10-14 [Java - Enum]

소유자	종수 김
태그	

열거형 - ENUM

문자열과 타입 안전성 - 1

열거형 예제

1. 비즈니스 요구 사항
    a. 고객을 3등급으로 나누고, 상품 구매시 등급별로 할인을 적용
        i. BASIC → 10% 할인
        ii. GOLD → 20% 할인
        iii. DIAMOND → 30% 할인

Ex) GOLD 유저가 10,000원 구매시 2,000원 할인.

```
package enumeration.ex0;

public class DiscountService {
    public int discount(String grade, int price) {
        int discountPercent = 0;

        if ("BASIC".equals(grade)) {
            discountPercent = 10;
        } else if ("GOLD".equals(grade)) {
            discountPercent = 20;
        } else if ("DIAMOND".equals(grade)) {
            discountPercent = 30;
        } else  {
            System.out.println(grade + " : 할인 X");
        }

        return price * discountPercent / 100;
```

```
    }
}


package enumeration.ex0;

public class StringGradeEx0_2 {
    public static void main(String[] args) {
        int price = 10000;

        DiscountService discountService = new DiscountService

        // 존재하지 않는 등급
        int vip = discountService.discount("VIP", price);
        System.out.println("vip = " + vip);

        // 오타
        int diamond = discountService.discount("DlAMOND", pri
        System.out.println("diamond = " + diamond);

        // 소문자 입력
        int gold = discountService.discount("gold", price);
        System.out.println("gold = " + gold);
    }
}
```

- 존재하지 않는 등급.
- 오타 발생.
- 소문자 입력.

타입 안정성 부족 : 문자열은 오타가 발생하기 쉽고, 유효하지 않은 값이 입력될 수 있음.

데이터 일관성 : “GOLD”, ‘gold”, “Gold”등 다양한 형식으로 문자열을 입력할 수 있어 일관성 X

String 사용 시 타입 안정성 부족 문제

- **값의 제한 부족**: `String`으로 상태나 카테고리를 표현하면, 잘못된 문자열을 실수로 입력할 가능성이 있다. 예를 들어, "Monday", "Tuesday" 등을 나타내는 데 `String`을 사용한다면, 오타("Munday")나 잘못된 값("Funday")이 입력될 위험이 있다.
- **컴파일 시 오류 감지 불가**: 이러한 잘못된 값은 컴파일 시에는 감지되지 않고, 런타임에서만 문제가 발견되기 때문에 디버깅이 어려워질 수 있다.

이런 문제를 해결하려면 특정 범위로 값을 제한해야 한다. 예를 들어 `BASIC`, `GOLD`, `DIAMOND`라는 정확한 문자만 `discount()` 메서드에 전달되어야 한다. 하지만 `String`은 어떤 문자열이든 받을 수 있기 때문에 자바 문법 관점에서는 아무런 문제가 없다. 결국 `String` 타입을 사용해서는 문제를 해결할 수 없다.

문자열과 타입 안정성 - 2

위에서 String은 오타, 소문자 입력, 존재하지 않는 등급 등 String을 통한 문제를 예방하기 위해 상수를 사용.

```
package enumeration.ex1;

public class StringGrade {
    public static final String BASIC = "BASIC";
    public static final String GOLD = "GOLD";
    public static final String DIAMOND = "DIAMOND";
}

package enumeration.ex1;

public class DiscountService {
    public int discount(String grade, int price) {
        int discountPercent = 0;

        if (StringGrade.BASIC.equals(grade)) {
            discountPercent = 10;
        } else if (StringGrade.GOLD.equals(grade)) {
            discountPercent = 20;
        } else if (StringGrade.DIAMOND.equals(grade)) {
            discountPercent = 30;
        } else  {
            System.out.println(grade + " : 할인 X");
        }
```

```
        return price * discountPercent / 100;
    }
}

package enumeration.ex1;


public class StringGradeEx1_1 {
    public static void main(String[] args) {
        int price = 10000;

        DiscountService discountService = new DiscountService
        int basic = discountService.discount(StringGrade.BASI
        int gold = discountService.discount(StringGrade.GOLD,
        int diamond = discountService.discount(StringGrade.DI

        System.out.println("basic = " + basic);
        System.out.println("gold = " + gold);
        System.out.println("diamond = " + diamond);
    }
}
```

- 문제를 해결 할 수 있는 것으로는 보이나, 해당 상수 사용은 근본적으로 해결할 순 없음 = 문자열을 직접 매개변수로 넘길 수 있기 때문(타입 안정성이 보장이 되지 않음)

> 문자열 상수를 사용한 덕분에 전체적으로 코드가 더 명확해졌다. 그리고 `discount()`에 인자를 전달할 때도 `StringGrade`가 제공하는 문자열 상수를 사용하면 된다. 더 좋은 점은 만약 실수로 상수의 이름을 잘못 입력하면 컴파일 시점에 오류가 발생한다는 점이다. 따라서 오류를 쉽고 빠르게 찾을 수 있다.
>
> 하지만 문자열 상수를 사용해도, 지금까지 발생한 문제들을 근본적으로 해결할 수 는 없다. 왜냐하면 `String` 타입은 어떤 문자열이든 입력할 수 있기 때문이다. 어떤 개발자가 실수로 `StringGrade`에 있는 문자열 상수를 사용하지 않고, 다음과 같이 직접 문자열을 사용해도 막을 수 있는 방법이 없다.

타입 안전 열거형 패턴

타입 안전 열거형 패턴 - Type-Safe Enum Patter

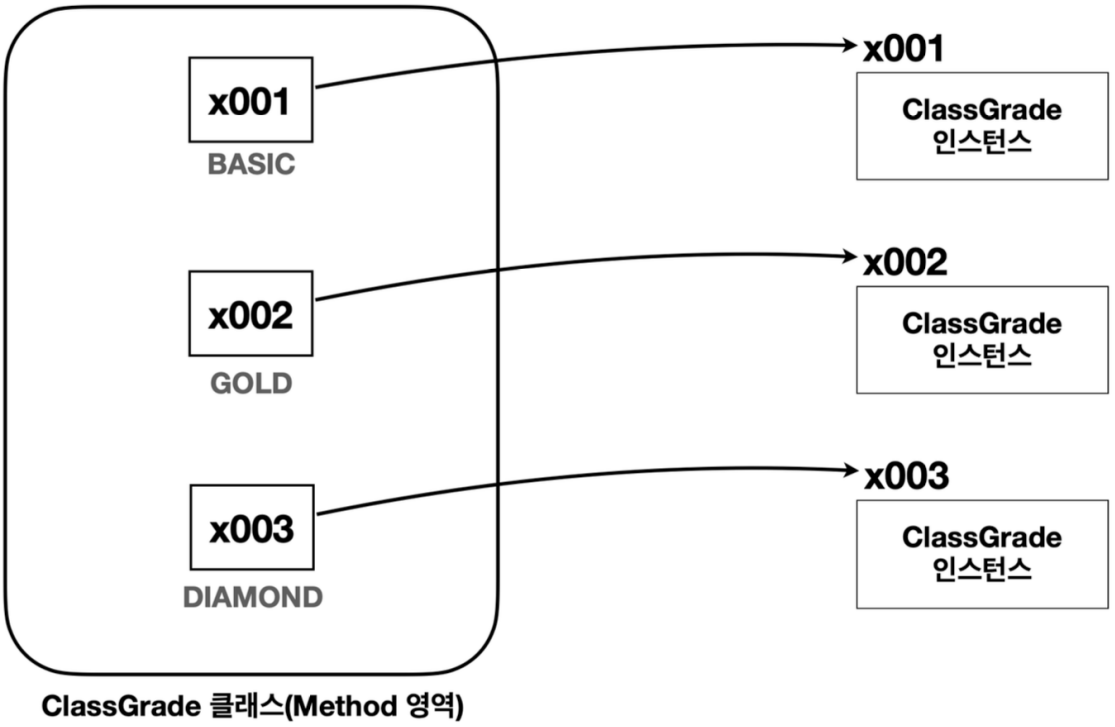
enum = enumeration = 열거 = 어떤 항목을 나열한다.

회원 등급인 BASIC, GOLD, DIAMOND를 나열하고, 타입 안전 열거형 패턴을 사용하여 나열한 항목만 사용할 수 있는 것이 핵심.

```
package enumeration.ex2;

public class ClassGrade {
    public static final ClassGrade BASIC = new ClassGrade();
    public static final ClassGrade GOLD = new ClassGrade();
    public static final ClassGrade DIAMOND = new ClassGrade()
}
```

- 먼저 회원 등급을 다루는 클래스를 만들고, 각각의 회원 등급별로 상수를 선언한다.
- 이때 각각의 상수마다 별도의 인스턴스를 생성하고, 생성한 인스턴스를 대입한다.
- 각각을 상수로 선언하기 위해 `static`, `final`을 사용한다.
  - `static`을 사용해서 상수를 메서드 영역에 선언한다.
  - `final`을 사용해서 인스턴스(참조값)를 변경할 수 없게 한다.




```
package enumeration.ex2;
```

```
public class ClassRefMain {
    public static void main(String[] args) {
        System.out.println("class BASIC = " + ClassGrade.BASI
        System.out.println("class GOLD = " + ClassGrade.GOLD.
        System.out.println("class DIAMOND = " + ClassGrade.BA

        System.out.println("ref BASIC = " + ClassGrade.BASIC)
        System.out.println("ref GOLD = " + ClassGrade.GOLD);
        System.out.println("ref DIAMOND = " + ClassGrade.DIAM
    }
}
```

실행 결과

```
class BASIC = class enumeration.ex2.ClassGrade
class GOLD = class enumeration.ex2.ClassGrade
class DIAMOND = class enumeration.ex2.ClassGrade
```

```
ref BASIC = enumeration.ex2.ClassGrade@x001
ref GOLD = enumeration.ex2.ClassGrade@x002
ref DIAMOND = enumeration.ex2.ClassGrade@x003
```

- 각각의 상수는 모두 `ClassGrade` 타입을 기반으로 인스턴스를 만들었기 때문에 `getClass()`의 결과는 모두 `ClassGrade`이다.
- 각각의 상수는 모두 서로 각각 다른 `ClassGrade` 인스턴스를 참조하기 때문에 참조값이 다르게 출력된다.

`static`이므로 애플리케이션 로딩 시점에 다음과 같이 3개의 `ClassGrade` 인스턴스가 생성되고, 각각의 상수는 같은 `ClassGrade` 타입의 서로 다른 인스턴스의 참조값을 가진다.

- `ClassGrade BASIC: x001`
- `ClassGrade GOLD: x002`
- `ClassGrade DIAMOND: x003`

여기서 `BASIC`, `GOLD`, `DIAMOND`를 상수로 열거했다. 이제 `ClassGrade` 타입을 사용할 때는 앞서 열거한 상수들만 사용하면 된다.

ClassGrade를 통해 타입 안정성을 확보할 수 있었고, static(상수)는 애플리케이션 로딩 시점에 메서드 영역에 인스턴스가 생성되므로 고유의 값을 가진다.

이를 통해, ClassGrade라는 타입 안정성을 확보할 수 있으며, ClassGrade내에 정의된 값만 사용할 수 있음.

Private 생성자

하지만 위 방법에도 문제가 있음.

```
package enumeration.ex2;

public class ClassGradeEx2_2 {
    public static void main(String[] args) {
        int price = 10000;
        DiscountService discountService = new DiscountServi
        ClassGrade classGrade = new ClassGrade();
        int result = discountService.discount(classGrade, p
        System.out.println("result = " + result);
    }
}
```

new를 통해 객체를 생성할 수 있음.

```
package enumeration.ex2;

public class ClassGrade {
    public static final ClassGrade BASIC = new ClassGrade()
    public static final ClassGrade GOLD = new ClassGrade();
    public static final ClassGrade DIAMOND = new ClassGrade

    private ClassGrade() {
    }
}
```

- `private` 생성자를 사용해서 외부에서 `ClassGrade`를 임의로 생성하지 못하게 막았다.
- `private` 생성자 덕분에 `ClassGrade`의 인스턴스를 생성하는 것은 `ClassGrade` 클래스 내부에서만 할 수 있다. 앞서 우리가 정의한 상수들은 `ClassGrade` 클래스 내부에서 `ClassGrade` 객체를 생성한다.
- 이제 `ClassGrade` 인스턴스를 사용할 때는 `ClassGrade` 내부에 정의한 상수를 사용해야 한다. 그렇지 않으면 컴파일 오류가 발생한다.
- 쉽게 이야기해서 `ClassGrade` 타입에 값을 전달할 때는 우리가 앞서 열거한 `BASIC`, `GOLD`, `DIAMOND` 상수만 사용할 수 있다.

이렇게 `private` 생성자까지 사용하면 타입 안전 열거형 패턴을 완성할 수 있다.

타입 안전 열거형 패턴"(Type-Safe Enum Pattern)의 장점

- **타입 안정성 향상**: 정해진 객체만 사용할 수 있기 때문에, 잘못된 값을 입력하는 문제를 근본적으로 방지할 수 있다.
- **데이터 일관성**: 정해진 객체만 사용하므로 데이터의 일관성이 보장된다.

조금 더 자세히

- **제한된 인스턴스 생성**: 클래스는 사전에 정의된 몇 개의 인스턴스만 생성하고, 외부에서는 이 인스턴스들만 사용할 수 있도록 한다. 이를 통해 미리 정의된 값들만 사용하도록 보장한다.
- **타입 안전성**: 이 패턴을 사용하면, 잘못된 값이 할당되거나 사용되는 것을 컴파일 시점에 방지할 수 있다. 예를 들어, 특정 메서드가 특정 열거형 타입의 값을 요구한다면, 오직 그 타입의 인스턴스만 전달할 수 있다. 여기서는 메서드의 매개변수로 `ClassGrade`를 사용하는 경우, 앞서 열거한 `BASIC`, `GOLD`, `DIAMOND`만 사용할 수 있다.

단점

이 패턴을 구현하려면 다음과 같이 많은 코드를 작성해야 한다. 그리고 `private` 생성자를 추가하는 등 유의해야 하는 부분들도 있다.

열거형 - Enum Type

타입 안전 열거형 패턴을 쉽게 사용할 수 있는 열거형(Enum Type)을 제공.

타입 안전 열거형 패턴을 프로그래밍 언어 차원에서 지원하는 것.

```
package enumeration.ex3;

public enum Grade {
    BASIC, GOLD, DIAMOND
}

// 상, 하 같은 코드.

package enumeration.ex2;

public class ClassGrade {
```

```
    public static final ClassGrade BASIC = new ClassGrade();
    public static final ClassGrade GOLD = new ClassGrade();
    public static final ClassGrade DIAMOND = new ClassGrade()

    private ClassGrade() {
    }
}
```

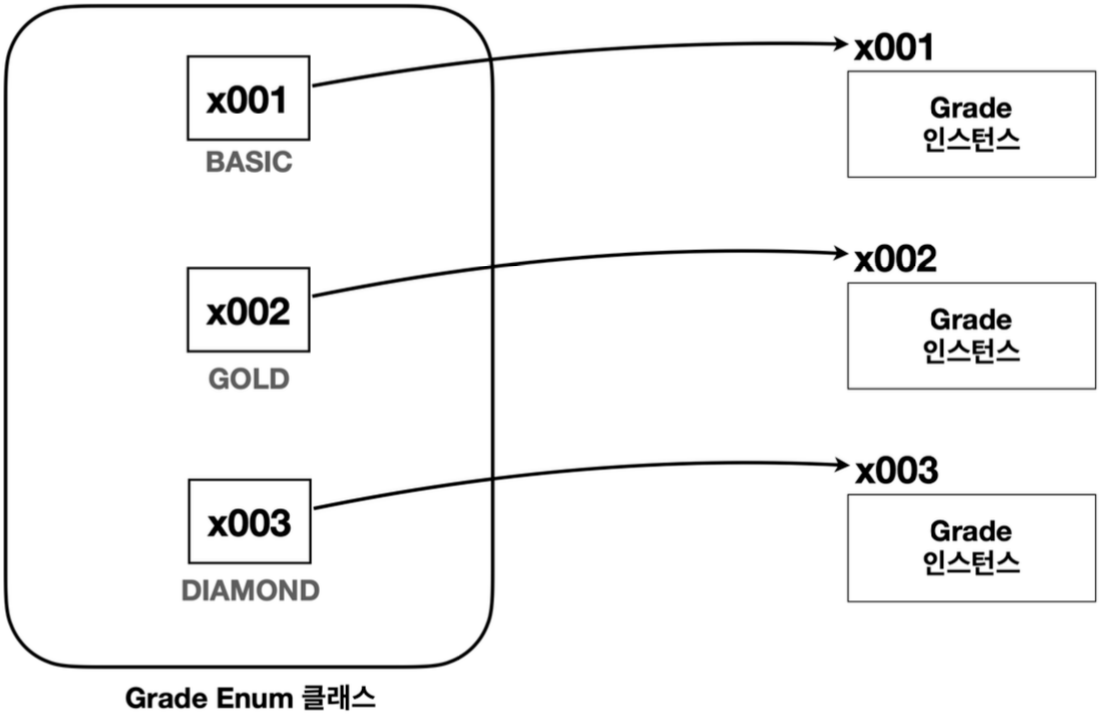
1. 열거형도 클래스이다. (상수만을 가진 클래스)
2. 열거형은 자동으로 java.lang.Enum을 상속받음.
3. 외부에서 임의로 직접 생성할 수 없음.(생성자가 private)

```
package enumeration.ex3;

public class EnumRefMain {
    public static void main(String[] args) {
        System.out.println("class BASIC = " + Grade.BASIC.get
        System.out.println("class GOLD = " + Grade.GOLD.getCl
        System.out.println("class DIAMOND = " + Grade.DIAMOND

        System.out.println("ref BASIC = " + refValue(Grade.BA
        System.out.println("ref GOLD = " + refValue(Grade.GOL
        System.out.println("ref DIAMOND = " + refValue(Grade.
    }

    private static String refValue(Object grade) {
        return Integer.toHexString(System.identityHashCode(gr
    }
}
```




```
class BASIC = class enumeration.ex3.Grade
class GOLD = class enumeration.ex3.Grade
class DIAMOND = class enumeration.ex3.Grade

ref BASIC = x001
ref GOLD = x002
ref DIAMOND = x003
```

- 실행 결과를 보면 상수들이 열거형으로 선언한 타입인 `Grade` 타입을 사용하는 것을 확인할 수 있다. 그리고 각각의 인스턴스도 서로 다른 것을 확인할 수 있다.
- 참고로 열거형은 `toString()`을 재정의 하기 때문에 참조값을 직접 확인할 수 없다. 참조값을 구하기 위해 `refValue()`를 만들었다.
  - `System.identityHashCode(grade)` : 자바가 관리하는 객체의 참조값을 숫자로 반환한다.
  - `Integer.toHexString()` : 숫자를 16진수로 변환, 우리가 일반적으로 확인하는 참조값은 16진수
- 열거형도 클래스이다. 열거형을 제공하기 위해 제약이 추가된 클래스라 생각하면 된다.

열거형(ENUM)의 장점

1. 타입 안정성 향상 : 열거형은 사전에 정의된 상수들로만 구성되므로, 유효하지 않은 값이 입력될 가능성이 없으며 이런 경우 컴파일 오류가 발생.

2. 간결성 및 일관성 : 열거형을 사용하면 코드가 더 간결하고 명확해지며, 데이터의 일관성이 보장됨.
3. 확장성 : 새로운 회원 등급을 타입을 추가하고 싶을 때, ENUM에 새로운 상수를 추가하기만 하면 됨.

열거형 - 주요 메서드

모든 열거형은 java.lang.Enum 클래스를 상속받음.

```
package enumeration.ex3;

import java.util.Arrays;

public class EnumMethodMain {
    public static void main(String[] args) {
        // 모든 ENUM 반환
        Grade[] values = Grade.values();
        System.out.println("values = " + Arrays.toString(valu
        for(Grade grade : values) {
            System.out.println("name = " + grade.name() + ",
        }

        //String -> ENUM 변환
        String input = "DIAMOND";
        Grade gold = Grade.valueOf(input);
        System.out.println("gold = " + gold);

    }
}
```

ENUM - 주요 메서드

- **values()**: 모든 ENUM 상수를 포함하는 배열을 반환한다.
- **valueOf(String name)**: 주어진 이름과 일치하는 ENUM 상수를 반환한다.
- **name()**: ENUM 상수의 이름을 문자열로 반환한다.
- **ordinal()**: ENUM 상수의 선언 순서(0부터 시작)를 반환한다.
- **toString()**: ENUM 상수의 이름을 문자열로 반환한다. `name()` 메서드와 유사하지만, `toString()`은 직접 오버라이드 할 수 있다.

주의 ordinal()은 가급적 사용하지 않는 것이 좋다.

- `ordinal()`의 값은 가급적 사용하지 않는 것이 좋다. 왜냐하면 이 값을 사용하다가 중간에 상수를 선언하는 위치가 변경되면 전체 상수의 위치가 모두 변경될 수 있기 때문이다.
- 예를 들어 중간에 `BASIC` 다음에 `SILVER` 등급이 추가되는 경우 `GOLD`, `DIAMOND`의 값이 하나씩 추가된다.

기존

- `BASIC: 0`
- `GOLD: 1`
- `DIAMOND: 2`

추가

- `BASIC: 0`
- `SILVER: 1`
- `GOLD: 2`
- `DIAMOND: 3`

기존 `GOLD`의 `ordinal()` 값인 `1`을 데이터베이스나 파일에 저장하고 있었는데, 중간에 `SILVER`가 추가되면 데이터베이스나 파일에 있는 값은 그대로 1로 유지되지만, 애플리케이션 상에서 `GOLD`는 2가 되고, `SILVER`는 1이 된다. 쉽게 이야기해서 `ordinal()`의 값을 사용하면 기존 `GOLD` 회원이 갑자기 `SILVER`가 되는 큰 버그가 발생할 수 있다.

열거형 정리

1. 열거형은 java.lang.Enum을 자동으로 상속 받는다.
2. 열거형은 이미 상속을 받았으므로 추가로 다른 클래스를 상속 받을 수 없음.
3. 열거형은 인터페이스를 구현할 수 있다.
4. 열거형에 추상 메서드를 선언하고, 구현할 수 있다.
    a. 이 경우 익명클래스를 사용해야 함.

열거형 - 리팩토링1

- 불필요한 `if`문을 제거하자.
- 이 코드에서 할인율(`discountPercent`)은 각각의 회원 등급별로 판단된다. 할인율은 결국 회원 등급을 따라간다. 따라서 회원 등급 클래스가 할인율(`discountPercent`)을 가지고 관리하도록 변경하자.

AS-IS

```
package enumeration.ex2;

public class ClassGrade {
    public static final ClassGrade BASIC = new ClassGrade();
    public static final ClassGrade GOLD = new ClassGrade();
    public static final ClassGrade DIAMOND = new ClassGrade()

    private ClassGrade() {
    }
}

package enumeration.ex2;


public class DiscountService {
    public int discount(ClassGrade classGrade, int price) {
        int discountPercent = 0;

        if (classGrade == ClassGrade.BASIC){
            discountPercent = 10;
        } else if (classGrade == ClassGrade.GOLD) {
            discountPercent = 20;
        } else if (classGrade == ClassGrade.DIAMOND) {
            discountPercent = 30;
        } else {
            System.out.println(" 할인 X");
        }

        return price * discountPercent / 100;
    }
}
```

TO-BE

```
package enumeration.ref1;

public class ClassGrade {
    public static final ClassGrade BASIC = new ClassGrade(10)
    public static final ClassGrade GOLD = new ClassGrade(20);
    public static final ClassGrade DIAMOND = new ClassGrade(3

    private final int discountPercent;

    private ClassGrade(int discountPercent) {
        this.discountPercent = discountPercent;
    }

    public int getDiscountPercent() {
        return discountPercent;
    }
}

package enumeration.ref1;

public class DiscountService {
    public int discount(ClassGrade classGrade, int price) {
        return price * classGrade.getDiscountPercent() / 100;
    }
}
```

- `ClassGrade`에 할인율(`discountPercent`) 필드를 추가했다. 조회 메서드도 추가한다.
- 생성자를 통해서만 `discountPercent`를 설정하도록 했고, 중간에 이 값이 변하지 않도록 불변으로 설계했다.
- 정리하면 상수를 정의할 때 각각의 등급에 따른 할인율(`discountPercent`)이 정해진다.

- 기존에 있던 `if` 문이 완전히 제거되고, 단순한 할인율 계산 로직만 남았다.
- 기존에는 `if` 문을 통해서 회원의 등급을 찾고, 각 등급 별로 `discountPercent` 의 값을 지정했다.
- 변경된 코드에서는 `if` 문을 사용할 이유가 없다. 단순히 회원 등급안에 있는 `getDiscountPercent()` 메서드를 호출하면 인수로 넘어온 회원 등급의 할인율을 바로 구할 수 있다.

열거형 - 리팩토링 2

열거형을 사용하여 리팩토링

AS-IS

```
package enumeration.ex2;

public class ClassGrade {
    public static final ClassGrade BASIC = new ClassGrade();
    public static final ClassGrade GOLD = new ClassGrade();
    public static final ClassGrade DIAMOND = new ClassGrade()

    private ClassGrade() {
    }
}

package enumeration.ex2;


public class DiscountService {
    public int discount(ClassGrade classGrade, int price) {
        int discountPercent = 0;

        if (classGrade == ClassGrade.BASIC){
            discountPercent = 10;
        } else if (classGrade == ClassGrade.GOLD) {
            discountPercent = 20;
        } else if (classGrade == ClassGrade.DIAMOND) {
            discountPercent = 30;
        } else {
            System.out.println(" 할인 X");
        }
```

```
        return price * discountPercent / 100;
    }
}
```

TO-BE

```
package enumeration.ref2;

public enum Grade {
    BASIC(10)
    ,GOLD(20)
    ,DIAMOND(30)
    ;

    private final int discountPercent;

    public int getDiscountPercent() {
        return discountPercent;
    }

    private Grade(int discountPercent) {
        this.discountPercent = discountPercent;
    }
}

package enumeration.ref2;

public class DiscountService {
    public int discount(Grade grade, int price) {
        return price * grade.getDiscountPercent() / 100;
    }
}
```

- `discountPercent` 필드를 추가하고, 생성자를 통해서 필드에 값을 저장한다.
- 열거형은 상수로 지정하는 것 외에 일반적인 방법으로 생성이 불가능하다. 따라서 생성자에 접근제어자를 선언할 수 없게 막혀있다. `private`이라고 생각하면 된다.
- `BASIC(10)`과 같이 상수 마지막에 괄호를 열고 생성자에 맞는 인수를 전달하면 적절한 생성자가 호출된다.
- 값을 조회하기 위해 `getDiscountPercent()` 메서드를 추가했다. 열거형도 클래스이므로 메서드를 추가할 수 있다.

- Grade라는 객체에, BASIC이라는 이름을 가진 인스턴스가 있고, 멤버 변수로 discountPercent가 있다.

열거형 - 리팩토링3

- 이 코드를 보면 할인율 계산을 위해 `Grade`가 가지고 있는 데이터인 `discountPercent`의 값을 꺼내서 사용한다.
- 결국 `Grade`의 데이터인 `discountPercent`를 할인율 계산에 사용한다.
- 객체지향 관점에서 이렇게 자신의 데이터를 외부에 노출하는 것 보다는, `Grade` 클래스가 자신의 할인율을 어떻게 계산하는지 스스로 관리하는 것이 캡슐화 원칙에 더 맞다.

단순 할인율 계산

AS-IS

```
public class DiscountService {
    public int discount(Grade grade, int price) {
        return price * grade.getDiscountPercent() / 100;
    }
}
```

TO-BE

```
package enumeration.ref3;

public enum Grade {
    BASIC(10)
    ,GOLD(20)
    ,DIAMOND(30)
    ;
```

```
    private final int discountPercent;

    public int getDiscountPercent() {
        return discountPercent;
    }

    private Grade(int discountPercent) {
        this.discountPercent = discountPercent;
    }

    public int discount(int price) {
        return price * discountPercent / 100;
    }
}


package enumeration.ref3;

public class DiscountService {
    public int discount(Grade grade, int price) {
        return grade.discount(price);
    }
}
```

- Grade가 객체지향 캡슐화 원칙에 맞추어 본인의 상태를 직접 객체 스스로가 변경하고 있음.
  때문에 DiscountService는 더 간략화 되고, 필요가 없어짐.

TO-BE

```
package enumeration.ref3;


public class ClassGradeRefMain3_2 {
    public static void main(String[] args) {
        int price = 10000;

        System.out.println("basic = " + Grade.BASIC.discount(
        System.out.println("gold = " + Grade.GOLD.discount(pr
```

```
        System.out.println("diamond = " + Grade.DIAMOND.disco
    }
}

// DiscountService가 사라짐.
```

중복 제거 TO-BE

```
package enumeration.ref3;


public class ClassGradeRefMain3_3 {
    public static void main(String[] args) {
        int price = 10000;

        printDiscount(Grade.BASIC, price);
        printDiscount(Grade.GOLD, price);
        printDiscount(Grade.DIAMOND, price);
    }
    public static void printDiscount(Grade grade, int price)
        System.out.println(grade.name() + " 등급의 할인 금액 " +
    }
}
```

등급 추가 시 변경 없이 출력

```
package enumeration.ref3;


import java.util.Arrays;

public class ClassGradeRefMain3_4 {
    public static void main(String[] args) {
        int price = 10000;

        Grade[] values = Grade.values();
        Arrays.stream(values)
                .forEach(o -> printDiscount(o, price));
```

```
    }
    public static void printDiscount(Grade grade, int price)
        System.out.println(grade.name() + " 등급의 할인 금액 " +
    }
}
```