이펙티브 자바 열거 타입과 애너테이션

item34. int 상수 대신 열거 타입을 사용하라

int 상수 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;
```

특정 경우를 상수 값으로 치환하여 표현하는 방식이다.

int 상수 패턴의 단점

- 타입 안전을 보장할 방법 x, 표현력도 안좋음
- 네임스페이스가 없기에 동일한 접두어가 반복됨
- 오렌지와 사과를 대체해도 == 연산자가 동일한 결과가 나온다.
- 중간에 값 하나가 빠져서 상수 값이 바뀌면 모두 다시 작성해야 한다.
- 문자열로 출력하기 까다로움

자바 열거 타입(Enum)

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

- 상수 하나당 인스턴스를 하나씩 만들어 public static final 필드로 공개하는 것이다.
  - 인스턴스가 통제된다.
  - 원소가 하나이면 싱글턴으로 볼 수 있다.
- 컴파일 타입 안정성을 제공한다.
  - 이전의 int 상수 패턴처럼 ORANGE가 갈 곳에 APPLE이 간다면, 명확히 타입 에러가 발생한다.
- 네임스페이스를 제공하여, 이름이 같은 상수도 평화롭게 공존할 수 있다.
  - APPLE.RED와 ORANGE.RED는 구분된다.
- toString()이 출력하기에 적합한 문자열을 내어준다.
- 열거 타입에는 다양한 메서드나 필드도 추가 가능하다.
  - 추가로 임의의 인터페이스도 구현하게 할 수 있다.

열거타입 상수마다 동작이 달라지는 메서드

```
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;
            }
            default -> throw new IllegalStateException();
        }
    }
}

@Test
public void operationApplyTest() {
    double x = 10;
    double y = 15;

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

- 간단히 switch문을 활용하여 구성해보았다.
- 이 방법엔 단점이 두개 있다.
  - 도달할 일 없는 throw 문을 작성해야 한다.
  - 새로운 상수가 생길 때마다 case를 추가하는 것을 잊으면 안된다.
    - 만일 잊게 되면, 런타임 에러를 만나게 될 것이다.

열거타입 상수마다 동작이 달라지는 메서드 → 추상 메서드 이용하기

```
enum 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;

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

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

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

@Test
```

```
public void operationApplyTest() {
    double x = 10;
    double y = 15;

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

- 상수별 클래스 몸체에 apply 메서드를 재정의하였다.
- 위와 같이 추상 클래스를 이용하면, case를 이용할 때처럼 무언가 빼먹을 일이 없다.
    - 실수로 재정의하지 않았다면, 컴파일 오류로 알려준다.
- 내부적으로 symbol이라는 필드를 두어, +, , , / 등 알맞은 기호를 저장했다.
- toString()을 재정의하여 symbol 필드를 반환하도록 만들었다.

열거타입 상수끼리 코드 공유

```
enum PayrollDay {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, S
UNDAY;

    private static final int MINS_PER_SHIFT = 8 * 60;

    int pay(int minutesWorked, int payRate) {
        int basePay = minutesWorked * payRate;
        int overtimePay;

        switch(this) {
// 주말case SATURDAY : case SUNDAY :
                overtimePay = basePay / 2;
                break;
// 주중default:
```

```
                overtimePay = minutesWorked <= MINS_PER_SHIFT
? 0 : (minutesWorked - MINS_PER_SHIFT) * payRate / 2;
        }

        return basePay + overtimePay;
    }
}

@Test
public void payrollDayTest() {
    int pay1 = PayrollDay.FRIDAY.pay(480, 200);
    System.out.println("pay1 = " + pay1);

    int pay2 = PayrollDay.FRIDAY.pay(540, 200);
    System.out.println("pay2 = " + pay2);

    int pay3 = PayrollDay.SUNDAY.pay(480, 200);
    System.out.println("pay3 = " + pay3);
}
```

switch문을 이용한 형태는 이전과 같이 case를 반드시 같이 추가해주어야 한다.

열거타입 상수끼리 코드 공유 → 전략 상수 패턴 사용

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

    private final PayType payType;
```

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

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

// 전략 열거 타입enum PayType {
        WEEKDAY {
            @Override
            int overtimePay(int minsWorked, int payRate) {
                return minsWorked <= MINS_PER_SHIFT ? 0 : (mi
nsWorked - MINS_PER_SHIFT) * payRate / 2;
            }
        },
        WEEKEND {
            @Override
            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, payRat
e);
        }
    }
}
```

- 새로운 상수를 추가할 때 무조건 잔업수당 전략을 선택해야 한다.
- 이 패턴은 switch문보다 조금 더 복잡하지만, 더 안전하고 유연하다.
- 단, 기존 열거 타입에 상수별 동작을 혼합해 넣는다면, switch문이 더 좋은 선택이 될 수 있다.

정리

- 열거 타입은 정수 상수보다 뛰어나다.
  - 가독성도 더 좋고 안전하고 강력하다.
- 대다수 열거 타입은 명시적 생성자나 메서드 없이 쓰이지만, 각 상수를 특정 데이터와 연결짓거나 상수마다 다르게 동작할 때는 필요하다.
  - 이 경우, 보통은 추상 메서드를 선언한 뒤, switch문 대신 상수별 메서드 구현이 낫다.
  - 열거 타입 상수가 같은 코드를 공유한다면, 전략 열거 타입 패턴을 사용하자.

item35. ordinal 메서드 대신 인스턴스 필드를 사용하라

잘못된 예시

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

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

```
@Test
public void ensembleTest() {
    Ensemble solo = Ensemble.SOLO;
    System.out.println("solo = " + solo);
    System.out.println("solo.numberOfMusicians() = " + solo.n
umberOfMusicians());
}
```

- ordinal()의 값을 활용하여 오케스트라 단원의 수를 출력하는 메서드를 구현했다.
- 하지만 이는 잘못된 구현이다. ordinal()은 EnumSet, EnumMap과 같이 열거타입 기반의 범용 자료구조에 쓰일 목적으로 설계되었다.
    - 만일 중간에 다른 상수라도 추가되면 numberOfMusicians()는 혼란에 빠진다.
- ordinal()을 사용하지 말고, 각 인스턴스의 필드 값으로 저장해두자.

EnumSet에서의 ordinal()의 활용

```
public boolean contains(Object e) {
    if (e == null)
        return false;
    Class<?> eClass = e.getClass();
    if (eClass != elementType && eClass.getSuperclass() != el
ementType)
        return false;

    return (elements & (1L << ((Enum<?>)e).ordinal())) != 0;
}
```

EnumSet은 ordinal()의 값을 ***bit-vector***로 이용하여 중복 판단을 한다.

EnumMap에서의 ordinal()의 활용

```
public boolean containsKey(Object key) {
    return isValidKey(key) && vals[((Enum<?>)key).ordinal()]
!= null;
}
```

EnumMap은 ordinal()의 값을 값 배열의 인덱스로 활용한다.

좋은 예시

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

    private final int numberOfMusicians;

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

    public int numberOfMusicians() {
        return numberOfMusicians;
    }
}

@Test
public void ensembleTest() {
```

```
    Ensemble solo = Ensemble.SOLO;
    System.out.println("solo = " + solo);
    System.out.println("solo.numberOfMusicians() = " + solo.n
umberOfMusicians());
}
```

item36. 비트 필드 대신 EnumSet을 사용하라

비트 필드 열거 상수

```
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 String applyStyles(int styles) {
    StringJoiner sj = new StringJoiner(", ");

    if((styles & STYLE_BOLD) == 1) {
        sj.add("BOLD");
    }

    if((styles & STYLE_ITALIC) == 2) {
        sj.add("ITALIC");
    }

    if((styles & STYLE_UNDERLINE) == 4) {
        sj.add("UNDERLINE");
    }

    if((styles & STYLE_STRIKETHROUGH) == 8) {
```

```
            sj.add("STRIKETHROUGH");
        }

        return sj.toString();
    }

    @Test
    public void bitFieldTest() {
        String appliedStyles = applyStyles(STYLE_BOLD | STYLE_STR
    IKETHROUGH | STYLE_UNDERLINE);
        System.out.println("appliedStyles = " + appliedStyles);
    }
```

- 비트 필드 열거 상수 코드의 예이다.
- 여러 상수를 하나의 집합으로 모을 수 있는 장점을 가졌다.
  - 이렇게 만들어진 집합을 비트 필드라고 한다.
  - 합집합 교집합 등의 연산에 유리하다.
- 정수 열거 상수의 단점을 그대로 가져간다.
- 모든 원소를 순회하기 쉽지 않다.
- 최대 몇비트가 필요한지 API 작성 시 미리 예측하여 적절한 타입을 선택해야 한다.
  - 안그러면 나중에 API에서 int, long 등 타입을 수정해야 하는 일이 벌어진다.

EnumSet으로 비트 필드 대체하기

```
enum Style { BOLD, ITALIC, UNDERLINE, STRIKETHROUGH }

public void applyStyles(Set<Style> styles) {
    StringJoiner sj = new StringJoiner(", ");

    for (Style style : styles) {
        sj.add(style.name());
```

```
    }

    System.out.println("applied styles = " + sj);
}

@Test
public void enumSetTest() {
    applyStyles(EnumSet.of(Style.BOLD, Style.ITALIC, Style.UN
DERLINE));
}
```

- 원소가 총 64개 이하라면 RegularEnumSet을 사용하여 비트필드만큼의 성능을 내준다.
  - 그 이상은 JumboEnumSet을 사용하도록 내부적으로 판단해서, 크기에도 유연하다.
- 원소를 순회하기도 쉽다.

item37. ordinal 인덱싱 대신 EnumMap을 사용하라

ordinal()을 배열 인덱스로 이용한 예제

```
static 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 "Plant{" +
                "name='" + name + '\'' +
                ", lifeCycle=" + lifeCycle +
                '}';
    }
}

@Test
public void plantsByLifeCycleTest() {
    Set<Plant>[] plantsByLifeCycle = (Set<Plant>[]) new Set[P
lant.LifeCycle.values().length];

    List<Plant> garden = new ArrayList<>(List.of(
            new Plant("A", Plant.LifeCycle.ANNUAL),
            new Plant("B", Plant.LifeCycle.PERENNIAL),
            new Plant("C", Plant.LifeCycle.BIENNIAL),
            new Plant("D", Plant.LifeCycle.ANNUAL)
    ));

    for (int i = 0; i < plantsByLifeCycle.length; i++) {
        plantsByLifeCycle[i] = new HashSet<>();
    }

    for (Plant plant : garden) {
        plantsByLifeCycle[plant.lifeCycle.ordinal()].add(plan
t);
    }

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

- plantsByLifeCycle을 Set의 배열로 values().length만큼 생성해두었다.
    - 그리고 ordinal()을 통해 인덱스를 짚고 있다.

단점

- Set 클래스는 제네릭 타입을 받는데, 제네릭 타입은 배열과 호환성이 좋지 않다.
    - 그래서 비검사 형변환을 수행해야 하고, 깔끔히 컴파일되지 않는다.
- 정확한 정수값을 사용하는지 스스로 보증해야 한다. 열거 타입만큼 명확하지 않다.

EnumMap을 사용한 예

```
@Test
public void plantEnumMapTest() {
    List<Plant> garden = new ArrayList<>(List.of(
            new Plant("A", Plant.LifeCycle.ANNUAL),
            new Plant("B", Plant.LifeCycle.PERENNIAL),
            new Plant("C", Plant.LifeCycle.BIENNIAL),
            new Plant("D", Plant.LifeCycle.ANNUAL)
    ));

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

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

    for (Plant plant : garden) {
        plantsByLifeCycle.get(plant.lifeCycle).add(plant);
    }
```

```
    System.out.println("plantsByLifeCycle = " + plantsByLifeC
ycle);
}
```

- ordinal()을 사용할 때보다 코드가 많이 깔끔해졌다.
- 제네릭 타입도 이전에 배열로 하던 방식과 다르게 컴파일 경고나 오류 없이 이용할 수 있다.
- EnumMap은 성능도 좋다. (내부적으로는 배열을 사용한다.)

정리

- 배열의 인덱스를 위해 ordinal()을 쓰는 것은 일반적으로 좋지 않다.
- 대신 EnumMap을 사용하자.
- 다차원 관계는 EnumMap<..., EnumMap<...>>으로 표기하자.
- ordinal()은 웬만해선 쓰지 말자.

item38. 확장할 수 있는 열거 타입이 필요하면 인터페이스를 사용하라

기본적으로 열거 타입의 확장은 지원하지 않지만 필요할때 인터페이스를 통해 확장하면 된다.

```
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;
    }
}

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;
    }
}
```

- Operation 인터페이스를 구현하여, BasicOperation과 ExtendedOperation을 만들었다.
    - 지원하는 메서드의 형태가 같아 어디서든 대체하여 쓸 수 있다.

클래스로 받기

```
@Test
public void operationTest() {
    double x = 4.0;
    double y = 2.0;
    getEnumClass(BasicOperation.class, x, y);
```

```
    getEnumClass(ExtendedOperation.class, x, y);
}

private <T extends Enum<T> & Operation> void getEnumClass(Cla
ss<T> enumClass, double x, double y) {
    for (Operation operation : enumClass.getEnumConstants())
{
        System.out.printf("%f %s %f = %f%n", x, operation, y,
operation.apply(x, y));
    }
}
```

- enum 타입의 클래스를 받아 getEnumConstants()를 통해 클래스가 가진 apply 메서드들을 실행해보았다.
- <T extends Enum<T> & Operation>의 의미는 enum이며 Operation을 상속받았음을 이야기한다.

정리

- 열거 타입(enum)은 확장할 수 없다.
- 단, 인터페이스를 통해 여러 열거 타입에 동일한 인터페이스를 구현하게 하면 마치 확장하는 것과 비슷한 효과를 낸다.
- 인터페이스로 작성되었다는 가정하에 서로 얼마든지 대체도 가능하다.

item39. 명명 패턴보다 애너테이션을 사용하라

명명패턴

메서드의 이름 앞을 test...로 짓는 등 이름에 패턴을 주어 Reflection 등으로 해당 패턴 검출 시 특정 작업을 수행하는 식의 코딩 형식이다.

단점

- 오탈자의 위험
- 메서드, 파라미터, 클래스명 등 영역에 대한 설정이 불가능하다.
- 프로그램 요소를 매개변수로 전달할 마땅한 방법이 없다.
  - ex) 특정 예외가 던져져야 올바르게 실행되는 메서드가 있다면?

예시: @MethodTest, 일반 메서드 애너테이션

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

- @Retention과 @Target은 메타 애너테이션이라 불린다.
- @Retention은 생존기간을 나타낸다.
  - RetentionPolicy.RUNTIME: 런타임에도 유지되어야 한다는 표시이다.
- @Target(ElementType.METHOD): 해당 애너테이션이 반드시 메서드에 적용되어야 한다는 것을 알려준다.
- @MethodTest: 애너테이션과 같이 아무 매개변수 없이 단순히 대상에 마킹하는 애너테이션을 마크 애너테이션이라고 한다.

```
public class Sample {
    @MethodTest
    public static void m1() { }
    public static void m2() { }
```

```
    @MethodTest
    public static void m3() {
        throw new RuntimeException("실패");
    }
    public static void m4() { }

    @MethodTest
    public void m5() { }// 잘못 사용한 예: 정적 메서드가 아니다.publ
ic static void m6() { }
    @MethodTest
    public static void m7() {
        throw new RuntimeException("실패");
    }
    public static void m8() { }
}

@Test
public void sampleClassAnnotationTest() throws ClassNotFoundE
xception {
    int tests = 0;
    int passed = 0;
    Class<?> testClass = Class.forName("item39.Sample");
    Method[] declaredMethods = testClass.getDeclaredMethods
();

    for (Method m : declaredMethods) {
        if(m.isAnnotationPresent(MethodTest.class)) {
            tests++;
            try {
                m.invoke(null);
                System.out.println(m + ", 성공");
                passed++;
            } catch (InvocationTargetException wrappedExc) {
                Throwable exc = wrappedExc.getCause();
                System.out.println(m + ", 실패: " + exc);
```

```
            } catch (Exception exc) {
                System.out.println("잘못 사용한 @Test: " + m +
", " + exc);
            }
        }
    }

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

- @Test 애너테이션은 Sample 클래스의 의미에 직접적인 영향을 주진 않고, 추가 정보를 주어 이 애너테이션에 관심이 있다면 특별한 처리를 할 수 있는 기회를 준다.
- 클래스를 가져와서 메서드를 가져오고(getDeclaredMethods()) 리플렉션 API(Method)를 통해 해당 클래스의 메서드를 불러온다.
- isAnnotationPresent()는 특정 애노테이션이 붙어있는지 확인할 수 있는 메서드이다.
- invoke(null)을 통한 메서드 호출에 성공한다면, 아직 인스턴스화 전에 호출할 수 있는 메서드였으므로 정적 메서드였을 것이다.
    - 그러므로, 정적 메서드가 아닌 메서드는 호출에 실패하고 null을 넘긴 덕에 NullPointerException을 던지게 될 것이다.

정리

- 애너테이션으로 할 수 있는 일을 굳이 명명패턴으로 처리하지 말자
- 자바 프로그래머라면 애너테이션 타입들을 잘 사용하도록 노력해보자

item40. @Override 애너테이션을 일관되게 사용하라

@Override 가 없을때 하기 쉬운 실수

```
public class Item40Test {
    static 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;
        }
    }

    @Test
    public void bigramTest() {
        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));
            }
        }

        Assertions.assertEquals(26, s.size());// 실제 값 260
    }
}
```

- bigramTest() 가 원한 결과는 s.size()가 26인 것이지만 실제로는 260이 나왔다.
- equals()에 @Override 애너테이션을 붙이지 않아서 생긴 실수가 있다.

- Object에서 상속받는 equals()는 원래 Object 타입의 파라미터를 받는데, Bigram의 파라미터를 받고 있다.
- 그래서 Set에서 비교에 사용되는 equals()가 제대로 정의되지 않고, 오직 객체 주소의 동치만 비교하는 기본 Object의 equals()가 쓰이고 있던 것이다.
- 실제 오버라이딩(재정의)된 것이 아니라 오버로딩을 하고 있는 것이다.

적용하기

```
@Override
public boolean equals(Object o) {
    if(!(o instanceof Bigram)) {
        return false;
    }

    Bigram b = (Bigram) o;
    return b.first == first && b.second == second;
}
```

- 위와 같이 @Override 애너테이션을 달면, 실제로 상속받은 메서드가 아니면 에러를 내주기 때문에 실수할 확률이 적어진다.
- @Override를 통해 시그니쳐가 올바른지 다시한번 확인하자.

item41. 정의하려는 것이 타입이라면 마커 인터페이스를 사용하라

마커 인터페이스

자신을 구현하는 클래스가 특정 속성을 가짐을 표시해주는 인터페이스

- ex) Serializable 은 자신을 구현한 클래스의 인스턴스는 ObjectOutputStream 을 통해 쓸 수 있다(직렬화할 수 있다)고 알려준다

마커 인터페이스 장점

- 마커 인터페이스는 이를 구현한 클래스의 인스턴스들을 구분하는 타입으로 사용할 수 있다.
  - 마커 애너테이션은 타입으로 사용할 수는 없다.
  - 타입으로 사용할 수 있기 때문에 런타임에야 발견할만한 오류를 컴파일 타임에 발견할 수도 있다.
- 마커 인터페이스는 적용 대상을 더 정밀하게 지정할 수 있다.
  - ex) 특정 인터페이스를 구현한 클래스에만 적용하고 싶다면, 그 클래스에서만 인터페이스를 구현(확장)하면 된다.
    - 위와 같이 설정하면 자동으로 하위 타입임이 보장된다.

ObjectOutputStream의 설계 미스

ObjectOutputStream 의 직렬화 메서드인 writeObject()의 구현 코드는 아래와 같다.

```
public final void writeObject(Object obj) throws IOException
{
    if (enableOverride) {
        writeObjectOverride(obj);
        return;
    }
    try {
        writeObject0(obj, false);
    } catch (IOException ex) {
        if (depth == 0) {
            writeFatalException(ex);
        }
        throw ex;
```

```
    }
}
```

Serializable 마커 인터페이스의 경우, 설계 미스로 직렬화 메서드인 writeObject()
가 Serializable 타입을 받게 했어야 하는데, Object 객체를 받게 해서 직렬화할 수 없는 객체를 넘겨도 런타임에 알 수 밖에 없는 치명적인 단점이 생겼다.

마커 애너테이션 장점

- 거대한 애너테이션 시스템의 지원을 받는다.
    - 애너테이션을 적극 활용하는 프레임웍에서는 마커 애너테이션을 쓰는 쪽이 유리하다.

정리

- 파라미터로서 활용하거나 반환 타입으로서 활용하는 등 타입으로 쓸 목적이 있다면, 마커 인터페이스를 쓰자
- 프레임웍에서 애너테이션을 적극 활용한다면 마커 애너테이션을 쓰자
    - ElementType.TYPE 인 마커 애너테이션을 작성하고 있다면, 마커 인터페이스를 고려해보자.