6장 모든 객체의 공통 메서드

분류	서적 이펙티브 자바
시작일자	@2022년 5월 11일 오전 10:33
종료일자	@2022년 5월 14일 오후 2:39
참고	참고 블로그 [자바봄 기술 블로그]
스터디 링크	GitHub Link [이펙티브 자바 스터디]
서적	
Java	

🚀 아이템 34. int 상수 대신 열거 타입을 사용하라.

```
// 과일을 종류별로 상수로 지정해 놓은 Foods 클래스
public class Foods {
    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;
}
```

- 타입의 **안전성 보장이 되지 않는다.**

- 오렌지를 건네는 세머드에 사과를 보내고 동등 연산자로 비교해도 컴파일러는 경고 없이 통과하게 된다.
- 자바는 별도의 namespace를 지원하지 않아서 접두어(APPLE, ORANGE)를 붙혀야 한다.
- 상수를 나열한 것이기 때문에 깨지기 쉽다.

> **컴파일시 그 값이 클라이언트 파일에 그대로 새겨지는데, 상수의 값이 바뀌면 클라이언트도 다시 컴파일 해야 한다.**

```
public void apple(int apple) { // 애플 관련 상수만 들어가야한다.
    ...
}
...
function(APPLE_FUJI); // OK
function(ORANGE_NAVEL); // 오렌지가 들어와서 오류가 생기지 않는다.
```

✅ 열거타입 예시 1

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

```
public void apple(Apple apple) { //
    ...
}
...
function(APPLE_FUJI); // OK
function(ORANGE_NAVEL); // 컴파일 에러 발생
```

- 공개되는 건 필드의 이름뿐이라서 추가하거나 순서를 바꿔도 다시 컴파일하지 않아도 된다.
- 상수를 하나 제거하더라도 제거한 상수를 참조하지 않는 클라이언트에서는 아무 영향이 없다.

  > **제거한 상수를 참조하는 클라이언트가 있으면 컴파일러가 발생하여 즉시 알아차릴 수 있다.**

- 인스턴스를 통제할 수 있다.
  - 클라이언트가 인스턴스를 집접 생성하거나 확장할 수 없다. ***(public 생성자를 제공하지 않기 때문)***
  - 열거 타입 인스턴스들은 **딱 하나만 존재함이 보장**이 된다.

✅ 열거타입 예시 2

```
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 구문 없이는 컴파일되지가 않는다.
    throw new AssertionError("알 수 없는 연산 :" + this);
  }
}
```

> 새로운 Operation 타입이 추가되면 switch 구문의 case가 추가해야 하는데 이 과정에서 실수하기가 쉽다.

```
// 위 단점 개선 코드
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);

    private final String symbol;

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

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

> 열거 타입에 추상메서드를 구현한 것이다.
> 상수-한정 메서드 구현이라고도 하는데 새로운 Opertaion이 추가되더라도 다른 곳에는 영향을 아예 줄 수가 없기에 이전 코드의 아쉬운점을 해결해준다.

```
@Test
@DisplayName("상수-한정 메서드 구현")
void 상수_한정_메서드() {
    double x = 2.0;
    double y = 4.0;

    for (Operation value : Operation.values()) {
        System.out.printf("%f %s %f = %f%n", x, value, y, value.apply(x, y));

    }
}
```

> 단점을 개선했을 뿐만 아니라 가독성 또한 늘렸다.

✅ 열거타입 예시 3

```
/**
 * 1. 업무시간(분) : 8 * 60(1일 8시간 근무 기준)
 * 2. 주중 오버타임은 잔업시간에 추가 된다.
 * 3. 주말에는 무조건 잔업수당이 주어진다.
 */

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) {
            case SATURDAY: case SUNDAY: // 주말인 경우
                overtimePay = basePay / 2;
                break;
            default://주중
                overtimePay = minutesWorked <= MINS_PER_SHIFT
                        ? 0
                        : (minutesWorked - MINS_PER_SHIFT) * payRate / 2;
        }
        return basePay + overtimePay;
    }
}
```

⛔ **문제 1**. 주말과 주중을 구분하여 주중에 오버타임에 대해서는 구분을 해놨지만 그외의 잔업이 발생하는 경우에 대한 고려가 안돼있다.
⛔ **문제 2**. 공휴일/휴가기간 등 여러가지 추가 경우의 수를 매번 case로 추가하거나 조건을 수정해야 하는데 그럼 예시2번의 문제랑 동일해진다.

❓ 어떻게 해야할까? -> 전략 패턴을 사용하자!

```
/**
 * 1. 평일 휴일을 private 중첩 열거 타입으로 만들어 옮기자.
 * 2. PayrollDay 열거타입의 생성자에서 주입받도록 한다.
 */

public 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);
    }
```

```
    enum PayType {
        WEEKDAY {
            @Override
            int overtimePay(int minWorked, int payRate) {
                return minWorked <= MINS_PER_SHIFT
                        ? 0
                        : (minWorked - MINS_PER_SHIFT) * payRate / 2;
            }
        },

        WEEKEND {
            @Override
            int overtimePay(int minWorked, int payRate) {
                return minWorked * payRate / 2;
            }
        };

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

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

❓ 그럼 switch 문은 사용안하는게 좋은것인가?

> 무조건 그런것은 아니다.
> 하지만 보통은 switch를 안쓰고 위처럼 코딩하는편이 실수가 적어지며 컴파일시 오류잡기가 편해진다.

```
// 각각의 타입의 반대 연산을 반환하는 메서드를 구현해야 한다면 적절한 선택이 될 수도 있다.
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);
    }
}
```

📋 정리

- 성능은 정수 상수와 크게 차이가 나지는 않는다.
- 위에서 작성한 사칙연산과 같이 원소를 컴파일 타엠에서 다 알 수 있는 **상수 집합이라면 열거 타입을 사용**하자.

> 실수를 방지하기 위해서 switch문 대신 전략 패턴 사용하는 것을 습관화하자.

🚀 아이템 35. ordianl 메서드 대신 인스턴스 필드를 사용하라.

```
// oridnal 예시 -> 따라하지 말것
// 상수가 몇 번째 위치인지를 반환하는 ordinal메서드
public enum Ensemble {
    SOLO, DUET, TRIO, QUARTET, QUINTET, SEXTET, SEPTET, OCTET, NONET, DECTET;

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

- 연주자가 1명인 솔로부터 10명인 디텍트까지 정의한 열거타입이다.
- `numberOfmusicians()` 를 호출하면 적절한 숫자가 반환되며 원하는대로 작동한다.

```
class EnsembleTest {

    @Test
    @DisplayName("열거타입 테스트")
    void 열거타입테스트() {
        Assertions.assertEquals(Ensemble.DECTET.numberOfMusicians(), 10);
    }
}
```

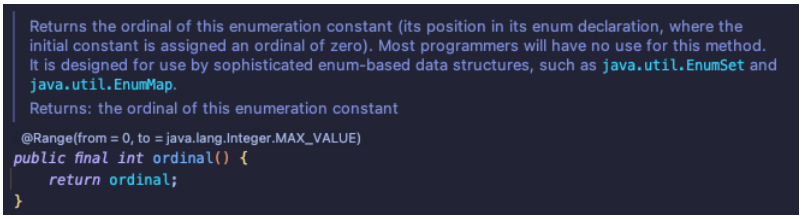
❓ 잘 작동하는데 왜 인스턴스 필드를 사용하라는거지?

순서에 영향을 준다는것이 문제이다.

> **SOLO와 DUET위치를 바꾸기만해도 반환값이 달라지게 된다.**
> **그리고 3중 4중주(triple quartet) 같은 것을 추가할려면 12의 값이므로 11번째 값도 추가해야하는데 11명 짜리 연주를 일컫는 말이 없다.**

📋 해결 방법 : 인스턴스 필드에 저장하자.

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

    private final int numberOfMusicians;

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

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



> Enum API에서도 ordinal 메서드는 대부분의 프로그래머가 쓸 일이 없으며
> EnumSet, EnumMap과 같이 열거 타입 기반에서 사용되는 것이라고 명시돼 있다.

🚀 아이템 36. 비트 필드 대신 EnumSet을 사용하라.

비트 필드 : 비트별 OR을 사용해 여러 상수를 하나로 모은 집합을 비트 필드라고 한다.

```
// 구닥다리 방법
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

    public void applyStyles(int styles) {
    }
}
```

위와 같은 비트 필드 열거 상수가 있다고 할 때, 아래와 같이 만들어진 집합을 비트필드라고 한다.

```
text.applyStyles(STYLE_BOLD | STYLE_ITALIC);
```

비트필드를 사용하면 비트 연산을 사용해 합집합과 교집합 같은 집합 연산을 효율적으로 수행할 수 있다.

❓ 그럼 왜 EnumSet을 사용하라고 하는걸까?

1. 정수 열거 상수의 단점을 그대로 가져간다.
2. 비트 필드 값이 그대로 출력되면 해석하기가 곤란하다.
3. 비트 필드 하나에 녹아 있는 모든 원소를 순회하기가 까다롭다.

4. 최대 몇 비트가 필요한지 API 작성시 미리 예측해 적절한 타입을 선택해야 한다. (int, long)

> 위와 같은 이유 때문에 비트 필드 대신 EnumSet을 추천한다.

📋 EumSet 사용해서 해결해보자.

```
public class Text {
    public enum Style { BOLD, ITALIC, UNDERLINE, STRIKETHROUGH }
    // 어떤 Set을 넣어도 되지만 EnumSet이 제일 좋다.
    public void applyStyles(Set<Style> styleSet){ ... }
}
```

> 비트를 직접 다룰 때 겪는 문제들을 `EnumSet`이 대부분 해결해준다.

```
text.applyStyles(EnumSet.of(Style.BOLD, Style.ITALIC));
```

❓ EnumSet이 가장 좋다면서 왜 Set을 넘길까?

모든 클라이언트가 EnumSet을 건네리라 짐작되는 상황이더라도 **인터페이스로 받는게 좋은 습관**이다.(아이템 64)

> 짐작이 될 뿐 확실하게 모든 클라이언트가 넘기지 않을 경우를 대비한다는 느낌으로 이해하면 좋다.

🚀 아이템 37. ordinal 인덱싱 대신 EnumMap을 사용하라

```
public class Plant {
    // 식물의 생애주기를 Enum 필드로 갖는다.
    enum LifeCycle { ANNUAL, PERENNIAL, BIENNIAL }

    final String name;
    final LifeCycle lifeCycle;

    public Plant(String name, LifeCycle lifeCycle) {
        this.name = name;
        this.lifeCycle = lifeCycle;
    }

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

Plant가 여럿 있고, 필요에 의해 생애주기를 기준으로 집합을 만들어 관리하고 싶을때 아래와 같이 만들 수 있다.

```
public class PlantApp {
    public void addPlant(List<Plant> garden) {
        @SuppressWarnings("unchecked")
        Set<Plant>[] plantsByLifeCycle = 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);
        }

        // 인덱스의 의미를 알 수 없어 직접 레이블을 달아 데이터 확인 작업을 요한다.
        for (int i = 0; i < plantsByLifeCycle.length; i++) {
            System.out.printf("%s: %s%n", Plant.LifeCycle.values()[i], plantsByLifeCycle[i]);
        }

        // 이런식으로 출력이 불가능하다.
        // System.out.println(plantsByLifeCycle);
        // 출력 결과 [Ljava.util.Set;@2accdbb5
    }
}
```

테스트 코드

```
class PlantAppTest {

    private List<Plant> plants = new ArrayList<>();

    @BeforeEach
    void 식물_기본_세팅() {
        plants.add(new Plant("장미", Plant.LifeCycle.BIENNIAL));
```

```
        plants.add(new Plant("벚꽃", Plant.LifeCycle.ANNUAL));
        plants.add(new Plant("국화", Plant.LifeCycle.PERENNIAL));
    }

    @Test
    @DisplayName("[비추천] 정원 식물 등록 테스트")
    void 식물_등록_비추천() {
        PlantApp plantApp = new PlantApp();

        plantApp.addPlant(plants);
    }
```

> 동작은 하지만 다양한 문제들이 존재한다.

- 배열과 제네릭은 호환되지 않기에 **비검사 형변환**을 수행해야 한다.
- 배열은 각 인덱스의 의미를 모르기에 출력 결과에 직접 레이블을 달아야 한다.
- 정확한 정숫값을 사용한다는 것을 개발자가 직접 보증해야 한다.
  - **정수는 열거 타입과 달리 타입이 안전하지 않기 때문**이다.
- 잘못된 값을 사용하는 경우 배열의 인덱스에 따른 값이 의도한 값을 보장할 수 없고, `ArrayIndexOutOfBoundsException`을 발생할 수 있다.

📋 해결방안 - EnumMap

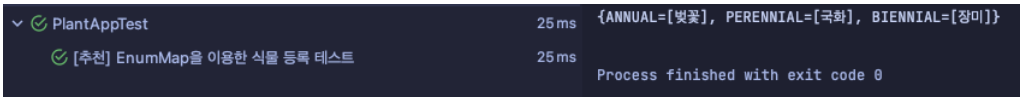
배열은 실질적으로 **열거 타입 상수를 값으로 매핑하는 역할**을 한다.

- 이를 Map으로 사용할 수 있도록 한다.
- 열거 타입을 키로 사용하도록 설계하는 `EnumMap`이 존재한다.

```
public void addPlantTypeEnumMap(List<Plant> garden) {
    Map<Plant.LifeCycle, Set<Plant>> plantByLifeCycle = new EnumMap<>(Plant.LifeCycle.class);

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

    for (Plant p : garden) {
        plantByLifeCycle.get(p.lifeCycle).add(p);
    }

    System.out.println(plantByLifeCycle);
}
```

테스트 코드

```
@Test
@DisplayName("[추천] EnumMap을 이용한 식물 등록 테스트")
void 식물_등록_추천() {
    PlantApp plantApp = new PlantApp();

    plantApp.addPlantTypeEnumMap(plants);
}
```

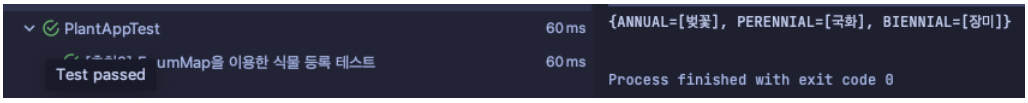


> 🌟 **코드가 훨씬 짧고 깔끔해졌으며 Map을 사용했기에 타입이 안정적이고 성능 또한 향상된다.**

> 성능이 빠른 이유는 내부적으로 배열을 사용하기 때문인데
> 이러한 구현을 내부로 숨겨 Map의 타입 안전성과 배열의 성능을 모두 얻어냈다.

Stream을 이용한 테스트코드

```
@Test
@DisplayName("[추천2] EnumMap을 이용한 식물 등록 테스트")
void 식물_등록_추천_EnumMap_최적화() {
    EnumMap<Plant.LifeCycle, Set<Plant>> garden = this.plants.stream()
            .collect(groupingBy(
                            p -> p.lifeCycle,
                            () -> new EnumMap<>(Plant.LifeCycle.class),
                            toSet()
                    )
            );


    System.out.println(garden);
}
```



✅ Stream과 EnumMap 비교

- Stream을 사용하면 EnumMap을 사용했을 때와는 다르게 동작한다.
  - EnumMap 버전은 언제나 식물의 생에 주기당 하나씩의 중첩 맵을 만들어준다.
  - Steram은 해당 생애주기에 속하는 식물이 있을 때만 만든다.
  - 한 해 살이와 여러해살이 2개의 종류만 있을 경우
    → EnumMap Map을 3개 만들고, Stream 버전에서는 2개를 만든다.

✅ 응용 - 상태전이 배열

```
// 액체(LiQUID) -> 고체(SOLID)로의 전이는 응고(FREEZE)
// 액체(LiQUID) -> 기체(GAS)로의   전이는 기화(BOIL)
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}
        };

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

> **두 상태를 조합해서 그 결과를 반환하는 코드이다.**

- 컴파일러가 ordinal과 배열 인덱스의 관계를 알 수 없다.
  - `Phase`나 `Transition` 메서드의 상태 혹은 상수의 순서가 바뀌었을때 이를 인지하여 `Transition` 메서드를 맞춰 수정해주지 않을 경우 런타임 오류가 발생한다.
- 상태가 커질수록 상전이 표의 크기는 기하급수적으로 커지게 된다.

EnumMap을 사용해서 해결한 코드

- 전이 하나를 얻기 위해서는 이전 상태(from)과 이후 상태(to)가 필요하다.
- Map 2개를 중첩하여 해결해보기
  - 내부 Map : 이전 상태 + Transition
  - 외부 Map : 이후 상태 + 내부 Map
  - OuterMap → 이후 상태 & 내부 Map → 이전 상태 & Transition

```
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;
        }

        // 내부 Map : 이전 상태 + Transition
        // 외부 Map : 이후 상태 + 내부 Map
        private static final Map<Phase, Map<Phase, Transition>> m =
                // groupingBy 에서는 전이를 이전 상태 기준으로 묶었다.
                Stream.of(values()).collect(groupingBy(t -> t.from,
                        () -> new EnumMap<>(Phase.class),
                        // 이후 상태를 전이에 대응시키는 EnumMap을 생성한다.
                        toMap(t -> t.to,
                                t -> t,
                                (x, y) -> y, // 선언만 하고 사용되지 않는다.
                                // 단순히 EnumMap을 얻기위해서 MapFactory가 필요하기에 수집기로 점층적 팩토리를 제공받기 위해서

                                () -> new EnumMap<>(Phase.class))));

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

새로운 상태의 추가 및 삭제에 유연한 코드

```
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);

        // 생략
    }
}
```

🚀 아이템 38. 확장할 수 있는 열거 타입이 필요하면 인터페이스를 사용하라

✅ 타입 안전 열거 패턴 (typesafe enum pattern)

```
public class TypeSafeEnumPattern {

    private final String direction;

    public static final TypeSafeEnumPattern NORTH = new TypeSafeEnumPattern("N");
    public static final TypeSafeEnumPattern SOUTH = new TypeSafeEnumPattern("S");
    public static final TypeSafeEnumPattern EAST = new TypeSafeEnumPattern("E");
    public static final TypeSafeEnumPattern WEST = new TypeSafeEnumPattern("W");

    public TypeSafeEnumPattern(String direction) {
        this.direction = direction;
    }
}
```

> Enum이 없을 때 사용하던 방식이다.

✅ 열거 타입

```
enum Direction {
    NORTH, SOUTH, EAST, WEST;
}
```

열거 타입은 거의 모든 상황에서 타입 안전 열거 패턴보다 우수하다.

단 하나의 예외가 있다. 타입 안전 열거 패턴은 확장할 수 있으나 열거타입은 확장할 수 없다.

하지만 대부분의 상황에서 열거 타입을 확장하려는 시도가 무조건 좋은 것은 아니다.

확장 타입 원소는 기반 타입 원소로 취급되지만 반대의 경우가 성립되지 않고, 기반 타입 + 확장 원소를 모두 순회할 방법도 없다.

그리고 확장성을 높히려고 할 때 고려해야할 것들과 구현이 너무 복잡해진다.

❓ 그럼 확장하지 말라는 것인가?

그건 또 아니다. 특정 상황에서는 어울린다.

예) 계산기의 연산 기능

✅ 인터페이스를 활용한 확장

위 이야기들을 살펴봤을 때 **열거타입의 확장은 구현할 수는 있지만, 고려해야 할 점이 많기에 조심해야한다**.

```
// Operation 인터페이스
@FunctionalInterface
public interface Operation {
    // 연산이 수행되는지 체크하기 위한 메서드 정의
    double apply(double x, double y);
}
```

```
// BasicOperation enum
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; }
}
```

> 각각의 상수에서 Operation 인터페이스의 apply 메서드를 구현했다.

```
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; }
}
```

> 이 열거타입의 연산은 기존 연산을 사용하던 곳은 어디든 사용할 수 있다.
> 열거타입에서 이제 따로 추상 메서드를 선언하지 않아도 된다. 즉 인터페이스 덕분에 **다형성이** 보장된다는 것이다.

위 코드를 두 가지 방법으로 작성할 수 있다.

1. **한정적 타입 토큰을 이용한 방법**

```
public class OperationApp {
    public static void main(String[] args) {
        double x = 4, y = 2;
        test(ExtendedOperation.class, x, y);
    }

    // test 메서드에 인수로 class 리터럴을 넘겨 이용한 방법이다.
    // Class 객체가 열거타입인 동시에 Operation 인터페이스의 구현체여야 한다는 의미이다.
    // 개인적으로 복잡해서 사용한다면 2번째 방법을 선호할 것 같다.
    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));
        }
    }
}
```

```
/Library/Java/JavaVirtualMachines/jdk-11.0.13.jdk/Contents/Home/bin/java ...
4.000000 ^ 2.000000 = 16.000000
4.000000 % 2.000000 = 0.000000
```

2. **한정적 와일드카드 타입을 넘기는 방식**

```
public static void main(String[] args) {
    double x = 4, y = 2;
    test(Arrays.asList(ExtendedOperation.values()), x, y);
}

// 인수로 Collection<? extends Operation> 타입의 한전정 와일드카드 타입을 넘기는 방식이다.
// 1번 보다 간결하고 보기 좋다.
// 하지만 EnumMap이나 EnumSet을 사용할 수 없다는 단점이 있다.
private static void test(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));
    }
}
```

📋 핵심 정리

- 열거 타입 자체는 확장할 수 없다.
- 인터페이스와 인터페이스를 구현하는 기본 열거타입을 같이 사용해 확장한 것처럼 사용할 수 있다.
  - 클라이언트는 인터페이스를 구현해 자신만의 열거(혹은 다른 타입)을 만들 수 있다.
- API가 (기본 열거 타입을 직접 명시하지 않고) 인터페이스 기반으로 작성됐다면, 기본 열거 타입의 인스턴스가 쓰이는 모든 곳을 새로 확장한 열거 타입의 인스턴스로 대체해 사용할 수 있다.

🚀 아이템 39. 명명 패턴보다 애너테이션을 사용하라

🌟 **변수나 메서드의 이름을 일관된 방식으로 작성하는 패턴을 명명 패턴이라고 한다!**

> `Junit3` 에서는 테스트 메서드 이름을 `test` 로 시작하게끔 했다.
> 이러한 명명 패턴 방식은 가시성이 좋지만 단점도 꽤나 크다.

⛔ 명명 패턴의 단점 1. 오타가 나면 안된다.

만약 `test`로 시작 되어야 할 메서드 이름이 **오타**로 인해 `tset`로 지어 Junit에 넘겨주면 이 클래스는 그냥 건너띄게 된다.

⛔ 명명 패턴의 단점 2. 명명 패턴을 의도한 곳에서만 사용되리라 보증할 방법이 없다.

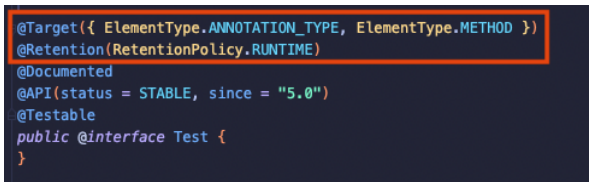
메서드가 아닌 클래스명을 `TestSafeMechanisms` 으로 지어 내부의 메서드를 테스트하길 기대할 수 있지만
JUnit은 클래스 이름에는 관심이 없기에 모두 무시해버린다.

⛔ 명명 패턴의 단점 3. 프로그램 요소를 매개변수로 전달할 마땅한 방법이 없다.

특정 예외를 던져야 성공하는 테스트가 있다고 할 때, 기대하는 예외 타입을 테스테 매개변수로 전달 할 방법이 마땅치가 않다.

✅ 대안책 - 애너테이션

> 🌟 애너테이션을 사용하면 위의 단점을 모두 해결할 수 있다.
> 그러므로 많은 단점을 가진 명명 패턴을 사용하기보다는 **애너테이션을 사용하자.**




```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface JunitTest {
}
```

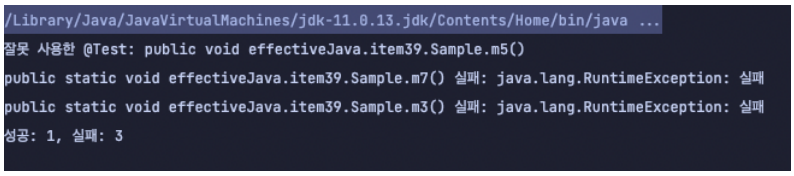
> 아무 매개변수 없이 단순히 대상에 마킹하는 용도로 사용하는 애너테이션이다.

- `@Test` 애너테이션 안에 작성된 애너테이션을 메타 애너테이션이라고 한다. (`@Target`, `@Retention` 등)
- `@Retention(RetentionPolicy.RUNTIME`
  이 애너테이션이 **런타임에도 유지되야 한다는 의미**다.
- `@Target({ ElementType.ANNOTATION_TYPE, ElementType.METHOD }`
  이 애너테이션이 애노테이션 타입의 **메타 애너테이션이나 메소드에 사용**할 수 있다는 의미

```
class SampleTest {
    @JunitTest
    public static void m1() { } //성공해야 한다.
    public static void m2() { }

    @JunitTest
    public static void m3() { //실패해야 한다.
        throw new RuntimeException("실패");
    }
    @JunitTest
    public void m5() { } //정적 메서드가 아니다.
    public static void m6(){ }

    @JunitTest
    public static void m7() { //실패해야 한다.
        throw new RuntimeException("실패");
    }
    public static void m8(){ }
}
```

> **`@Test` 애너테이션을 사용해서 성공한 메서드, 실패하는 메서드를 살펴보았다.**
> **실수로 `@Tset` 이라고 작성하게 되면 컴파일러에서 오류를 잡아주기에 앞서 말했던 문제점들이 해결이 된다.**

```
public class RunTests {
    public static void main(String[] args) throws Exception {
        int tests = 0;
        int passed = 0;
        Class<?> testClass = Class.forName("effectiveJava.item39.Sample");
        for (Method m : testClass.getDeclaredMethods()) {
            if (m.isAnnotationPresent(JunitTest.class)) {
                tests++;
                try {
                    m.invoke(null);
                    passed++;
                } catch (InvocationTargetException wrappedExc) {
                    Throwable exc = wrappedExc.getCause();
                    System.out.println(m + " 실패: " + exc);
                } catch (Exception exc) {
                    System.out.println("잘못 사용한 @Test: " + m);
                }
            }
        }
        System.out.printf("성공: %d, 실패: %d%n",
                passed, tests - passed);
    }
}
```



> 예제에는 Class.forName 부분이 args[0] 라고 돼있지만 이렇게 할 경우
> 아무것도 없는 배열을 사용하기 때문에 클래스 이름을 직접 적어줘야한다.

- 완전 정규화된 클래스 이름을 받아, 그 클래스에서 `@Test` 애너테이션이 달린 메서드를 차례로 호출한다.
  - `getDeclaredMethods` 메서드로 `testClass` 에 정의된 모든 메서드 정보를 for문으로 반복한다.
  - `isAnnotationPresent` 이 실행할 메서드를 찾아주는 메서드이다.
- 테스트 메서드가 예외를 던지면 리플렉션 메커니즘이 `InvocationTargetExcxeption` 으로 감싸 다시 던진다.

매개변수 없는 정적 메서드가 아닌 경우에는 `InvocationTargetException` 외의 다른 예외가 발생하기 때문에 추가적인 애너테이션 타입이 필요하다

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public@interfaceExceptionTest {
    Class<?extendsThrowable> value();
}
```

```
public class Sample2 {
    // 성공해야한다.
    @ExceptionTest(ArithmeticException.class)
    public static void m1() {
        int i = 0;
        i = i / i;
    }

    // 실패해야한다. (다른 예외 발생)
    @ExceptionTest(ArithmeticException.class)
    public static void m2() {
        int[] a = new int[0];
        int i = a[1];
    }
}
```

```
public class RunTests {
    public static void main(String[] args) throws Exception {
        int tests = 0;
        int passed = 0;
        Class<?> testClass = Class.forName("effectiveJava.item39.Sample2");
        for (Method m : testClass.getDeclaredMethods()) {
            //생략

            // 추가 부분
            if (m.isAnnotationPresent(ExceptionTest.class)) {
                tests++;
                try {
                    m.invoke(null);
```

```
                    System.out.printf("테스트 %s 실패: 예외를 던지지 않음%n", m);
                } catch (InvocationTargetException wrappedEx) {
                    Throwable exc = wrappedEx.getCause();
                    Class<? extends Throwable> excType =
                            m.getAnnotation(ExceptionTest.class).value();
                    if (excType.isInstance(exc)) {
                        passed++;
                    } else {
                        System.out.printf(
                                "테스트 %s 실패: 기대한 예외 %s, 발생한 예외 %s%n",
                                m, excType.getName(), exc);
                    }
                } catch (Exception exc) {
                    System.out.println("잘못 사용한 @ExceptionTest: " + m);
                }
            }
        }

        System.out.printf("성공: %d, 실패: %d%n",
                passed, tests - passed);
    }
}
```

```
테스트 public static void effectiveJava.item39.Sample2.m2() 실패: 기대한 예외 java.lang.ArithmeticException, 발생한 예외 java.lang.ArrayIndexOutOfBoundsException: Index 1 out of bounds for length 0
성공: 1, 실패: 1
```

> 실패 테스트이기 때문에 catch문에 추가적인 로직이 작성됐다.

- `m.getAnnotation(ExceptionTest.class).value()`
  - `@ExceptionTest` 애너테이션에 작성된 매개변수 값을 반환한다.
- 해당 예외의 클래스파일이 컴파일타임엔 존재하나 런타임에는 존재하지 않을 수 있다.
  - 테스트 러너가 `TypeNotPresentException` 을 던지게 된다.

❓ 예외가 여러개인 경우는 어떻게 처리할까?

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface ExceptionTests{
    Class<? extends Throwable>[] value();
}
```

> 앞서 작성했던것 처럼 단일 매개변수 선언도 수정없이 사용할 수 있다.

```
public class Sample3 {
    // 생략

    @ExceptionTests({IndexOutOfBoundsException.class, NullPointerException.class})
    public static void doublyBad() {   // 성공해야 한다.
        List<String> list = new ArrayList<>();

        // 자바 API 명세에 따르면 다음 메서드는 IndexOutOfBoundsException이나
        // NullPointerException을 던질 수 있다.
        list.addAll(5, null);
    }
}
```

```
// 배열 매개변수를 받는 애너테이션을 처리하는 코드 (243쪽)
  if (m.isAnnotationPresent(ExceptionTests.class)) {
      tests++;
      try {
          m.invoke(null);
          System.out.printf("테스트 %s 실패: 예외를 던지지 않음%n", m);
      } catch (Throwable wrappedExc) {
          Throwable exc = wrappedExc.getCause();
          int oldPassed = passed;
          Class<? extends Throwable>[] excTypes =
                  m.getAnnotation(ExceptionTests.class).value();
          for (Class<? extends Throwable> excType : excTypes) {
              if (excType.isInstance(exc)) {
                  passed++;
                  break;
              }
          }
          if (passed == oldPassed)
              System.out.printf("테스트 %s 실패: %s %n", m, exc);
      }
  }
```

🌟 **애너테이션이 나오면 절대다수의 상황**에서 **명명패턴보다 애너테이션을 활용하는 것을 적극 추천**한다.
명명패턴의 제약사항들을 애너테이션은 모두 해결해준다.

🚀 아이템 40. @Override 애너테이션을 일관되게 사용하라.

🌟 **자바의 기본으로 제공하는 어노테이션 중 보통의 프로그래머에게 가장 중요한 것은 `@Override` 일 것이다.**

```
public class OverrideTest implements Comparable<Object>{
    @Override
    public int compareTo(Object o) {
        return 0;
    }
}
```

⛔ 버그를 발생하는 코드

```
public class Bigram {
    private final char first;
    private final char second;

    public Bigram(char first, char second) {
        this.first = first;
        this.second = second;
    }

    public boolean equals(Bigram b) {
        return b.first == first && b.second == second;
    }

    public int hashCode() {
        return 31 * first + second;
    }

    public static void main(String[] args) {
        Set<Bigram> s = new HashSet<>();
        for (int i = 0; i < 10; i++) {
            for (char ch = 'a'; ch <= 'z'; ch++) {
                s.add(new Bigram(ch, ch));
            }
        }

        System.out.println("s.size() = " + s.size());
    }
}
```

- 소문자 2개로 구성된 바이그램 26개를 10번 반복해 집합에 추가한 다음 그 크기를 구한다.
- Set은 중복을 허용하지 않으므로 Bigram(a,a), Bigram(b,b) ~ Bigram(z,z) 까지 26개가 있어야 한다.

하지만 실행 결과는 아래와 같다.

```
s.size() = 260

Process finished with exit code 0
```

❓ 왜 26이 아닌 260이지?

우리의 아이템40 제목을 보면 바로 알아차릴 수 있다.
위 코드는 **오버라이딩 한게 아닌 오버로딩**을 한 것이다.

- Set은 중복을 허용하지 않기에 Object객체에서 선언한 equals, hashCode를 이용해서 비교한다.
- 하지만 우리의 equals는 오버로딩이기 때문에 재정의 한 것이 아니다.

즉 객체 식별성만을 비교하기에 모두 다른 객체로 인식하게 된 것이다.

✅ 해결 코드

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof Bigram)) return false;
    Bigram bigram = (Bigram) o;
    return first == bigram.first && second == bigram.second;
}

@Override
public int hashCode() {
    return Objects.hash(first, second);
}
```

> **아무 문제 없이 출력이 된다.**

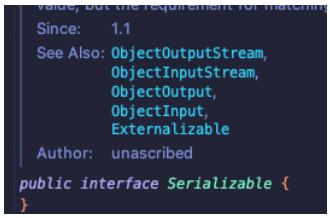
> 🌟 **상위 클래스의 메서드를 재정의 해야 할 때는 모든 메서드에 @Override 애너테이션을 추가하자.**

🚀 아이템 41. 정의하려는 것이 타입이라면 마커 인터페이스를 사용하라.

> 🌟 **마커 인터페이스란 아무 메서드도 선언하지 않은 인터페이스이다.**

위의 말로만 봤을 때는 무슨말이지? 라는 생각이 든다. 인터페이스 예시를 보면 바로 이해할 수 있다!

`Serializable` `Cloneable` `EventListener`



아래 코드는 `Serializable` 을 구현하지 않는 코드이다

```
@Getter
@Setter
@AllArgsConstructor
public class Book {
    private Long id;
    private String name;
    private int price;
}
```

```
class BookTest {
    @Test
    @DisplayName("직렬화 테스트")
    void 직렬화_테스트() throws IOException {
        Assertions.assertThrows(NotSerializableException.class, () -> {
            File file = new File("test");
            ObjectOutputStream objectOutputStream = new ObjectOutputStream(new FileOutputStream(file));
            objectOutputStream.writeObject(new Book(1L, "effective", 150000));
        });
    }
}
```

`Serializable` 을 구현하지 않은 상태에서 객체를 직렬화 할려고 하면 위처럼 `NotSerializableException` 을 던진다.

> 여기서 `NotSerializableException` 에러는 `Serializable` 구현되어 있는지 정도만 타입 확인만 하고 마는데,
> 이렇게 **단순히 타입 체크 정도만하는 인터페이스를 마커 인터페이스**라고 한다.

✅ 마커 애너테이션 vs 마커 인터페이스

1. **마커 인터페이스를 구현한 클래스의 인스턴스를 구분하는 타입으로 쓸 수 있다.**
   - **마커 애너테이션**은 구분하는 타입으로 사용할 수 없다.
   - **마커 애너테이션의 경우** 런타임시 발견할 오류를 마커인터페이스를 구현하면 **컴파일 타임에 발견**할 수 있다.
2. **마커 인터페이스는 적용 대상을 더 정밀하게 지정할 수 있다.**
   - **마커 애너테이션**은 `ElevmentType.Type` 으로 타겟을 지정하므로 모든 타입에 적용된다.
   - 마킹하고 싶은 특정 클래스에서만 **마커 인터페이스**를 구현하여 **적용 대상을 더 정밀하게 지정**할 수 있다.
3. **마커 어노테이션은 거대한 어노테이션 시스템의 지원을 받을 수 있다.**
   - 어노테이션을 적극적으로 사용하는 프레임워크에서는 마커 어노테이션을 쓰는 것이 **일관성을 지키는데 유리**하다.

❓ 그렇다면 언제 써야 하는걸까?

- 클래스와 인터페이스외의 프로그램 요소는 마킹이 필요한 경우 애너테이션을 사용할 수 밖에 없다.
- 클래스나 인터페이스는 마킹이 필요할 때 이 객체를 매개변수로 받는 메서드를 쓸일 있을까 고민하자.