아이템38 (확장할 수 있는 열거 타입이 필요하면 인터페이스를 사용하라)

타입 안전 열거 패턴

```
class Dir {
    static final int DIR_NORTH = 0;
    static final int DIR_WEST = 1;
    static final int DIR_EAST = 2;
    static final int DIR_SOUTH = 3;
}
```

타입 안전 열거 패턴은 확장할 수 있으나 열거 타입은 확장이 불가능하다. 즉 타입 안전 열거 패턴은 열거한 값들을 그대로 가져온 다음 값을 더 추가하여 다른 목적으로 쓸 수 있지만 열거타입은 불가능하다.

열거타입 상속

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

enum Dir2 extends Dir{

}
```

Dir 상속하는 Dir2에 에러가 발생해 상속이 불가능하다.

실수로 이렇게 설계한 것은 아니고 대부분 상황에서 열거 타입을 확장하는 것은 좋지 않은 생각이다.

확장이 좋지 않은 이유

1. 확장한 타입의 원소는 기반 타입의 원소로 취급하지만 그 반대는 성립하지 않는다면 이상하다.
2. 기반 타입과 확장된 타입들의 원소를 모두 순회할 방법도 마땅치 않다.
3. 확장성을 높이려면 고려할 요소가 늘어나 설계와 구현이 더 복잡해진다.

하지만 연산 코드 같은 경우 API가 제공하는 기본 연산 외에 사용자가 확장 연산을 추가할 수 있도록 열어줘야 할 때가 있는데 열거 타입으로 이를 해결이 가능하다. 방법은 열거 타입이 임의의 인터페이스를 구현할 수 있다는 것을 이용하면 된다. 연산 코드용 인터페이스를 정의하고 열거 타입이 이 인터페이스를 구현하게 하면 된다. 이때 열거 타입은 인터페이스의 표준 구현체 역할을 한다.

인터페이스를 사용해 확장한 열거타입 구현

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

enum ExtendOperation implements Operation{

    PLUS( s: "+"){
        public double apply(double x, double y){
            return x + y;
        }
    },MINUS( s: "-"){
        public double apply(double x, double y){
            return x + y;
        }
    };

    private final String symbol;

    ExtendOperation(String s) {
        this.symbol = s;
    }
}
```

열거 타입인 ExtendOperation은 확장할 수 없지만 Operation 인터페이스는 확장할 수 있고 이 인터페이스를 연산 타입으로 사용하면 된다.

Class를 넘겨 인터페이스를 확장한 열거타입 테스트하기

```
public class Main {
    public static void main(String[] args) {
        test(ExtendOperation.class,  x: 2,  y: 3);
    }

    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 ExtendedOperation의 class 리터럴을 넘겨 확장되는 연산들이 무엇인지 알려준다. 여기서 class 리터럴은 한정적 타입 토큰 역할을 한다.

```
<T extends Enum<T> & Operation>
```

이는 Class객체가 열거 타입인 동시에 Operation의 하위 타입이어야 한다는 뜻이다. 열거 타입이어야 원소를 순회할 수 있고 Operation이어야 원소가 뜻하는 연산을 수행할 수 있기 때문이다.

와일드카드 타입을 사용해 인터페이스를 확장한 열거타입 테스트하기

```
public class Main {
    public static void main(String[] args) {
        test(Arrays.asList(ExtendOperation.values()),  x: 2,  y: 3);
    }

    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));
        }
    }
}
```

Collection 타입을 받아 여러 구현 타입의 연산을 조합해 호출할 수 있도록 테스트가 유연해졌다.

정리

열거 타입 자체는 확장할 수 없지만 **인터페이스와 그 인터페이스를 구현하는 기본 열거 타입을 함께 사용해** 같은 효과를 낼 수 있다. 이렇게 하면 클라이언트는 이 인터페이스를 구현해 자신만의 열거 타입을 만들 수 있다. 그리고 **API가 기본 열거 타입을 직접 명시하지 않고 인터페이스 기반으로 작성되었다면 기본 열거 타입의 인스턴스가 쓰이는 모든 곳을 새로 확장한 열거 타입의 인스턴스로 대체해 사용할 수 있다**. 하지만 인터페이스를 이용해 확장 가능한 열거 타입을 흉내 내는 방식에도 한가지 사소한 문제가 있는데 바로 **열거 타입끼리 구현 상속할 수 없다**는 점이다.