAbsente: 제공된 키에 해당하는 값이 없으면 (혹은 null), 키를 이용해 새 값을
//                   계산하고 맵에 추가한다.
Map<String, String> map = new HashMap<>(Map.ofEntries(
                entry("죠르디", "공룡"),
                entry("스카피", "토끼"),
                entry("앙몬드", "하프물법")
        ));

System.out.println(map);

// nothing work
map.computeIfPresent("사자", (key, value) -> {
    System.out.println(key);
    System.out.println(value);
    System.out.println("----");

    return "강한 공룡";
});

// 죠르디의 값이 강한 공룡으로 변경
map.computeIfPresent("공룡", (key, value) -> {
    System.out.println(key);
    System.out.println(value);
    System.out.println("----");

    return "강한 공룡";
});

System.out.println(map);



// compute: 제공된 키와 값으로 계싼하고 맵에 저장한다.
```

```
Map<String, String> map = new HashMap<>(Map.ofEntries(
                entry("죠르디", "공룡"),
                entry("스카피", "토끼"),
                entry("앙몬드", "하프물법")
        ));

System.out.println(map);

map.compute("죠르디", (key, value) -> {
    System.out.println(key);
    System.out.println(value);
    System.out.println("----");

    return "강한 공룡";
});

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

삭제 패턴

Map이 key와 value 모두 일치하는 경우에 삭제할 수 있는 메소드를 제공한다.

```
Map<String, String> map = new HashMap<>(Map.ofEntries(
                entry("죠르디", "공룡"),
                entry("스카피", "토끼"),
                entry("앙몬드", "하프물법")
        ));

map.remove("죠르디", "곰");
System.out.println(map);

map.remove("스카피", "토끼");
System.out.println(map);
```

합침

특정 맵의 데이터를 모두 put

```
Map<String, String> niniz = new HashMap<>(Map.ofEntries(
                entry("죠르디", "공룡"),
                entry("스카피", "토끼"),
                entry("앙몬드", "하프물법")
        ));
Map<String, String> kakao = new HashMap<>(Map.ofEntries(
        entry("라이언", "사자")
));
```

```
niniz.putAll(kakao);

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

ConcurrentHashMap

HashMap에 비해 동시성이 뛰어난 버전으로, 락을 하는 곳을 최소화 함으로써 동시에 추가 및 수정 작업이 가능하다. 그래서 HashTable에 비해 읽기 쓰기 연산이 월등하다.

리듀스와 검색

ConcurrentHashMap은 forEach, reduce, search 연산을 추가적으로 지원한다.

계수

맵에서 보관하는 값의 수가 int로 표현되지 않을 가능 성이 있는경우 long type의 반환 타입을 가지는 mappingCount 메소드를 활용하는 것이 좋다.