6장 열거타입과 애너테이션

아이템 34. int 상수 대신 열거 타입을 사용하라.

- 열거 타입이란, 일정 개수의 상수 값을 정읳나 다음 그 외의 값은 허용하지 않는 타입이다.

정수 열거 패턴의 단점

- 타입 안전을 보장할 방법이 없으며 표현력도 좋지 않다.

  ex) 오렌지를 건네야할 메서드에 사과를 보낸 후 동등연산자로 비교해도 경고 메시지를 출력하지 않는다.
- 상수의 값이 변경되면 클라이언트도 다시 컴파일해야한다.
- 정수 상수는 숫자로 표현되기 때문에 toString과 같이 의미가 담겨 있는 문자열로 추력하기 힘들다.

열거 타입의 특징

- 열거 타입은 인스턴스 통제 된다.
  - 열거 타입 자체는 클래스 이며, 상수 하나당 자신의 인스턴스를 하나씩 만들어 public static final 필드로 공개한다.
- 열거 타입은 컴파일타입 타입 안전성을 제공한다.
- 열거 타입에는 각자의 이름 공간이 있어서, 이름 중복을 허용한다.
- 열거 타입에 임의의 메서드나 필드를 추가하거나, 인터페이스를 구현할 수 있다.

메서드와 필드를 갖는 열거 타입

```
데이터와 메서드를 갖는 열거 타입
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),
    URANUS (8.683e+25, 2.556e7),
    NEPTUNE(1.024e+26, 2.477e7);

    private final double mass;           // 질량(단위: 킬로그램)
    private final double radius;         // 반지름(단위: 미터)
    private final double surfaceGravity; // 표면중력(단위: m / s

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

    // 생성자
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
        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;  // F = ma
    }
}
```

- 이와 같이 열거 타입 상수 각각을 특정 데이터와 연결지으려면 생성자에서 데이터를 받아 인스턴스 필드에 저장하면 된다.
- **열거 타입은 불변이라 모든 필드는 final이어야한다.**

값에 따라 분기하는 열거타입

- 하나의 메서드가 상수별로 다르게 동작하는 경우이다.

1.switch문 열거 타입

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

- 해당 코드는 새로운 상수를 추가하면 case문도 추가해야한다.

2. 상수별 메서드 구현을 활용한 열거타입

```
public enum Operation {
    PLUS  { public double apply(double x, double y) { return x
    MINUS { public double apply(double x, double y) { return x
    TIMES { public double apply(double x, double y) { return x
    DIVIDE { public double apply(double x, double y) { return

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

apply가 추상 메서드이기 때문에 재정의하지 않는다면 컴파일 오류로 알려준다.

상수별 메서드 구현을 상수별 데이터와 결합할 수도 있다.

3.상수별 클래스 몸체와 데이터를 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);
```

위와 같이 toString을 재정의한다면 반환하는 문자열을 해당 열거 타입 상수로 변환해주는 fromString메서드도 함께 제공하면 좋다. 단, 타입 이름을 적절히 바꿔야하고 모든 상수 문자열 표현이 고유해야한다.

- 열거 타입용 fromString 메서드 구현

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

// 지정한 문자열에 해당하는 Operation이 존재한다면 반환
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 초기화**될 때 이다.

열거 타입 상수는 생성자에서 자신의 인스턴스를 맵에 추가할 수 없다. 열거 타입의 정적 필드 중 생성자에서 접근 할 수 있는 것은 상수 변수 뿐이기 때문이다.

전략 열거 타입 패턴

- 열거 타입 상수 일부가 같은 동작을 공유하는 경우이다.

1.값에 따라 분기하여 코드를 공유하는 열거 타입

```
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){  //switch문을 통한 분기
            case SATURDAY: CASE SUNDAY:
                overtimePay = basePAy / 2;
                break;
            default:
                overtimePay = minutesWorked <= MINS_PER_SHIFT

    }
    return basePay + overtimePay;
```

간결하지만 관리 관점에서는 위험한 코드다.

```
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(){
                 return minsWorked <= MINS_PER_SHIFT ? 0 :
                 (minsWorked - MINS_PER_SHIFT) * payRate / 2;
            }
        },
        WEEKEND{
            int overtimePay(){
              return minsWorked * payRate / 2;
            }
        };

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

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

이 패턴은 switch문 보다는 복잡하지만 더 안전하고 유연하다.

아이템35. ordinal 메서드 대신 인스턴스 필드를 사용하라.

Ordinal 메서드

- 열거타입에서 몇 번째 위치인지 반환하는 메서드

```
public enum Ensemble {
    Solo, DUET, TRIO

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

Ordinal 사용의 문제점

- 동일한 정수를 사용하는 상수는 사용할 수 없다.
- 중간 값을 비울 수 없다.

이를 해결하기 위해 열거타입 상수에 연결된 값은 ordinal 메서드로 얻지 말고 필드에 추가하자.

```
public enum Ensemble {
    Solo(1), DUET(2), TRIO(3)

    private final int numberOfMusicians;
    Ensemble(int size) {this.numberOfMusicians = size;}
    public int numberOfMusicians() { return ordinal() + 1; }
}
```

아이템 36. 비트 필드 대신 EnumSet을 사용하라.

비트 필드

- 정수 상수의 집합을 표현하는 방법 중 하나로, 각 비트를 특정 상수에 대응시키는 방식. 하지만 비트 필드는 가독성이 낮고, 오류가 발생하기 쉽다.

```
public class StyleWithBitField {
    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; //

    public void applyStyles(int styles) {
//        styleWithBitField.applyStyles(STYLE_BOLD | STYLE_UN
    }
}
```

EnumSet

- 열거형을 기반으로 하는 집합 구현체 중 하나. 이 클래스는 열거 상수 집합을 효율적으로 저장하고 다룰 수 있도록 최적화되어있다.

```
import java.util.EnumSet;
import java.util.Set;

public class Text {

    public void applyStyles(Set<Style> styles) {
//        applyStyles(EnumSet.of(Style.BOLD, Style.UNDERLINE)
    }

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

아이템 37. ordinal 인덱싱 대신 EnumMap을 사용하라.

ordinal 인덱싱

- 열거형 상수가 선언된 순서에 따라 숫자를 매기는 것을 말한다. 열거형의 첫번 째 상수는 0부터 시작하여 상수가 선언된 순서대로 1씩 증가하는 값을 가지게 된다.

```
public class Plant {
    final String name;
    final LifeCycle lifeCycle;

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

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

public enum LifeCycle {
    ANNUAL, PERNNIAL, BIENNIAL
}
```

Plant 클래스는 LifeCycle 열거 타입을 멤버 변수로 가지고 있다. 이를 이용해서 식물들을 생애주기로 묶어보자.

```
//Ordinal 사용 예
public static void usingOrdinalArray(List<Plant> garden) {
        Set<Plant>[] plantsByLifeCycle = (Set<Plant>[]) new S
        for (int i = 0 ; i < plantsByLifeCycle.length ; i++)
            plantsByLifeCycle[i] = new HashSet<>();
        }

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

        for (int i = 0 ; i < plantsByLifeCycle.length ; i++)
```

```
            System.out.printf("%s : %s%n",
                    LifeCycle.values()[i], plantsByLifeCycle[
        }
    }
```

EnumMap

- 열거형을 키로 사용하는 맵이다. 이 맵은 열거형의 모든 상수를 키로 사용할 수 있으며, 각 상수에 대한 값에 접근하기 위해 배열을 사용한다. 열거형을 키로 사용하기 때문에 EnumMap은 해시 테이블 대신 배열을 사용하여 효율적으로 구현된다.

```
public static void usingEnumMap(List<Plant> garden) {
        Map<LifeCycle, Set<Plant>> plantsByLifeCycle = new En

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

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

        //EnumMap은 toString을 재정의하였다.
        System.out.println(plantsByLifeCycle);
    }
```

ordinal을 사용한 코드와 다르게 안전하지 않은 형변환을 사용하지 않는다.

EnumMap 자체가 toString을 제공하기 때문에 번거롭지 않다.

내부에서 배열을 사용하기 때문에 내부 구현 방식을 안으로 숨겨서 Map의 타입 안정성과 배열 성능을 모두 얻어냈다.

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

```
// 확장 가능한 열거 타입을 위한 인터페이스
interface Animal {
    void sound();
}

// 기본 열거 타입 구현
enum DefaultAnimal implements Animal {
    DOG {
        @Override
        public void sound() {
            System.out.println("Woof");
        }
    },
    CAT {
        @Override
        public void sound() {
            System.out.println("Meow");
        }
    },
    BIRD {
        @Override
        public void sound() {
            System.out.println("Tweet");
        }
    }
}

// 새로운 열거 타입을 추가하기 위해 Animal을 구현하는 열거 타입 정의
enum CustomAnimal implements Animal {
    LION {
        @Override
        public void sound() {
            System.out.println("Roar");
        }
    },
    ELEPHANT {
        @Override
        public void sound() {
```

```
            System.out.println("Trumpet");
        }
    }
}

public class Main {
    public static void main(String[] args) {
        // 기본 열거 타입 사용
        DefaultAnimal.DOG.sound();  // 출력: Woof
        DefaultAnimal.CAT.sound();  // 출력: Meow

        // 새로운 열거 타입 사용
        CustomAnimal.LION.sound();  // 출력: Roar
        CustomAnimal.ELEPHANT.sound();  // 출력: Trumpet
    }
}
```

해당 코드에서는 animal 인터페이스를 정의하여 열거타입을 구현하도록 하였다. 그리고 DefaultAnimal 이라는 기본 열거 타입과 CustomAnimal이라는 새로운 열거 타입을 정의하여 각각의 상수에 대해 다른 동작을 구현하였다. 클라이언트는 Animal 인터페이스를 사용하여 두 열거 타입의 인스턴스를 다룰 수 있다.

아이템39. 명명 패턴보다 애너테이션을 사용하라.

명명 패턴

- 변수나 함수의 이름을 일관된 방식으로 작성하는 패턴
- 전통적으로 도구나 프레임워크에서 특별히 다뤄야할 프로그램 요소를 구분하기 위해 사용됨

명명 패턴의 단점

- 오타가 나면 안된다.
- 올바른 프로그램 요소에서만 사용되리라 보증할 방법이 없다.
- 프로그램 요소를 매개변수로 전달할 마땅한 방법이 없다.

애너테이션을 사용하면 위의 단점들을 해결가능하다.

마커 애너테이션

```
/**
* @Test
* 테스트 메서드임을 선언하는 애너테이션이다.
*/
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test{}
```

메타 애너테이션

- 애너테이션 선언에 다는 애너테이션
- @Traget - 적용 대상, 반드시 메서드에 선언되어야 한다는 표시
- @Retention - 런타임에도 유지되어야한다는 표시

마커 애너테이션 처리기

- 마커 애너테이션은 적절한 애너테이션 처리기가 필요하다.

```
// 마커 애너테이션 처리기
public class RunTests {
  public static void main(String[] args) throws Exception {
    int tests = 0;
    int passed = 0;
    Class<?> testClass = Class.forName(args[0]);
    for (Method m : testClass.getDeclaredMethods()) {
      if (m.isAnnotationPresent(Test.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);
    }
  }
```

리플렉션을 이용하여 마커 애너테이션을 찾고, 예외 발생 시 InvocationTargetException으로 래핑 되어 해당 예외에 담긴 실패 정보를 추출해서 출력한다.

아이템 40. @Override 애너테이션을 일관되게 사용하라.

- @Override 애너테이션은 상위 타입의 메서드를 재정의했음을 나타낸다.
- 메서드 선언에만 달 수 있으며, 일관되게 사용하면 여러 가지 버그들을 예방할 수 있다.

@Override의 장점

- @Override 애너테이션을 달아줌으로써 '재정의한다.'라는 의도를 명확히 나타낼 수 있다.
- 코드를 잘못 작성(Overriding이 아닌 Overloading)했을 경우 컴파일러가 잘못된 부분을 명확히 알려준다.

@Override를 달지 않아도 되는 경우

- 구체 클래스에서 상위 클래스의 추상 메서드를 재정의할 때는 @Override를 달지 않아도 된다.
- 구체 클래스인데 아직 구현하지 않은 추상 메서드가 있다면 컴파일러가 이를 알려주기 때문이다.

웬만하면 @Override를 달자

- 실수로 추가한 메서드가 있는지 확인하기 위해 추상 클래스나 인터페이스에서는 상위 클래스나 상위 인터페이스의 메서드를 재정의하는 모든 메서드에 @Override를 붙여주는 게 좋다.