Use EnumMap instead of ordinal indexing

상황

array나 list의 Index로 사용하기 위해 ordinal method를 사용하는 경우를 종종 볼 수 있습니다. 다음 예제에서 살펴보죠.

예제 - Plant

정원 내에 있는 식물의 LifeCycle에 따라 분류하여 묶고 싶다고 합시다.

이를 위해선 각 사이클 별로 3가지 Set을 선언하고, garden을 돌며 각 식물을 적절한 set에 넣는 겁니다.

```
class Plant {
  enum LifeCycle { ANNUAL, PERENNIAL, BIENNIAL }
  final String name;
  final LifeCycle lifeCycle;
  Plant(String name, LifeCycle lifeCycle) {
    this.name = name;
    this.lifeCycle = lifeCycle;
  }
  @Override public String toString() {
    return name;
  }
}
```

몇몇 프로그래머들은 ordinal을 이용하여 리스트로 담아낼 수도 있습니다.

`[Set<ANNUAL>, Set<PERENIAL>, Set<BIENNIAL>]`

이런 식으로 말이죠. 코드는 다음과 같습니다.

```
// Using ordinal() to index into an array - DON'T DO THIS!
Set<Plant>[] plantsByLifeCycle =
  (Set<Plant>[]) new Set[Plant.LifeCycle.values().length];

for (int i = 0; i < plantsByLifeCycle.length; i++)
  plantsByLifeCycle[i] = new HashSet<>();

for (Plant p : garden) plantsByLifeCycle[p.lifeCycle.ordinal()].add(p);
```

```
// Print the results
for (int i = 0; i < plantsByLifeCycle.length; i++) {
  System.out.printf("%s: %s%n",
                    Plant.LifeCycle.values()[i], plantsByLifeCycle[i]);
}
```

문제

이전에 배운 아이템들을 떠올리시면 문제가 많은 코드라는 것을 바로 알아차릴 수 있을 겁니다.

1. 배열과 generic은 어울리지 못합니다.

이전 아이템에 배운 내용과 동일합니다. 런타임 중에 paramterized type이 지워지기 때문이죠.

2. ordinal을 사용한다면 type safe하지 못합니다.

ordinal의 return 타입은 int입니다. 따라서 올바른 int값을 입력하길 바래야죠.

임의의 array 범위 내 int값이 들어와서 에러를 눈치채지 못한 채 앱이 돌아갈 수도 있고, 넘는 값이 들어온다면 `ArrayIndexOutOfBoundsException`이 발생합니다.

EnumMap 쓰세요~

```
// Using an EnumMap to associate data with an enum
Map<Plant.LifeCycle, Set<Plant>> plantsByLifeCycle =
  new EnumMap<>(Plant.LifeCycle.class);

for (Plant.LifeCycle lc : Plant.LifeCycle.values())
  plantsByLifeCycle.put(lc, new HashSet<>());

for (Plant p : garden)
  plantsByLifeCycle.get(p.lifeCycle).add(p);

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

Enum을 Key값으로 Map을 만들 수 있는 EnumMap을 제공해줍니다. 여기서 LifeCycle이라는 bounded type token으로 key값을 써줍니다.(Item-33)

이점

1. 짧고
2. 명료하고
3. 출력 결과에 굳이 레이블을 달 필요가 없습니다.
4. 배열 인덱스 계산하는 과정에서 오류가 없습니다.
5. 속도도 비등합니다.

   내부적으로 array를 사용하기 때문에 ordinal과 속도가 비등하죠. 즉, 외부는 Map으로 노출시키고, 내부 구현을 숨긴 사례라 할 수 있죠.

stream으로 구현하기

1) bad-case (naive stream-based approach)

```
// Naive stream-based approach - unlikely to produce an EnumMap!
System.out.println(Arrays.stream(garden)
           .collect(groupingBy(p -> p.lifeCycle)));
```

보기에는 간결하나, EnumMap이 아닌 Map의 구현체를 적용했기 때문에 메모리도 많이 차지하고 느립니다.

2)good-case (groupBy 100% 이용하기)

```
// Using a stream and an EnumMap to associate data with an enum
System.out.println(Arrays.stream(garden)
                   .collect(groupingBy(p -> p.lifeCycle,
                                       () -> new EnumMap<>(LifeCycle.class),
toSet())));
```

EnumMap과 차이점

EnumMap version은 항상 모든 식물의 Life-cycle key를 생성하지만 stream based map은 식물에 존재하는 life-cycle만 key값을 가지죠.

예제 - Phase

한 상태에서 다른 상태로 변할 때 전이의 이름을 출력하는 예제입니다. `(from, to) -> transition`

예시로 고체에서 액체 -> 융해

출력하도록 하는 것이죠.

Map<> map1

map.get(SOLID) => MAP<phase, transition> map2

map2.get(LIQUID) => MELT

base-case

```
// Using ordinal() to index array of arrays - DON'T DO THIS!
public enum Phase {
  SOLID, LIQUID, GAS;

  public enum Transition {
    MELT, FREEZE, BOIL, CONDENSE, SUBLIME, DEPOSIT;

    // Rows indexed by from-ordinal, cols by to-ordinal
    private static final Transition[][] TRANSITIONS = {
      { null,    MELT,     SUBLIME },
      { FREEZE,  null,     BOIL    },
      { DEPOSIT, CONDENSE, null    }
    };

    // Returns the phase transition from one phase to another
    public static Transition from(Phase from, Phase to) {
      return TRANSITIONS[from.ordinal()][to.ordinal()];
    }
  }
}
```

이차원 배열로 표현하여 겉보기에는 멋있지만 다음 문제를 가지고 있습니다.

문제

1. ordinal의 indexing문제
2. type safety
3. 새로운 상태 추가시 TRANSITIONS를 수정해야줘야 함.
    - 에러 발생하기 쉬운 구조가 되죠.
4. 메모리 문제

    Phase가 추가될 때마다 아무런 쓸모없는 null이 메모리를 차지하게 됩니다.

good-case (EnumMap)

EnumMap으로 보다 효율적으로 구현할 수 있습니다.

각 상 전이는 phase enums로 표현할 수 있죠.

from을 나타내는 Pharse EnumMap -> to를 나타내는 Pharse EnumMap -> transition

이 되도록 하는것이죠

> 가만보니 lambda calculus에서 본 모양이네요
>
> (from -> (to -> transition)) (pharse1) (pharse2)

```
// Using a nested EnumMap to associate data with enum pairs
public enum Phase {
    SOLID, LIQUID, GAS;

    public enum Transition {
        MELT(SOLID, LIQUID), FREEZE(LIQUID, SOLID),
        BOIL(LIQUID, GAS), CONDENSE(GAS, LIQUID),
        SUBLIME(SOLID, GAS), DEPOSIT(GAS, SOLID);
        private final Phase from;
        private final Phase to;

        Transition(Phase from, Phase to) {
            this.from = from;
            this.to = to;
        }

        // Initialize the phase transition map
        private static final Map<Phase, Map<Phase, Transition>>
                m = Stream.of(values()).collect(groupingBy(t -> t.from,
                () -> new EnumMap<>(Phase.class),
                toMap(t -> t.to, t -> t,
                        (x, y) -> y, () ->
                                new EnumMap<>(Phase.class))));

        public static Transition from(Phase from, Phase to) {
            return m.get(from).get(to);
        }
    }
}
```

설명

새로운 상태추가

이제 플라즈마라는 상태를 추가해보겠습니다. from method를 수정할 필요없이 새로운 phase와 transition만 추가해주면 됩니다.

```
// Adding a new phase using the nested EnumMap implementation
public enum Phase {
    SOLID, LIQUID, GAS, PLASMA;

    public enum Transition {
        MELT(SOLID, LIQUID), FREEZE(LIQUID, SOLID),
        BOIL(LIQUID, GAS), CONDENSE(GAS, LIQUID),
        SUBLIME(SOLID, GAS), DEPOSIT(GAS, SOLID),
        IONIZE(GAS, PLASMA), DEIONIZE(PLASMA, GAS);

      ... // Remainder unchanged
    }
}
```

기존 방법과 비교해봅시다. 기존엔 플라즈마 상태를 추가한다면 3x3 -> 4x4 TRANSITIONS 배열로 나타내야 했습니다. 이 과정에서 실수로 순서를 바꾼다면 에러를 발생시킬 것이며, 메모리도 더 많이 차지했겠죠.

> 참고로 이 예제에서는 같은 상태일 경우 null 값을 반환하도록 하였습니다. NullPointException을 던지기 때문에 우아한 코드는 아니지만, 개선하려면 코드가 복잡해지기 때문에 이 부분은 제외했습니다.

요약

ordinal을 index로 쓰지 마시고 EnumMap쓰세요.

만약에 다차원을 나타내고 싶으면 `EnumMap<..., EnumMap<...>>`을 사용하세요.