



# 이펙티브 자바 CP.5

🕒 작성 일시	@2023년 2월 6일 오후 8:44
🕒 최종 편집 일시	@2023년 2월 8일 오후 11:22
📄 유형	이펙티브 자바
👤 작성자	
👥 참석자	

## 5 열거 타입과 애너테이션

선행 내용 (Inpa님 블로그)

- 34. `int` 상수 대신 열거 타입을 사용하라
- 35. `ordinal` 메서드 대신 인스턴스 필드를 사용하라
- 36. 비트 필드 대신 `EnumSet` 을 사용하라
- 37. `ordinal` 인덱싱 대신 `EnumMap` 을 사용하라
- 38. 확장할 수 있는 열거 타입이 필요하다면 인터페이스를 사용하라
- 39. 명명 패턴보다 애너테이션을 사용하라
- 40. `@Override` 애너테이션을 일관되게 사용하라
- 41. 정의하려는 것이 타입이면 마커 인터페이스를 사용하라

## 5 열거 타입과 애너테이션

### ▼ 선행 내용 (Inpa님 블로그)

- [\[JAVA\] 열거형Enum-타입-문법-활용-정리](#)

### ▼ 34. `int` 상수 대신 열거 타입을 사용하라

- 열거 타입 (`Enum`)
  - 일정 개수의 상수 값을 정의한 다음, 그 외의 값은 허용하지 않는 타입이다.
  - 자바가 열거 타입을 지원하기 전에는 정수 열거 패턴을 사용했다.
  - ex ) 정수 열거 패턴

```
public static final int APPLE_FUJI      = 0;
public static final int APPLE_PIPPIN    = 1;
public static final int APPLE_GRANNY_SMITH = 2;

public static final int ORANGE_NAVEL    = 0;
public static final int ORANGE_TEMPLE    = 1;
public static final int ORANGE_BLOOD    = 2;
```

## ■ 단점

- 타입 안전을 보장할 방법이 없으며, 표현력도 좋지 않다.
- 오렌지와 사과를 사용해 동등 비교를 하더라도 컴파일러는 경고를 출력하지 않는다.
- 상수 이름 충돌을 막기 위해 접두어 (prefix) 를 사용한다.
- 프로그램이 깨지기 쉽다.
  - 평범한 상수를 나열한 것 뿐이라, 컴파일시 그 값이 클라이언트 파일에 그대로 새겨진다. 따라서 상수 값이 변경되면 클라이언트도 반드시 다시 컴파일해야 한다.
- 정수 상수는 문자열로 사용하기에 까다롭다.
  - 값을 출력해도, 숫자로만 보여서 도움이 되지 않는다.
  - 정수 대신 문자열 상수를 사용하는 변형 패턴이 있다.
    - 상수(문자열)의 의미를 출력할 수는 있지만, 문자열 값을 그대로 하드 코딩하게 되고, 오타가 있어도 컴파일러는 확인할 방법이 없어, 런타임 버그가 생긴다. 문자열 비교로 당연히 성능 저하가 된다.

## ◦ ex) 열거 타입

```
public enum Apple { FUJI, PIPPIN, GRANNY_SMITH }  
public enum Orange { NAVEL, TEMPLE, BLOOD }
```

## ■ 장점

- 완전한 형태의 클래스라서, 다른 언어 (C, C++, C#)의 열거 타입보다 훨씬 강력함
- 상수 하나당 자신의 인스턴스를 하나씩 만들어 `public static final` 필드로 공개한다. 열거 타입은 밖에서 접근할 수 있는 생성자를 제공하지 않음으로 사실상 `final` 이다.
- 인스턴스들은 오직 하나만 존재한다. (싱글톤 일반화 형태)
- 열거 타입은 컴파일 타임에서 타입 안전성을 제공한다.
  - 위 예제의 `Apple` 열거 타입을 매개변수로 받는 메서드를 선언했다면, 건네 받은 참조는 `Apple` 의 열거 값 중에 하나임을 확실합니다. 다른 타입의 값을 넘기려면 컴파일 오류가 발생한다.
- 열거 타입에는 각자의 이름공간이 있어서 이름이 같은 상수도 공존 가능하다. 정수 패턴과 달리 상수 값이 클라이언트로 컴파일되어 각인되지 않기 때문이다.
- 열거 타입의 `toString` 메서드는 출력하기 적합한 문자열을 제공

- 임의의 메서드나 필드를 추가할 수 있고, 임의의 인터페이스를 구현하게 할 수 있습니다.
  - `Object` 메서드들은 잘 구현해왔고, `Comparable`, `Serializable` 도 구현되어있다.
- ex) 메서드나 필드를 추가할 필요를 근거하는 태양계 행성 클래스

```
public enum Planet {
    MERCURY(3.302e+23, 2.439e6),
    VENUS(4.869e+24, 6.052e6),
    EARTH(5.975e+24, 6.378e6),
    MARS(6.419e+23, 3.393e6),
    JUPITER(1.899e+27, 7.149e7),
    SATURN(5.685e+26, 6.027e7),
    URAUS(8.683e+25, 2.556e7),
    NEPTUNE(1.024e+26, 2.477e7);

    private final double mass;
    private final double radius;
    //표면중력
    private final double surfaceGravity;

    //중력상수 (단위: m^3 / kg s^2)
    private static final double G = 6.67300E-11;

    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
        this.surfaceGravity = G * mass / (radius * radius);
    }

    public double mass() {
        return mass;
    }

    public double radius() {
        return radius;
    }

    public double surfaceGravity() {
        return surfaceGravity;
    }

    public double surfaceWeight(double mass) {
        return mass * surfaceGravity;
    }
}
```

- 열거 타입 상수 각각을 특정 데이터와 연결지으려면 생성자에서 데이터를 받아 인스턴스 필드에 저장하면 된다.
    - 열거 타입은 근본적으로 불변이라 모든 필드를 `final` 이어야 한다 (아이템 17)
    - 필드를 `public` 으로 선언해도 되지만, `private` 으로 두고 별도의 `public` 접근자 메서드를 두는게 낫다 (아이템 16)
- ex) 행성 클래스를 사용하는 클라이언트 코드

```

public class WeightTable {
    public static void main(String []args) {
        double earthWeight = Double.parseDouble(args[0]);
        double mass = earthWeight / Planet.EARTH.surfaceGravity();
        for(Planet p : Planet.values()) {
            System.out.printf("%s에서의 무게는 %f이다.%n", p, p.surfaceWeight(mass));
        }
    }
}

```

- **values** 메서드 자신 안의 정의된 상수들의 배열(순서를 보장)에 담아 반환하는 정적 메서드
- 만약, 여기서 EARTH 라는 열거 타입이 없어진다면, 컴파일 시 해당 상수를 사용하는 줄에 컴파일 오류를 뿜을 것이다.  
클라이언트를 다시 컴파일 하지 않으면 런타임에, 정수 열거 패턴에서는 기대할 수 없는 가장 바람직한 대응이라 본다.
- 열거 타입을 선언한 클래스 혹은 그 패키지에서만 유용한 기능은 **private** 나 **package-private** 메서드로 구현해라 (아이템 15)
- 널리 쓰이는 열거 타입은 톱레벨 클래스로 만들고, 특정 톱 레벨 클래스에서만 쓰인다면 해당 클래스의 멤버 클래스(아이템 24) 로 만든다.
- 더 다양한 기능을 하는 열거 타입
  - ex) 상수 마다 동작이 달라야 하는 상황 (상수별 메서드 적용 X)

```

public enum Operation {
    PLUS, MINUS, TIMES, DIVIDE;

    public double apply(double x, double y) {
        switch(this) {
            case PLUS: return x + y;
            case MINUS: return x - y;
            case TIMES: return x * y;
            case DIVIDE: return x / y;
        }
        throw new AssertionError("알 수 없는 연산: " + this);
    }
}

```

- throw 문에 실제로는 도달할 일이 없지만, 기술적으로는 도달할 수 있기 때문에, 생략하면 컴파일 조차 되지 않는다.
- 상수를 추가 후 반드시 apply 메서드에 해당 case를 추가해줘야만 한다.  
즉, 깨지기 쉬운 코드이다.
- ex) 상수 마다 동작이 달라야 하는 상황 (상수별 메서드 적용 O)

```

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; }},
    TIMES { public double apply(double x, double y) { return x * y; }},
    DIVIDE { public double apply(double x, double y) { return x / y; }},

    public abstract double apply(double x, double y);
}

```

- `apply` 라는 추상 메서드를 선언하고 각 상수별 클래스 몸체, 즉 각 상수에서 자신에 맞게 재정의 하는 방법  
이를 상수별 메서드 구현이라고 한다.
- 상수에 `apply` 메서드를 재정의하지 않으면 컴파일 오류가 난다.
- ex) 필드와 메서드를 사용한 열거 타입

```

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;
        }
    },
    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;

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

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

    public abstract double apply(double x, double y);
}

```

- 열거 타입에는 상수 이름을 입력받아 그 이름에 해당하는 상수를 반환해주는 `valueOf(String)` 메서드가 자동 생성된다.  
열거 타입의 `toString` 메서드를 재정의하려거든, `toString` 이 반환하는 문자열을 해당 열거 타입 상수로 변환해주는 `fromString` 메서드도 함께 제공하는 걸 고려해보자.

- ex) 열거 타입용 `fromString` 메서드 구현

```
private static final Map<String, Operation> stringToEnum =
    Stream.of(values()).collect(Collectors.toMap(Object::toString, e -> e));
// 지정한 문자열에 해당하는 Operation을 존재한다면 반환한다.
public static Optional<Operation> fromString(String symbol) {
    return Optional.ofNullable(stringToEnum.get(symbol));
}
```

- `Operation` 상수가 `stringToEnum` 맵에 추가되는 시점은 열거 타입 상수 생성 후 정적 필드가 초기화될 때다.
- 열거 타입의 정적 필드 중 열거 타입의 생성자에서 접근할 수 있는 것은 상수 변수뿐이다. (아이템 24)
- 열거 타입 생성자가 실행되는 시점에는 정적 필드들이 아직 초기화되기 전이라, 자기 자신을 추가하지 못하게 하는 제약이 꼭 필요하다.  
특수한 예로, 열거 타입 생성자에서 같은 열거 타입의 다른 상수에도 접근할 수 없다.
- `fromString` 이 `Optional<Operation>` 을 반환하는 점도 주의하자.  
주어진 문자열이 가리키는 연산이 존재하지 않을 수 있음을 클라이언트에 알리고, 그 상황을 클라이언트에서 대처하도록 한 것이다.
- 상수별 메서드 구현은 열거 타입 상수끼리 코드 공유가 어렵다.
  - ex) 값에 따라 분기하여 코드를 공유하는 열거 타입

```
public enum PayrollDay {
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY;

    private static final int MINS_PER_SHIFT = 8 * 60;

    int pay(int minutesWorked, int payRate) {
        int basePay = minutesWorked * payRate;
        int overtimePay;
        switch (this) {
            //주말
            case SATURDAY:
            case SUNDAY:
                overtimePay = basePay / 2;
                break;
            //주중
            default:
                overtimePay = minutesWorked <= MINS_PER_SHIFT ?
                    0 : (minutesWorked - MINS_PER_SHIFT) * payRate / 2;
        }

        return basePay + overtimePay;
    }
}
```

```
}
}
```

- 깔끔하지만, 유지보수 측면에서 위험하다. 휴가와 같은 새로운 값을 열거 타입에 추가하려면 case 문을 잊지 말고 쌍으로 넣어줘야한다.
- 상수별 메소드 구현으로 급여를 계산하는 방법
  1. 잔업 수당을 계산하는 코드를 모든 상수에 중복해서 넣는다.
  2. 계산 코드를 평일용과 주말용으로 나눠 각각을 도우미 메서드로 작성후 각 상수가 자신에게 필요한 메서드를 적절히 호출한다.
    - 하지만 위 두 방식 모두 코드가 장황해져 가독성이 떨어지고 오류 가능성이 올라간다.
  3. 평일 잔업수당 계산용 메서드를 구현하고, 주말 상수에만 재정의해 쓴다.
    - 장황한 코드는 줄어들지만, switch 문을 썼을 때와 같은 단점이 드러난다. 새로운 상수를 추가하면서 계산용 메서드를 재정의하지 않으면, 그대로 물려받을 가능성이 있다.
  4. 가장 깔끔한 방법으로 **전략(전략 열거 타입 패턴)**을 선택하도록 하는 것이다.
    - ex) 전략 열거 타입 패턴

```
enum PayrollDay {
    MONDAY(WEEKDAY), TUESDAY(WEEKDAY), WEDNESDAY(WEEKDAY),
    THURSDAY(WEEKDAY), FRIDAY(WEEKDAY),
    SATURDAY(WEEKEND), SUNDAY(WEEKEND);

    private final PayType payType;

    PayrollDay(PayType payType) { this.payType = payType; }

    int pay(int minutesWorked, int payRate) {
        return payType.pay(minutesWorked, payRate);
    }

    // 전략 열거 타입
    enum PayType {
        WEEKDAY {
            int overtimePay(int minsWorked, int payRate) {
                return minsWorked <= MINS_PER_SHIFT ? 0 :
                    (minsWorked - MINS_PER_SHIFT) * payRate / 2;
            }
        },
        WEEKEND {
            int overtimePay(int minsWorked, int payRate) {
                return minsWorked * payRate / 2;
            }
        }
    };

    abstract int overtimePay(int mins, int payRate);
    private static final int MINS_PER_SHIFT = 8 * 60;
}
```

```
int pay(int minsWorked, int payRate) {
    int basePay = minsWorked * payRate;
    return basePay + overtimePay(minsWorked, payRate);
}
}
```

- 잔업 수당 계산을 `private` 중첩 열거 타입(`PayType`)으로 옮기고 `PayrollDay` 열거 타입의 생성자에서 이중 적당한 것을 선택한다. `PayrollDay` 열거 타입은 잔업수당 계산을 그 전략 열거 타입에 위임하여, `PayrollDay` 열거 타입에서 `switch` 문이나 상수별 메서드 구현이 필요 없게 된다.

#### ◦ 열거 타입에서 `switch` 문

- 코드가 예쁘지 않고, 깨지기 쉽다.
- 새로운 상수를 추가하면 `case` 문도 추가해야 하고, 혹시라도 깜빡한다면 제대로 동작하지 않게 된다.
- 이러한 단점을 제외하고 **기존 열거 타입에 상수별 동작을 혼합해 넣을 때는 `switch` 문이 좋은 선택이 될 수 있다.**
- ex) 원래 열거 타입에 없는 기능을 수행한다.

```
public static Operation inverse(Operation op) {
    switch(op) {
        case PLUS: return Operation.MINUS;
        case MINUS: return Operation.PLUS;
        case TIMES: return Operation.DIVIDE;
        case DIVIDE: return Operation.TIMES;

        default: throw new AssertionError("알 수 없는 연산: " + op);
    }
}
```

#### ◦ 열거 타입을 사용할 때

- **필요한 원소를 컴파일 타임에 다 알 수 있는 상수 집합이라면 항상 열거 타입을 사용하자**
  - ex) 태양계 행성, 한 주의 요일, 메뉴 아이템, 연산 코드, 명령줄 플래그 등
- **열거 타입에 정의된 상수 개수가 영원히 고정 불변일 필요는 없다.**

#### • 정리

- 열거 타입은 확실히 정수 상수보다 뛰어나다.
  - 더 읽기 쉽고 안전하고 강력하다.
- 대다수 열거 타입이 명시적 생성자나 메서드 없이 사용되지만, 각 상수를 특정 데이터와 연결짓거나 상수마다 다르게 동작하게 할 때는 필요하다.



- 하나의 메서드가 상수별로 다르게 동작해야 할 때는, `switch` 문 대신 **상수별 메서드 구현**을 사용하자.
- 열거 타입 상수 일부가 같은 동작을 공유한다면, **전략 열거 타입 패턴**을 사용하자.

- ▼ 35. `ordinal` 메서드 대신 **인스턴스 필드**를 사용하라
- ▼ 36. 비트 필드 대신 `EnumSet` 을 사용하라
- ▼ 37. `ordinal` 인덱싱 대신 `EnumMap` 을 사용하라
- ▼ 38. 확장할 수 있는 열거 타입이 필요하다면 **인터페이스**를 사용하라
- ▼ 39. 명명 패턴보다 **애너테이션**을 사용하라
- ▼ 40. `@Override` 애너테이션을 일관되게 사용하라
- ▼ 41. 정의하려는 것이 타입이면 **마커 인터페이스**를 사용하라