
이펙티브 자바

item37 ~ item 38

24/03/19

아이템 37 ordinal 인덱싱 대신 EnumMap을 사용하라

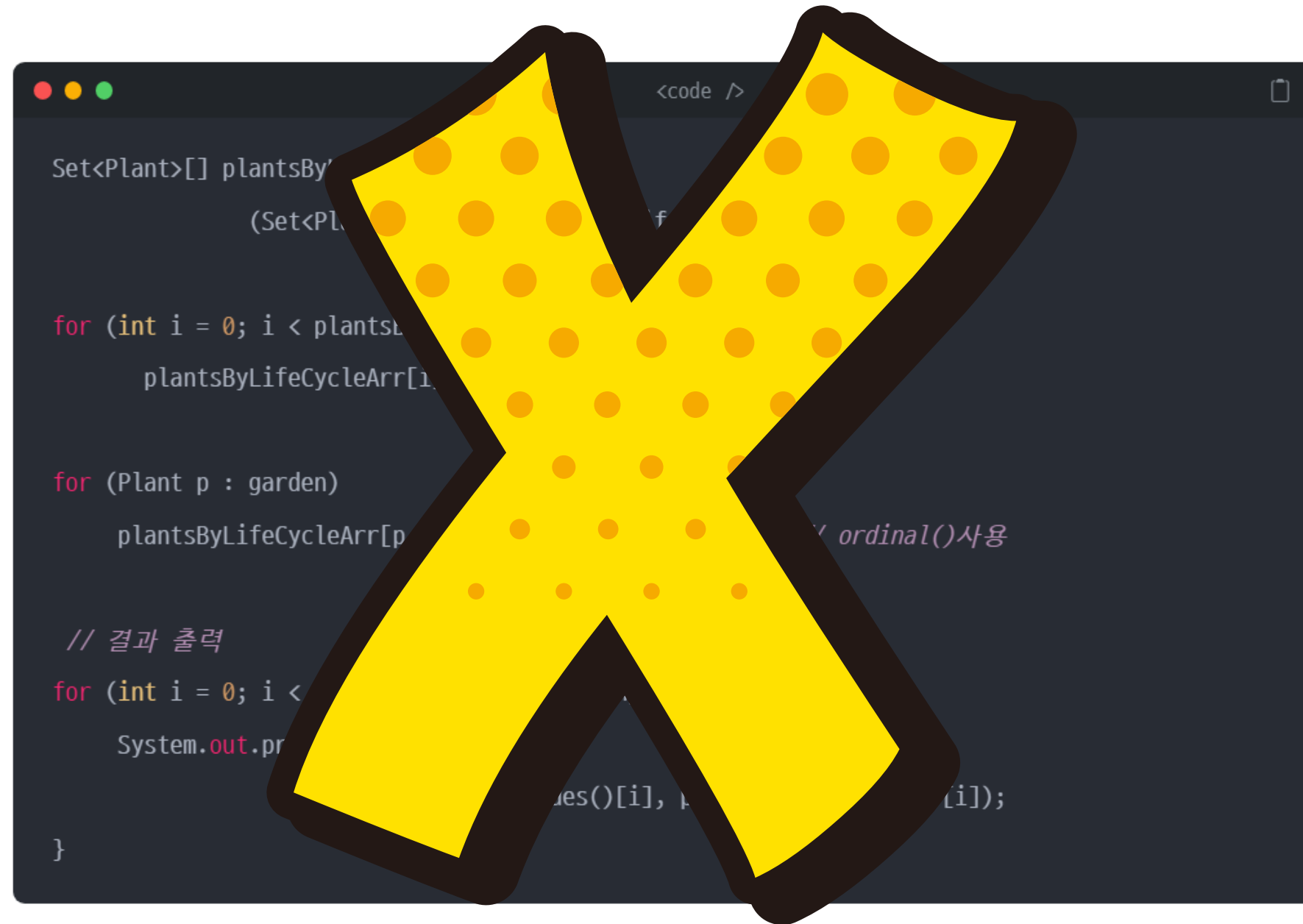
배열이나 리스트에서 원소를 꺼낼 때와 같이 인덱싱이 필요할 때는, ordinal() 메서드(아이템 35)로 인덱스를 얻을 수 있다.

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

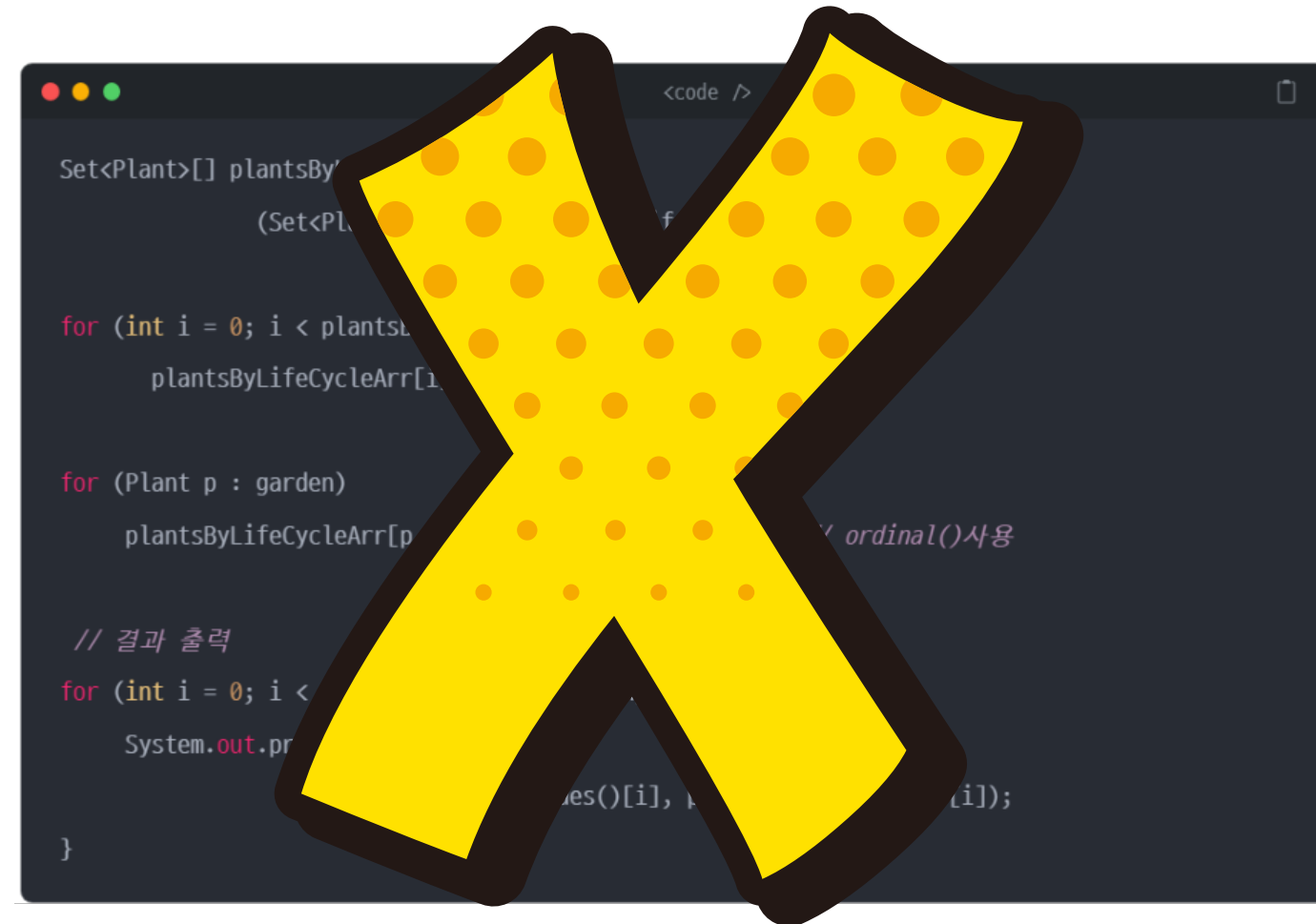
아이템 37 ordinal 인덱싱 대신 EnumMap을 사용하라

```
Set<Plant>[] plantsByLifeCycleArr =  
    (Set<Plant>[]) new Set[Plant.LifeCycle.values().length];  
  
for (int i = 0; i < plantsByLifeCycleArr.length; i++)  
    plantsByLifeCycleArr[i] = new HashSet<>();  
  
for (Plant p : garden)  
    plantsByLifeCycleArr[p.lifeCycle.ordinal()].add(p); // ordinal()사용  
  
// 결과 출력  
for (int i = 0; i < plantsByLifeCycleArr.length; i++) {  
    System.out.printf("%s: %s%n",  
        Plant.LifeCycle.values()[i], plantsByLifeCycleArr[i]);  
}
```

아이템 37 ordinal 인덱싱 대신 EnumMap을 사용하라



아이템 37 ordinal 인덱싱 대신 EnumMap을 사용하라



1. 배열은 제네릭과 호환되지 않으니(아이템 28) 비검사 형변환을 수행해야 한다.
2. 정확한 정수값을 사용한다는 것을 직접 보증해야 한다. (정수는 타입 안전하지 않음)

아이템 37 ordinal 인덱싱 대신 EnumMap을 사용하라

EnumMap이란?

EnumMap은 열거 타입을 키로 가지며,
배열과 마찬가지로
실질적으로 열거 타입 상수를 값으로 매핑
하는 일을 한다.

아이템 37 ordinal 인덱싱 대신 EnumMap을 사용하라

EnumMap이란?

EnumMap은 열거 타입을 키로 가지며,
배열과 마찬가지로
실질적으로 열거 타입 상수를 값으로 매핑
하는 일을 한다.

```
<code />
Map<Plant.LifeCycle, Set<Plant>> plantsByLifeCycle =
    new EnumMap<>(Plant.LifeCycle.class); // EnumMap

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);
```

아이템 37 ordinal 인덱싱 대신 EnumMap을 사용하라

EnumMap이란?

EnumMap은 열거 타입을 키로 가지며,
배열과 마찬가지로
실질적으로 열거 타입 상수를 값으로 매핑
하는 일을 한다.



1. 안전하지 않은 형변환을 쓰지 않는다.
2. 맵의 키인 열거 타입이 그 자체로 출력용 문자열을 제공하니 출력 결과에 직접 레이블을 달 필요 없다.
3. EnumMap 내부 구현은 배열 방식이므로 성능이 동일하다.

```
<code />
Map<Plant.LifeCycle, Set<Plant>> plantsByLifeCycle =
    new EnumMap<>(Plant.LifeCycle.class); // EnumMap

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);
```


아이템 37 ordinal 인덱싱 대신 EnumMap을 사용하라

EnumMap + Stream 사용

```
<code />  
  
System.out.println(Arrays.stream(garden)  
    .collect(groupingBy(p → p.lifeCycle,  
        () → new EnumMap<>(LifeCycle.class), toSet())));  
  
}
```

스트림 : 해당 생애주기에 속하는 식물이 있을 때만 만듦
EnumMap : 언제나 생애주기당 하나씩 중첩 맵을 만듦

아이템 37 ordinal 인덱싱 대신 EnumMap을 사용하라

두 가지 상태를 전이와 매핑

```
public enum Phase {  
    SOLID, LIQUID, GAS;  
    public enum Transition {  
        MELT, FREEZE, BOIL, CONDENSE, SUBLIME, DEPOSIT;  
  
        private static final Transition[][] TRANSITIONS = {  
            {null, MELT, SUBLIME },  
            {FREEZE, null, BOIL},  
            {DEPOSIT, CONDENSE, null}  
        };  
  
        // 한 상태에서 다른 상태로의 전이를 반환  
        public static Transition from(Phase from, Phase to){  
            return TRANSITIONS[from.ordinal()][to.ordinal()];  
        }  
    }  
}
```

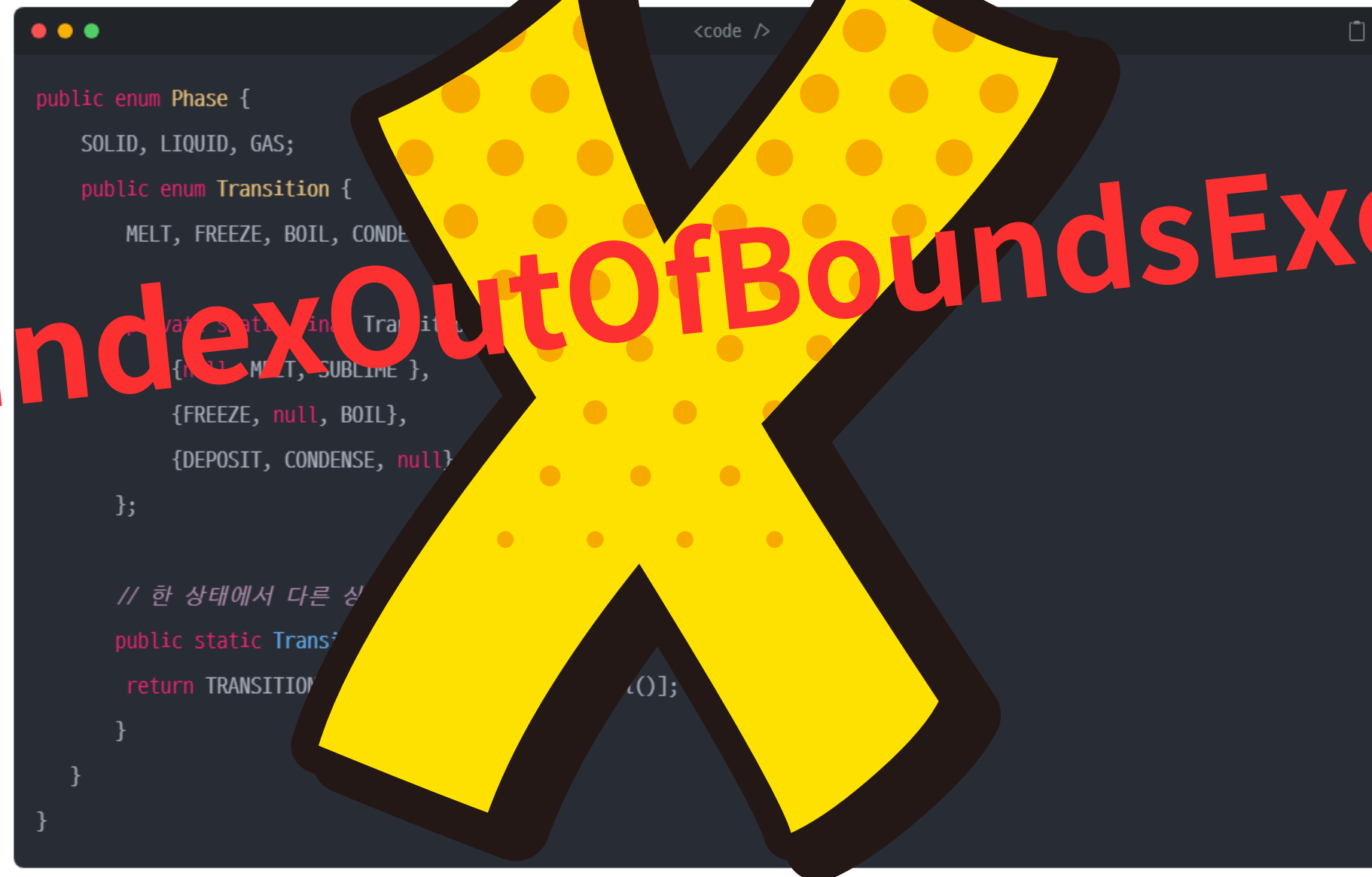
아이템 37 ordinal 인덱싱 대신 EnumMap을 사용하라

중첩 EnumMap : 다차원 관계 표현



아이템 37 ordinal 인덱싱 대신 EnumMap을 사용하라

ArrayIndexOutOfBoundsException



```
<code />

public enum Phase {
    SOLID, LIQUID, GAS;

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

        private static final Transition[] TRANSITIONS = {
            null, MELT, SUBLIME },
            {FREEZE, null, BOIL},
            {DEPOSIT, CONDENSE, null}
        };

        // 한 상태에서 다른 상태로
        public static Transition transition(Phase current, Phase next) {
            return TRANSITIONS[current.ordinal() * TRANSITIONS.length + next.ordinal()];
        }
    }
}
```

아이템 37 ordinal 인덱싱 대신 EnumMap을 사용하라



아이템 37 ordinal 인덱싱 대신 EnumMap을 사용하라

중첩 EnumMap : 다차원 관계 표현

```
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;  
        }  
    }  
}
```

// 상전이 맵을 초기화

```
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);  
}  
}
```

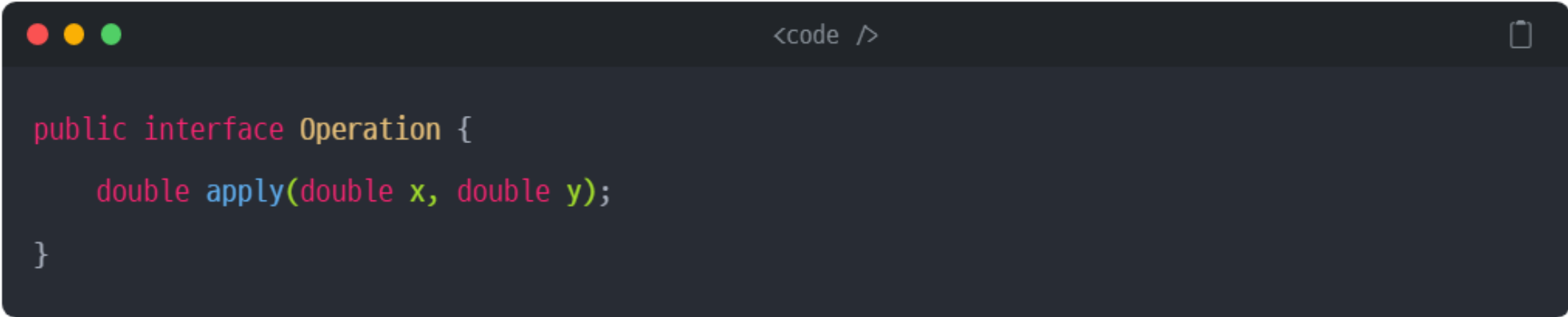
아이템 37 ordinal 인덱싱 대신 EnumMap을 사용하라

NOTE 결론

배열의 인덱스를 얻기 위해 ordinal을 쓰는 것은 일반적으로 좋지 않으니, 대신 EnumMap을 사용하라. 다차원 관계는 EnumMap<..., EnumMap<...>> 으로 표현하라.

아이템 38 확장할 수 있는 열거 타입이 필요하다면 인터페이스를 사용하라

인터페이스를 통한 열거 타입 확장

A code editor window with a dark background. The title bar shows three colored circles (red, yellow, green) on the left and a copy icon on the right. The text inside the editor is:

```
<code />

public interface Operation {
    double apply(double x, double y);
}
```

열거 타입을 확장하는 것은 좋지 못한 선택..

아이템 38 확장할 수 있는 열거 타입이 필요하다면 인터페이스를 사용하라

인터페이스를 통한 열거 타입 확장

```
public interface Operation {  
    double apply(double x, double y);  
}
```

연산 코드에는 쓰임새 ○

```
public enum BasicOperation implements Operation {  
    PLUS("+") {  
        public double apply(double x, double y) { return x + y; }  
    },  
    MINUS("-") {  
        public double apply(double x, double y) { return x - y; }  
    },  
    TIMES("*") {  
        public double apply(double x, double y) { return x * y; }  
    },  
    DIVIDE("/") {  
        public double apply(double x, double y) { return x / y; }  
    };  
  
    private final String symbol;
```

아이템 38 확장할 수 있는 열거 타입이 필요하다면 인터페이스를 사용하라

연산 타입을 확장한 예시 ➡

```
public enum ExtendedOperation implements Operation {  
    EXP("^") {  
        public double apply(double x, double y) {  
            return Math.pow(x, y);  
        }  
    },  
    REMAINDER("%") {  
        public double apply(double x, double y) {  
            return x % y;  
        }  
    };  
    private final String symbol;  
    ExtendedOperation(String symbol) {  
        this.symbol = symbol;  
    }  
    @Override public String toString() {  
        return symbol;  
    }  
}
```

아이템 38 확장할 수 있는 열거 타입이 필요하다면 인터페이스를 사용하라

열거 타입끼리 구현을 상속할 수 없다는 사소한 문제가 존재.

- ➡ 아무 상태에도 의존하지 않는 경우에는 디폴트 구현(아이템 20)을 이용해 인터페이스에 추가
- ➡ 공유하는 기능이 많아 중복량이 높다면, 그 부분을 별도의 도우미 클래스나 정적 도우미 메서드로 분리

아이템 38 확장할 수 있는 열거 타입이 필요하다면 인터페이스를 사용하라

NOTE 결론

열거 타입 자체는 확장할 수 없지만, 인터페이스와 그 인터페이스를 구현하는 기본 열거 타입을 함께 사용해 같은 효과를 낼 수 있다. 이렇게 하면 클라이언트는 이 인터페이스를 구현해 자신만의 혹은 다른 열거 타입을 생성할 수 있다.