

# 이펙티브 자바 CP.5

① 작성 일시	@2023년 2월 6일 오후 7:44
① 최종 편집 일시	@2023년 2월 11일 오후 7:55
① 유형	이펙티브 자바
① 작성자	
① 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_FUJII      = 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 수 는 있지만, 문자열 값을 그대로 하드코딩하게 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, 건네받은 참조는 `Apple`의 열거 값 중에 하나임을 확실합니다. 다른 타입의 값을 넘기려면 컴파일 오류가 발생합니다.
- 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 구현해뒀고, `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 토큰 클래스로 만들고, 특정 토큰 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, 생각하면 컴파일 조차 되지 않는다.
- 상수를 추가 후 반드시 `apply` 메서드에 해당 `case`를 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 방법을 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 = 0;
        }
        return basePay + overtimePay;
    }
}

```

```

        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 필드를 사용하라

- 대부분의 열거 타입 상수는 자연스럽게 하나의 정숫값에 대응한다.
- 그리고 모든 열거 타입은 해당 상수가 그 열거 타입에서 몇 번째 위치인지를 반환하는 `ordinal`이라는 메서드를 제공한다.
- ex) `ordinal`을 잘못 사용

```
public enum Ensemble {
    SOLO, DUET, TRIO, QUARTET, QUINTET, SEXTET, SEPTET, OCTET, NONET, DECTET;

    public int numberOfMusicians() {
        return ordinal() + 1;
    }
}
```

- 상수 선언 순서를 바꾸는 순간 `numberOfMusicians`가 오작동한다.
- 이미 사용중인 정수와 값이 같은 상수는 추가할 방법이 없다. 또한 중간에 값을 비워둘 수 없다.
- **열거 타입 상수에 연결된 값은 `ordinal` 메서드로 얻지 말고 인스턴스 필드에 저장하자.**
- ex) 인스턴스 필드로 사용한 경우

```
public enum Ensemble {
    SOLO(1), DUET(2), TRIO(3), QUARTET(4), QUINTET(5), SEXTET(6),
    SEPTET(7), OCTET(8), DOUBLE_QUARTET(8), NONET(9), DECTET(10),
    TRIPLE_QUARTET(12);

    private final int numberOfMusicians;

    Ensemble(int numberOfMusicians) {
        this.numberOfMusicians = numberOfMusicians;
    }

    public int numberOfMusicians() {
        return numberOfMusicians;
    }
}
```

- Enum API 문서

“대부분의 프로그래머는 이 메서드를 쓸 일이 없다. 이 메서드는 EnumSet과 EnumMap 같이 열거 타입 기반의 범용 자료구조에 쓸 목적으로 설계되었다.”

### ▼ 36. 비트 필드 대신 EnumSet 을 사용하라

- ex) 비트 필드 열거 상수

```
public class Text {
    public static final int STYLE_BOLD      = 1 << 0;
    public static final int STYLE_ITALIC    = 1 << 1;
    public static final int STYLE_UNDERLINE = 1 << 2;
    public static final int STYLE_STRIKETHROUGH = 1 << 3;

    // 매개변수 styles는 0개 이상의 STYLE_ 상수 비트별 OR 한 값이다.
    public void applyStyles(int styles) { ... }
}
```

- 다음과 같은 식으로 비트별 OR 를 사용해 여러 상수를 하나의 집합으로 모을 수 있으며, 이렇게 만들어진 집합을 비트 필드 라고 한다.
- `text.applyStyle(STYLE_BOLD | STYLE_ITALIC);`
- 비트 필드를 사용하면 비트별 연산을 사용해 합집합과 교집합 같은 집합 연산을 효율적으로 수행할 수 있다.
- 하지만 비트 필드는 정수 열거 상수의 단점을 그대로 지니며, 추가로 다음 문제까지 안고 있다.
  - 비트 필드 값이 그대로 출력되면 단순한 정수 열거 상수를 출력할 때보다 해석하기가 훨씬 어렵다.
  - 비트 필드 하나에 녹아 있는 모든 원소를 순회하기도 까다롭다.
  - 최대 몇 비트가 필요한지를 API 작성 시 미리 예측하여 적절한 타입(`int`, `long`)을 선택해야한다. API를 수정하지 않고는 비트 수(32 bit or 64 bit)를 더 늘릴 수 없기 때문임.
- 더 나은 대안 EnumSet 클래스

```
public class Text {
    public enum Style { STYLE_BOLD, STYLE_ITALIC, STYLE_UNDERLINE, STYLE_STRIKETHROUGH };

    // 어떤 Set을 넘겨도 되나, EnumSet 이 가장 좋다.
    public void applyStyles(int styles) { ... }
}
```

- 열거 타입 상수의 값으로 구성된 집합을 효과적으로 표현해준다.
- Set 인터페이스를 완벽히 구현하며, 타입 안전하고, 다른 어떤 Set 구현체와도 사용할 수 있다.
- 내부는 비트 벡터로 구현되어 있다.  
원소가 총 64개 이하라면, 대부분의 경우에 EnumSet 전체를 long 변수 하나로 표현하여 비트 필드에 비슷한 성능을 보여준다.
- removeAll 과 retainAll 같은 대량 작업은 비트를 효율적으로 처리할 수 있는 산술 연산으로 구현했다.
- 비트를 직접 다룰 때 흔히 겪는 오류들에서 해방된다. (어떤 경우인지 머리에 안 그려짐)
- EnumSet 의 집합 생성 등 다양한 기능의 정적 팩토리를 제공한다.
  - `text.applyStyles(EnumSet.of(Style.BOLD, Style.ITALIC));`
    - 집합 생성 of 메서드
- 정리
  - 열거할 수 있는 타입을 한데 모아 집합 형태로 사용한다고 해도 비트 필드를 사용할 이유는 없다.
  - EnumSet 클래스가 비트 필드 수준의 명료함과 성능을 제공하고 열거 타입의 장점을 얻을 수 있다.
  - 단점으로는 불변 EnumSet 을 만들 수 없다는 것이다.
    - 불변으로 만들자한다면, Collections.unmodifiableSet 으로 EnumSet 을 감싸 사용하자.

### ▼ 37. ordinal 인덱싱 대신 EnumMap 을 사용하라

- ex) 식물을 간단히 나타낸 클래스

```
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;
    }
}
```

- ex) ordinal() 을 배열 인덱스로 사용 - **사용 금지!**

```
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) // garden 메서드의 파라미터
    plantsByLifeCycle[p.lifeCycle.ordinal()].add(p);

// 결과 출력
for (int i = 0; i < plantsByLifeCycle.length; i++)
    System.out.printf("%s: %s\n", Plant.LifeCycle.values()[i], plantsByLifeCycle[i]);
```

- 배열은 제네릭과 호환되지 않으니 (아이템 28) 비검사 형변환을 수행해야 하고 깔끔히 컴파일되지 않을 것이다. (초기화 줄)
- 배열은 각 인덱스의 의미를 모르니 출력 결과에 직접 레이블을 달아야 한다. (출력 줄)
- **정확한 정숫값을 사용한다는 것을 사용자가 직접 보증해야한다.**
  - 정수는 열거 타입과 달리 타입 안전하지 않기 때문이다.
  - 잘못된 값을 사용하면 잘못된 동작을 수행하거나, `ArrayIndexOutOfBoundsException` 을 던질 것이다.
- 해결책 열거 타입을 키로 사용한 클래스 `EnumMap`
  - 위에서 배열은 실질적으로 열거 상수를 값으로 매핑하는 일로 함으로 `Map` 으로 해당 역할을 대신 사용 할 수 있을 것이다.

```
Map<Plant.LifeCycle, Set<Plant>> plantByLifeCycle = 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);
```

- 더 짧고 명료하고 안전하고 성능도 비슷하다.
  - 성능이 비슷한 이유는 `Map` 은 배열을 사용함
- 안전하지 않은 형변환을 쓰지 않고, 맵의 키인 열거 타입이 그 자체로 출력용 문자열을 제공하니 출력 결과에 직접 레이블을 달 일도 없다.
- 배열의 인덱스를 계산 시 오류가 발생할 가능성도 원천봉쇄된다.
- 내부 구현 방식을 안으로 숨겨서 `Map` 의 타입 안전성과 배열의 성능을 모두 얻었다.
- `EnumMap` 의 생성자가 받는 키 타입의 `Class` 객체는 한정적 타입 토큰으로 런타임 제네릭 타입 정보를 제공한다. (아이템 33)
- `Stream` (아이템 45) 을 사용한 코드

```
System.out.println(Arrays.stream(garden)
    .collect(groupingBy(p -> p.lifeCycle)));
```



- 이 코드에서는 `EnumMap` 이 아닌 고유한 맵 구현체 (jdk 11 기준 - `HashMap`)를 사용했기 때문에, `EnumMap` 을 써서 얻은 공간과 성능 이점이 없어진다.
- `Stream` 사용한 코드 - `EnumMap` 을 명시함.

```
System.out.println(Arrays.stream(garden)
    .collect(groupingBy(p -> p.LifeCycle,
        () -> new EnumMap<>(LifeCycle.class), toSet())));
```

- `Stream` 을 사용하면 `EnumMap` 만 사용했을 때와 살짝 다르게 동작한다.
  - `EnumMap` 은 열거 타입 상수 별로 하나씩 Key를 전부 다 만든다.
  - `Stream` 은 존재하는 열거 타입 상수만 Key로 만든다.
- ex) 더 복잡한 `ordinal()` 을 사용한 상태(`Phase`) 클래스

```
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 상태로 전이를 반환한다.
        public static Transition from(Phase from, Phase to) {
            return TRANSITIONS[from.ordinal()][to.ordinal()];
        }
    }
}
```

- 앞에 `ordinal` 메서드를 사용한 예와 다를 게 없다.
  - 컴파일러는 `ordinal` 과 배열 인덱스의 관계를 알 도리가 없다.
  - 즉, `Phase`, `Phase.Transition` 열거 타입을 수정하면서 `TRANSITIONS` 를 함께 수정하지 않으면 에러가 나거나, 이상하게 동작할 수 있다.
- ex) `EnumMap` 으로 변환한 상태 클래스

```
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;
        }

        // 이전 상태에서 '이후' 상태에서 전이로 맵에 대응시키는 맵
        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))));

        // 한 상태에서 다른 상태로 전이를 반환한다.
        public static Transition from(Phase from, Phase to) {
            return m.get(from).get(to);
        }
    }
}
```

- `groupingBy` 에 선은 전이를 이전 상태를 기준으로 묶는다.
  - 바깥 `Map` 의 `key` 를 지정하고 `EnumMap` 으로 구현체를 지정한다.
  - `groupingBy` 내용 (jdk 11)

```

Returns a Collector implementing a cascaded "group by" operation on input elements of type T, grouping elements according to the classification function. The classification function maps elements to some key type K. The downstream collector operates on elements of type T.

Params:
  classifier - a classifier function mapping input elements to keys
  mapFactory - a function which, when called, produces a new empty Map of the desired type
  downstream - a Collector implementing the downstream reduction

Returns:
  a Collector implementing the cascaded group-by operation

```

- `toMap` 에는 이후 상태를 전이에 대응시키는 `EnumMap` 을 생성한다.
  - 바깥 `Map` 의 `value` 부분으로, 안쪽 `Map` 를 구현한다.
  - `(x, y) -> y` 부분은 사실 `x, y` 중 아무거나 써도 별 상관 없는 듯 하다.
  - `toMap` 내용 (jdk 11)

```

Returns a Collector that accumulates elements into a Map whose keys and values are the result of applying the provided mapping functions. If the mapped keys contains duplicates (according to Object.equals(Object)), the value mapping function is applied to the duplicate elements.

Params:
  keyMapper - a mapping function to produce keys
  valueMapper - a mapping function to produce values
  mergeFunction - a merge function, used to resolve collisions between values associated with the same key, as supplied by mapSupplier
  mapSupplier - a function which returns a new, empty Map into which the results will be inserted

Returns:
  a Collector which collects elements into a Map whose keys are the result of applying a key mapping function to the elements

```

- ex) 새로운 상태가 추가된다 `PLASMA`

```

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);
    }

    //나머지 코드는 그대로
    ...
}

```

- 기체에서 플라스마로 변하는 이온화 (`IONIZE`), 플라스마에서 기체로 변하는 탈이온화 (`DEIONIZE`)
- `EnumMap` 을 사용하지 않은 상태에서는 많은 수정이 들어가야하며, 잘못 수정되면 에러나 이상하게 동작할 것 이다.

- 정리
  - 배열의 인덱스를 얻기 위해 `ordinal` 을 쓰는 것은 일반적으로 좋지 않으니, `EnumMap` 을 사용하자.
  - 다차원 관계는 `EnumMap<..., EnumMap<...>>` 으로 구현해라.

### ▼ 38. 확장할 수 있는 열거 타입이 필요하다면 인터페이스를 사용하라

- 열거 타입은 확장을 할 수 없다. 하지만 인터페이스를 구현해 같은 효과를 낼 수 있다.
- ex) 인터페이스를 구현해 확장 가능 열거 타입을 흉내 낸다. (계산기 관련 클래스)

```

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

```

```

public enum BasicOperation implements Operation {
    PLUS("+") {
        @Override
        public double apply(double x, double y) { return x + y; }
    },
    MINUS("-") {
        @Override
        public double apply(double x, double y) { return x - y; }
    },
    TIMES(" * ") {
        @Override
        public double apply(double x, double y) { return x * y; }
    },
    DIVIDE("/") {
        @Override
        public double apply(double x, double y) { return x / y; }
    };

    private final String symbol;

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

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

```

- 열거 타입인 `BasicOperation` 을 확장 할 수 없지만 인터페이스인 `Operation` 은 확장 할 수있고, 이 인터페이스를 연산의 타입으로 사용하면 된다.
- ex) 확장 가능 열거 타입

```

public enum ExtendedOperation implements Operation {
    EXP("^") {
        @Override
        public double apply(double x, double y) { return Math.pow(x, y); }
    },
    REMAINDER("%") {
        @Override
        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; }
}

```

- 새로 작성한 연산은 기존 연산을 쓰던 곳이면 어디든 사용 가능하다.
- `Operation` 인터페이스를 사용하도록 작성되어 있지만 하면 된다.
- `interface` 를 구현해 사용함으로 다형성을 보장하고, 열거 타입에서 추상 메서드를 선언하지 않아도 된다. 개별 인스턴스 수준에서만 아니라 타입 수준에서도, 기본 열거 타입 대신 확장된 열거 타입을 넘겨 확장된 열거 타입의 원소 모두를 사용하게 할 수도 있다.
- 모든 원소를 테스트 하는 방법
  1. 한정적 타입 토큰 사용

```

public static void main(String[] args) {
    double x = Double.parseDouble(args[0]);
    double y = Double.parseDouble(args[1]);
    test(ExtendedOperation.class, x, y);
}

private static <T extends Enum<T> & Operation> void test(
    Class<T> opEnumType, double x, double y) {
    for (Operation op : opEnumType.getEnumConstants()) {
        System.out.printf("%f %s %f = %f\n", x, op, y, op.apply(x, y));
    }
}

```

- `test` 메서드에 `ExtendedOperation` 의 `class` 리터널을 넘겨 확장된 연산이 무엇인지 알려준다.
- `<T extends Enum<T> & Operation> Class<T>` 는 `Class` 객체가 열거타입인 동시에 `Operation` 을 구현해야 한다는 것이다.
- `getEnumConstants` 메소드는 `Class` 객체에 구현되어 있으며 구현된 메서드를 모두 불러온다.

## 2. 한정적 와일드 카드 타입 사용

```
public static void main(String[] args) {
    double x = Double.parseDouble(args[0]);
    double y = Double.parseDouble(args[1]);
    testV2(Arrays.asList(ExtendedOperation.values()), x, y);
}

private static void testV2(Collection<? extends Operation> opSet, double x, double y) {
    for (Operation op : opSet) {
        System.out.printf("%f %s %f = %f%n", x, op, y, op.apply(x, y));
    }
}
```

- 코드가 좀 덜 복잡하고 `test` 메서드가 좀 더 유연해졌다.  
여러 구현 타입의 연산을 조합해 호출 할 수 있다.  
`Arrays.asList(ExtendedOperation.values(), BasicOperation.values())` 이런식으로

- 특정 연산에서 `EnumMap` 이나 `EnumSet` 을 사용할 수 없다는 단점이 있다.

### • 문제점

- 인터페이스를 이용해 확장 가능한 열거 타입으로 흉내 내는 방식에도 문제가 있다.
- 열거 타입끼리 구현을 상속할 수 없다는 것이다.
  - 아무 상태에도 의존하지 않는 경우 (stateless) 에는 디폴트 메소드를 구현하면 된다.(아이템 20)
  - 반면 `Operation` 처럼 연산 기호 (상태 의존 - stateful)를 저장하고 찾는 로직이 `BasicOperation`, `ExtendedOperation` 에 모두 들어가야만 한다.  
공유하는 기능이 있다면 별도의 도우미 클래스나 정적 도우미 메서드로 분리하는 방식으로 코드 중복을 줄이자.

### • 정리

- 열거 타입 자체는 확장 할 수 없지만, 인터페이스와 그 인터페이스를 구현하는 기본 열거 타입을 함께 사용해 같은 효과를 낼 수 있다.
- 인터페이스를 구현해 자신만의 열거 타입(혹은 다른 타입)을 만들 수 있다.
- API 가 (기본 열거 타입을 직접 명시하지 않고) 인터페이스 기반으로 작성되어있다면, 기본 열거 타입의 인스턴스가 쓰이는 곳을 새로 확장한 열거 타입의 인스턴스로 대체해 사용 할 수 있다. [그냥 리스코프 치환 원칙을 말한 것으로 보인다.]

책에서는 타입 안전 열거 패턴을 언급하나 초반에서 소개하였으나 현재 열거 타입에 `interface` 를 구현해 사용하는 것으로 추천되는 것으로 보인다.

## ▼ 39. 명명 패턴보다 애너테이션을 사용하라

## ▼ 40. `@Override` 애너테이션을 일관되게 사용하라

## ▼ 41. 정의하려는 것이 타입이면 마커 인터페이스를 사용하라