

Effective Java

Item 34 : int 상수 대신 열거 타입을 사용하라

Item 35 : original 메서드 대신 인스턴스 필드를 사용하라

Item 36 : 비트 필드 대신 EnumSet을 사용하라

Item 34

int 상수 대신 열거 타입을 사용하라

가장 단순한 열거 타입

```
클래스 public enum Operation {  
    PLUS, MINUS, TIMES, DIVIDE  
    public static final 필드(싱글턴)  
}
```

데이터와 메서드를 갖는 열거 타입

멤버변수

가능하면 private으로

생성자

메서드

```
public enum Speaker {  
    PebbleV2(43_900, 8), 생성자 데이터  
    PebbleV3(64_900, 8),  
    T60(129_000, 30);
```

```
    private final int price;  
    private final int outputWatt;  
    private static final String priceUnit = "KRW";
```

```
    Speaker(int price, int outputWatt) {  
        this.price = price;  
        this.outputWatt = outputWatt;  
    }
```

```
    public int getOutputWatt() {return outputWatt;}  
  
    public double getPriceToDollar() {  
        return (double) price / 1100;  
    }
```

```
}
```

* 주의 : enum은 싱글톤이다.

```
public static void main(String[] args) {  
    Speaker speaker1 = Speaker.PebbleV2;  
    System.out.println(speaker1.getPrice());  
  
    speaker1.changePrice(2);  
  
    Speaker speaker2 = Speaker.PebbleV2;  
    System.out.println(speaker2.getPrice());  
}
```

43900

2

가능하면 멤버 변수는 **private final**로 선언하자

상수별 메서드 구현

멤버변수와 메서드

```
public enum Operation {  
    PLUS("+") {  
        public double apply(double x, double y) {  
            return x + y;  
        }  
    },  
    MINUS("-") {  
        public double apply(double x, double y) {  
            return x - y;  
        }  
    }  
};
```

```
private final String symbol;  
public abstract double apply(double x, double y);
```

```
Operation(String symbol) {  
    this.symbol = symbol;  
}
```

```
@Override  
public String toString() {  
    return symbol;  
}
```

```
}
```

abstract로 선언 안하고
@Override해도 기능은 동작

fromString 구현

```
public static Optional<Operation> fromString(String symbol) {
    return Optional.ofNullable(stringToEnum.get(symbol));
}
```

```
public enum Operation {
    ...
    private static final Map<String, Operation> map
        = new HashMap<>();

    Operation(String symbol){
        this.symbol = symbol;
        map.put(symbol, this);
    }
}
```

enum 생성자 내에서
static 필드 접근이 불가능(∴ 아직 초기화 전)



```
public enum Operation {
    ...
    private static final Map<String, Operation> testMap =
        new HashMap<>();
    static {
        testMap.put("+", Operation.PLUS);
        testMap.put("-", Operation.MINUS);
        testMap.put("*", Operation.TIMES);
        testMap.put("/", Operation.DIVIDE);
    }
}
```



```
public enum Operation {
    ...
    private static final Map<String, Operation> testMap =
        Map.of("+", Operation.PLUS,
              "-", Operation.MINUS,
              "*", Operation.TIMES,
              "/", Operation.DIVIDE);
}
```



```
public enum Operation {
    ...
    private static final Map<String, Operation> stringToEnum =
        Stream.of(values()).collect(
            toMap(Object::toString, e -> e));
}
```



전략 열거 타입 패턴

private 열거 타입

```
public enum PayrollDay {  
    MONDAY(PayType.WEEKDAY), SUNDAY(PayType.WEEKEND);  
  
    private final PayType payType;  
    PayrollDay(PayType payType) { this.payType = payType; }  
  
    int pay(int minutesWorked, int payRate) {  
        return payType.pay(minutesWorked, payRate);  
    }  
}
```

switch 문이나 상수별 메서드 구현보다 안전 및 유연

```
private enum PayType {  
    WEEKDAY {  
        int overtimePay(int minutesWorked, int payRate) {  
            // return do something  
        }  
    },  
    WEEKEND {  
        int overtimePay(int minutesWorked, int payRate) {  
            // return do something  
        }  
    };  
    abstract int overtimePay(int minutesWorked, int payRate);  
    int pay(int minutesWorked, int payRate) {  
        // do calculation  
    }  
}
```


Item 35

ordinal 메서드 대신 인스턴스 필드를 사용하라

Enum 메서드 종류


메서드	설명	리턴 타입
name()	enum 객체의 문자열 리턴	String
ordinal()	enum 객체의 순번(0부터 시작)을 리턴	int
compareTo()	enum 객체를 비교해서 순번 차이를 리턴	int
valueOf(String name)	문자열을 입력받아 일치하는 enum 객체를 리턴	enum
values()	모든 enum 객체들을 배열로 리턴	enum[]

Enum 메서드 종류

```
public enum Week {  
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;  
  
    public int numberOfWeek() {  
        return ordinal() + 1;  
    }  
}
```



```
public enum Week {  
    MONDAY(1), TUESDAY(2), WEDNESDAY(3),  
    THURSDAY(4), FRIDAY(5), SATURDAY(6), SUNDAY(7);  
  
    Week(int number) { this.number = number; }  
    private final int number;  
    public int numberOfWeek() { return number; }  
}
```



Item 36

비트 필드 대신 EnumSet을 사용하라

실제 사례

<https://github.com/you-can-cook/Gobong/blob/master/backend/src/main/java/org/youcancook/gobong/domain/recipe/entity/Cookware.java>

```
@Entity
@NoArgsConstructor(access = AccessLevel.PROTECTED)
public class Cook {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private long cookware;
}
```

개선 방안 두 가지

1. 테이블을 분리한다.
2. @Convert

```
public enum Cookware {
    MICROWAVE(0), AIR_FRYER(1), OVEN(2), GAS_RANGE(3),
    MIXER(4), ELECTRIC_KETTLE(5), PAN(6);

    private final long value;
    Cookware(int power) {
        this.value = 1L << power;
    }

    public long getValue() { return value; }

    public static List<String> bitToList(long cookwaresBit) {
        return Stream.of(Cookware.values())
            .filter(cookware -> (cookware.value & cookwaresBit) != 0)
            .map(Enum::name)
            .collect(Collectors.toList());
    }

    public static long namesToCookwareBit(List<String> cookwares) {
        return cookwares.stream()
            .map(Cookware::valueOf)
            .mapToLong(Cookware::getValue)
            .reduce(0L, Cookware::or);
    }

    private static Long or(Long a, Long b) { return a | b; }
}
```

비트 필드 대신 EnumSet 사용

```
public class Text {  
    public static final int STYLE_BOLD          = 1 << 0; // 1  
    public static final int STYLE_ITALIC        = 1 << 1; // 2  
    public static final int STYLE_UNDERLINE      = 1 << 2; // 4  
    public static final int STYLE_STRIKETHROUGH = 1 << 3; // 8  
  
    // 매개변수 styles는 0개 이상의 STYLE_ 상수를 비트별 OR한 값이다.  
    public void applyStyles(int styles) { ... }  
}  
text.applyStyles(STYLE_BOLD | STYLE_ITALIC);
```



가능하면 인터페이스로 받기

```
public class Text {  
    public enum Style { BOLD, ITALIC, UNDERLINE, STRIKETHROUGH }  
    public void applyStyles(Set<Style> styles) { ... }  
}  
text.applyStyles(EnumSet.of(Style.BOLD, Style.ITALIC));
```

