5. 열거형 - ENUM

#1.인강/0.자바/3.자바-중급1편

- /문자열과 타입 안전성1
- /문자열과 타입 안전성2
- /타입 안전 열거형 패턴
- /열거형 - Enum Type
- /열거형 - 주요 메서드
- /열거형 - 리팩토링1
- /열거형 - 리팩토링2
- /열거형 - 리팩토링3
- /문제와 풀이1
- /문제와 풀이2
- /정리

문자열과 타입 안전성1

자바가 제공하는 열거형(Enum Type)을 제대로 이해하려면 먼저 열거형이 생겨난 이유를 알아야 한다. 예제를 순서대로 따라가며 열거형이 만들어진 근본적인 이유를 알아보자.

비즈니스 요구사항

고객은 3등급으로 나누고, 상품 구매시 등급별로 할인을 적용한다. 할인시 소수점 이하는 버린다.

- `BASIC` ➜ 10% 할인
- `GOLD` ➜ 20% 할인
- `DIAMOND` ➜ 30% 할인

예) `GOLD` 유저가 10000원을 구매하면 할인 대상 금액은 2000원이다.

예제를 구현해보자.
회원 등급과 가격을 입력하면 할인 금액을 계산해주는 클래스를 만들어보자.
예를 들어서 `GOLD`, 10000원을 입력하면 할인 대상 금액인 2000원을 반환한다.

```
package enumeration.ex0;

public class DiscountService {
```

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

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

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

- `price * discountPercent / 100`: 가격 * 할인율 / 100을 계산하면 할인 금액을 구할 수 있다.
- 회원 등급 외 다른 값이 입력되면 `할인X`를 출력한다. 이 경우 `discountPercent`가 `0`이므로 할인 금액도 `0`원으로 계산된다.
- 예제를 단순화하기 위해 회원 등급에 `null`은 입력되지 않는다고 가정한다.

```
package enumeration.ex0;

public class StringGradeEx0_1 {

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

        DiscountService discountService = new DiscountService();
        int basic = discountService.discount("BASIC", price);
        int gold = discountService.discount("GOLD", price);
        int diamond = discountService.discount("DIAMOND", price);

        System.out.println("BASIC 등급의 할인 금액: " + basic);
        System.out.println("GOLD 등급의 할인 금액: " + gold);
        System.out.println("DIAMOND 등급의 할인 금액: " + diamond);
    }
}
```

실행 결과

```
BASIC 등급의 할인 금액: 1000
GOLD 등급의 할인 금액: 2000
DIAMOND 등급의 할인 금액: 3000
```

실행 결과를 보면 각각의 회원 등급에 맞는 할인이 적용된 것을 확인할 수 있다.

그런데 지금과 같이 단순히 문자열을 입력하는 방식은, 오타가 발생하기 쉽고, 유효하지 않는 값이 입력될 수 있다.
다음 예를 보자.

```
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 등급의 할인 금액: " + vip);

        // 오타
        int diamondd = discountService.discount("DIAMONDD", price);
        System.out.println("DIAMONDD 등급의 할인 금액: " + diamondd);

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

실행 결과

```
VIP: 할인X
VIP 등급의 할인 금액: 0

DIAMONDD: 할인X
DIAMONDD 등급의 할인 금액: 0

gold: 할인X
gold 등급의 할인 금액: 0
```

예제에서는 다음과 같은 문제가 발생했다.

- 존재하지 않는 VIP라는 등급을 입력했다.
- 오타: DIAMOND 마지막에 D가 하나 추가되었다.
- 소문자 입력: 등급은 모두 대문자인데, 소문자를 입력했다.

등급에 문자열을 사용하는 지금의 방식은 다음과 같은 문제가 있다.

- **타입 안정성 부족**: 문자열은 오타가 발생하기 쉽고, 유효하지 않은 값이 입력될 수 있다.
- **데이터 일관성**: "GOLD", "gold", "Gold" 등 다양한 형식으로 문자열을 입력할 수 있어 일관성이 떨어진다.

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 안전성2

이번에는 대안으로 문자열 상수를 사용해보자. 상수는 미리 정의한 변수명을 사용할 수 있기 때문에 문자열을 직접 사용하는 것 보다는 더 안전하다.

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

        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.BASIC, price);
        int gold = discountService.discount(StringGrade.GOLD, price);
        int diamond = discountService.discount(StringGrade.DIAMOND, price);

        System.out.println("BASIC 등급의 할인 금액: " + basic);
        System.out.println("GOLD 등급의 할인 금액: " + gold);
        System.out.println("DIAMOND 등급의 할인 금액: " + diamond);
    }
}
```

실행 결과

```
BASIC 등급의 할인 금액: 1000
GOLD 등급의 할인 금액: 2000
DIAMOND 등급의 할인 금액: 3000
```

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.

```
package enumeration.ex1;

public class StringGradeEx1_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 등급의 할인 금액: " + vip);

        // 오타
        int diamondd = discountService.discount("DIAMONDD", price);
        System.out.println("DIAMONDD 등급의 할인 금액: " + diamondd);

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

실행 결과

```
VIP: 할인X
VIP 등급의 할인 금액: 0
DIAMONDD: 할인X
DIAMONDD 등급의 할인 금액: 0
gold: 할인X
gold 등급의 할인 금액: 0
```

그리고 사용해야 하는 문자열 상수가 어디에 있는지 `discount()`를 호출하는 개발자가 어떻게 알 수 있을까? 다음 코드를 보면 분명 `String`은 다 입력할 수 있다고 되어있다.

```
public int discount(String grade, int price) {}
```

결국 누군가 주석을 잘 남겨두어서, `StringGrade`에 있는 상수를 사용해달라고 해야 한다. 물론 이렇게 해도 누군가는 주석을 깜박하고 문자열을 직접 입력할 수 있다.

타입 안전 열거형 패턴

타입 안전 열거형 패턴 - Type-Safe Enum Pattern

지금까지 설명한 문제를 해결하기 위해 많은 개발자들이 오랜기간 고민하고 나온 결과가 바로 타입 안전 열거형 패턴이다.

여기서 영어인 `enum`은 `enumeration`의 줄임말인데, 번역하면 열거라는 뜻이고, 어떤 항목을 나열하는 것을 뜻한다. 우리의 경우 회원 등급인 `BASIC`, `GOLD`, `DIAMOND`를 나열하는 것이다. 여기서 중요한 것은 타입 안전 열거형 패턴을 사용하면 이렇게 나열한 항목만 사용할 수 있다는 것이 핵심이다. 나열한 항목이 아닌 것은 사용할 수 없다.

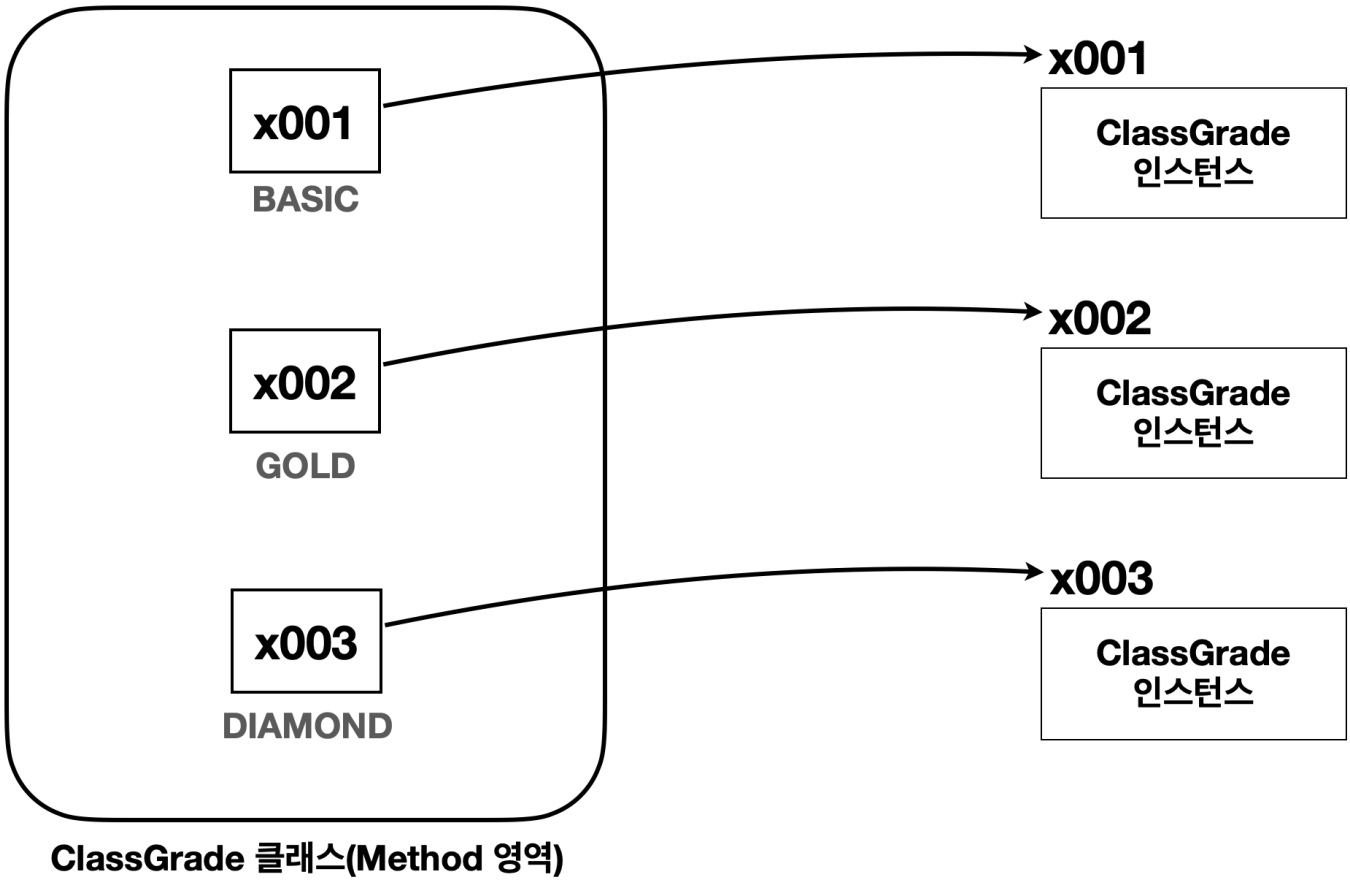
쉽게 이야기해서 앞서본 `String`처럼 아무런 문자열이나 다 사용할 수 있는 것이 아니라, 우리가 나열한 항목인 `BASIC`, `GOLD`, `DIAMOND`만 안전하게 사용할 수 있다는 것이다.

타입 안전 열거형 패턴을 직접 구현해보자.

```
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.



이 코드를 확실히 이해하기 위해 먼저 다음 코드를 실행해보자.

```
package enumeration.ex2;

public class ClassRefMain {

    public static void main(String[] args) {
        System.out.println("class BASIC = " + ClassGrade.BASIC.getClass());
        System.out.println("class GOLD = " + ClassGrade.GOLD.getClass());
        System.out.println("class DIAMOND = " + ClassGrade.DIAMOND.getClass());

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

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.

```
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("할인X");
        }

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

- `discount()` 메서드는 매개변수로 `ClassGrade` 클래스를 사용한다.
- 값을 비교할 때는 `classGrade == ClassGrade.BASIC`와 같이 `==` 참조값 비교를 사용하면 된다.
  - 매개변수에 넘어오는 인수도 `ClassGrade`가 가진 상수 중에 하나를 사용한다. 따라서 열거한 상수의 참조값으로 비교(`==`)하면 된다.

```java
package enumeration.ex2;

public class ClassGradeEx2_1 {

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

        DiscountService discountService = new DiscountService();
        int basic = discountService.discount(ClassGrade.BASIC, price);
        int gold = discountService.discount(ClassGrade.GOLD, price);
        int diamond = discountService.discount(ClassGrade.DIAMOND, price);

        System.out.println("BASIC 등급의 할인 금액: " + basic);
        System.out.println("GOLD 등급의 할인 금액: " + gold);
        System.out.println("DIAMOND 등급의 할인 금액: " + diamond);
    }
}
```

- `discount()`를 호출할 때 미리 정의한 `ClassGrade`의 상수를 전달한다.

실행 결과

```
BASIC 등급의 할인 금액: 1000
GOLD 등급의 할인 금액: 2000
DIAMOND 등급의 할인 금액: 3000
```

private 생성자

그런데 이 방식은 외부에서 임의로 `ClassGrade`의 인스턴스를 생성할 수 있다는 문제가 있다.

```java
package enumeration.ex2;

public class ClassGradeEx2_2 {

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

        DiscountService discountService = new DiscountService();

        ClassGrade newClassGrade = new ClassGrade(); //생성자 private으로 막아야 함
        int result = discountService.discount(newClassGrade, price);
        System.out.println("newClassGrade 등급의 할인 금액: " + result);
```

```
    }
}
```

실행 결과

```
할인X
newClassGrade 등급의 할인 금액: 0
```

이 문제를 해결하려면 외부에서 `ClassGrade`를 생성할 수 없도록 막으면 된다. 기본 생성자를 `private`으로 변경하자.

코드 변경

```
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 생성자 추가
    private ClassGrade() {}
}
```

코드 변경

```
package enumeration.ex2;

public class ClassGradeEx2_2 {

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

        DiscountService discountService = new DiscountService();
/*
        ClassGrade newClassGrade = new ClassGrade(); //생성자 private으로 막아야 함
        int result = discountService.discount(newClassGrade, price);
        System.out.println("newClassGrade 등급의 할인 금액: " + result);
*/
    }
}
```

- `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.

```
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 생성자 추가
    private ClassGrade() {}
}
```

열거형 - Enum Type

자바는 타입 안전 열거형 패턴"(Type-Safe Enum Pattern)을 매우 편리하게 사용할 수 있는 열거형(Enum Type)을 제공한다.
쉽게 이야기해서 자바의 열거형은 앞서 배운 타입 안전 열거형 패턴을 쉽게 사용할 수 있도록 프로그래밍 언어에서 지원하는 것이다.

영어인 `enum`은 `enumeration`의 줄임말인데, 번역하면 열거라는 뜻이고, 어떤 항목을 나열하는 것을 뜻한다.
"Enumeration"은 일련의 명명된 상수들의 집합을 정의하는 것을 의미하며, 프로그래밍에서는 이러한 상수들을 사용하여 코드 내에서 미리 정의된 값들의 집합을 나타낸다.
쉽게 이야기해서 회원의 등급은 상수로 정의한 `BASIC`, `GOLD`, `DIAMOND`만 사용할 수 있다는 뜻이다.
자바의 enum은 타입 안전성을 제공하고, 코드의 가독성을 높이며, 예상 가능한 값들의 집합을 표현하는 데 사용된다.

```
package enumeration.ex3;

public enum Grade {
    BASIC, GOLD, DIAMOND
}
```

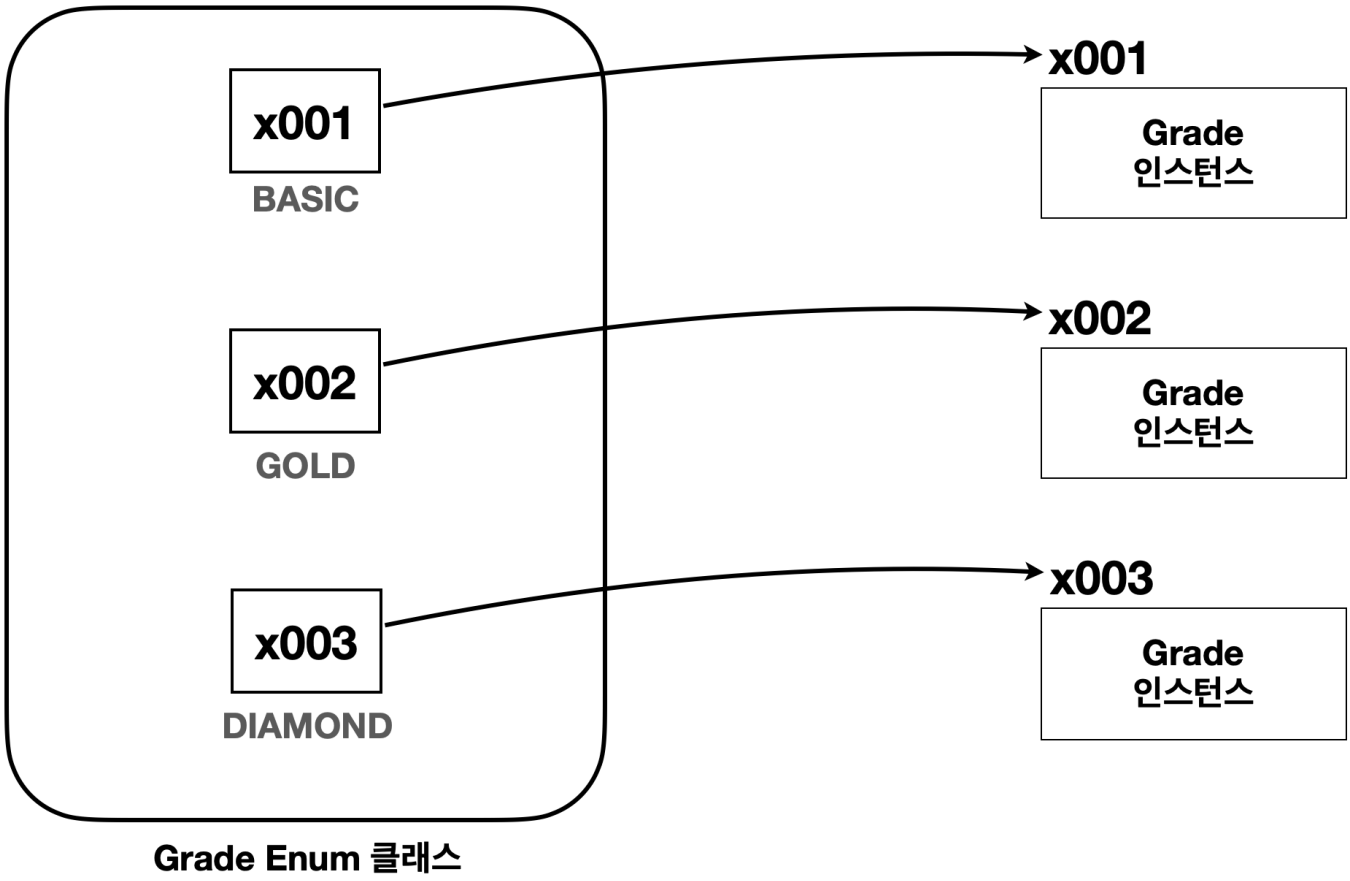
- 열거형을 정의할 때는 `class` 대신에 `enum`을 사용한다.
- 원하는 상수의 이름을 나열하면 된다.

앞서 직접 `ClassGrade`를 구현할 때와는 비교가 되지 않을 정도로 편리하다.
자바의 열거형으로 작성한 `Grade`는 다음 코드와 거의 같다.

```
public class Grade extends Enum {
    public static final Grade BASIC = new Grade();
    public static final Grade GOLD = new Grade();
    public static final Grade DIAMOND = new Grade();

    //private 생성자 추가
    private Grade() {}
}
```

- 열거형도 클래스이다.
- 열거형은 자동으로 `java.lang.Enum`을 상속 받는다.
- 외부에서 임의로 생성할 수 없다.



Grade Enum 클래스

열거형을 코드로 확인해보자.

```
package enumeration.ex3;

public class EnumRefMain {

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

        System.out.println("ref BASIC = " + refValue(Grade.BASIC));
        System.out.println("ref GOLD = " + refValue(Grade.GOLD));
        System.out.println("ref DIAMOND = " + refValue(Grade.DIAMOND));
    }

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

실행 결과

```
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.

자바의 열거형을 사용해서 코드를 작성해보자.

```
package enumeration.ex3;

public class DiscountService {

    public int discount(Grade grade, int price) {
        int discountPercent = 0;

        //enum switch 변경 가능
        if (grade == Grade.BASIC) {
            discountPercent = 10;
        } else if (grade == Grade.GOLD) {
            discountPercent = 20;
        } else if (grade == Grade.DIAMOND) {
            discountPercent = 30;
        } else {
            System.out.println("할인X");
        }

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

```
package enumeration.ex3;
```

```
public class EnumEx3_1 {
    public static void main(String[] args) {
        int price = 10000;
        DiscountService discountService = new DiscountService();
        int basic = discountService.discount(Grade.BASIC, price);
        int gold = discountService.discount(Grade.GOLD, price);
        int diamond = discountService.discount(Grade.DIAMOND, price);

        System.out.println("BASIC 등급의 할인 금액: " + basic);
        System.out.println("GOLD 등급의 할인 금액: " + gold);
        System.out.println("DIAMOND 등급의 할인 금액: " + diamond);
    }
}
```

실행 결과

```
BASIC 등급의 할인 금액: 1000
GOLD 등급의 할인 금액: 2000
DIAMOND 등급의 할인 금액: 3000
```

- 열거형의 사용법이 앞서 타입 안전 열거형 패턴을 직접 구현한 코드와 같은 것을 확인 할 수 있다.
- 참고로 열거형은 switch 문에 사용할 수 있는 장점도 있다.

열거형은 외부 생성 불가

```
package enumeration.ex3;

public class EnumEx3_2 {
    public static void main(String[] args) {
        int price = 10000;
        DiscountService discountService = new DiscountService();

/*
        Grade myGrade = new Grade(); //enum 생성 불가
        double result = discountService.discount(myGrade, price);
        System.out.println("result price: " + result);
*/
    }
}
```

- `enum`은 열거형 내부에서 상수로 지정하는 것 외에 직접 생성이 불가능하다. 생성할 경우 컴파일 오류가 발생한다.
    - 오류 메시지: `enum classes may not be instantiated`

열거형(ENUM)의 장점

- **타입 안정성 향상**: 열거형은 사전에 정의된 상수들로만 구성되므로, 유효하지 않은 값이 입력될 가능성이 없다. 이런 경우 컴파일 오류가 발생한다.
- **간결성 및 일관성**: 열거형을 사용하면 코드가 더 간결하고 명확해지며, 데이터의 일관성이 보장된다.
- **확장성**: 새로운 회원 등급을 타입을 추가하고 싶을 때, ENUM에 새로운 상수를 추가하기만 하면 된다.

참고

열거형을 사용하는 경우 `static import`를 적절하게 사용하면 더 읽기 좋은 코드를 만들 수 있다.

열거형 - 주요 메서드

모든 열거형은 `java.lang.Enum` 클래스를 자동으로 상속 받는다. 따라서 해당 클래스가 제공하는 기능들을 사용할 수 있다.

```
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(values));
        for (Grade value : values) {
            System.out.println("name=" + value.name() + ", ordinal=" +
value.ordinal());
        }

        //String -> ENUM 변환, 잘못된 문자면 IllegalArgumentException 발생
        String input = "GOLD";
        Grade gold = Grade.valueOf(input);
        System.out.println("gold = " + gold); //toString() 오버라이딩 가능
    }
}
```

실행 결과

```
values = [BASIC, GOLD, DIAMOND]
name=BASIC, ordinal=0
name=GOLD, ordinal=1
name=DIAMOND, ordinal=2
gold = GOLD
```

`Arrays.toString()` 배열의 참조값이 아니라 배열 내부의 값을 출력할 때 사용한다.

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 `SLIVER` 등급이 추가되는 경우 `GOLD`, `DIAMOND`의 값이 하나씩 추가된다.

기존

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

추가

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

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

열거형 정리

- 열거형은 `java.lang.Enum`를 자동(강제)으로 상속 받는다.
- 열거형은 이미 `java.lang.Enum`을 상속 받았기 때문에 추가로 다른 클래스를 상속을 받을 수 없다.
- 열거형은 인터페이스를 구현할 수 있다.
- 열거형에 추상 메서드를 선언하고, 구현할 수 있다.
  - 이 경우 익명 클래스와 같은 방식을 사용한다. 익명 클래스는 뒤에서 다룬다.

열거형 - 리팩토링1

이번 시간에는 지금까지 구현한 코드들을 더 읽기 쉽게 리팩토링해보자.
아직 열거형(ENUM)에 익숙하지 않으니, 앞서 클래스를 직접 사용해서 열거형 패턴을 구현했던 `ex2`의 코드를 먼저 리팩토링해보자.

`DiscountService.discount()` 코드를 살펴보자.

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

- 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.

```
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(30);

    private final int discountPercent;
```

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

    public int getDiscountPercent() {
        return discountPercent;
    }
}
```

- `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.

```
package enumeration.ref1;

public class DiscountService {

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

- 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.

```
package enumeration.ref1;

public class ClassGradeRefMain1 {

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

        DiscountService discountService = new DiscountService();
        int basic = discountService.discount(ClassGrade.BASIC, price);
        int gold = discountService.discount(ClassGrade.GOLD, price);
        int diamond = discountService.discount(ClassGrade.DIAMOND, price);

        System.out.println("BASIC 등급의 할인 금액: " + basic);
        System.out.println("GOLD 등급의 할인 금액: " + gold);
```

```
        System.out.println("DIAMOND 등급의 할인 금액: " + diamond);
    }
}
```

실행 결과

```
BASIC 등급의 할인 금액: 1000
GOLD 등급의 할인 금액: 2000
DIAMOND 등급의 할인 금액: 3000
```

실행 결과는 기존 코드와 같다.

열거형 - 리팩토링2

이제 열거형을 사용해보자.

열거형도 클래스이다. 앞서 했던 리팩토링을 열거형인 `Grade`에 동일하게 적용해보자.

```
package enumeration.ref2;

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

    private final int discountPercent;

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

    public int getDiscountPercent() {
        return discountPercent;
    }
}
```

- `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.

```
package enumeration.ref2;

public class DiscountService {

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

- 기존에 있던 `if` 문이 완전히 제거되고, 단순한 할인율 계산 로직만 남았다.

```
package enumeration.ref2;

public class EnumRefMain2 {
    public static void main(String[] args) {
        int price = 10000;
        DiscountService discountService = new DiscountService();
        int basic = discountService.discount(Grade.BASIC, price);
        int gold = discountService.discount(Grade.GOLD, price);
        int diamond = discountService.discount(Grade.DIAMOND, price);

        System.out.println("BASIC 등급의 할인 금액: " + basic);
        System.out.println("GOLD 등급의 할인 금액: " + gold);
        System.out.println("DIAMOND 등급의 할인 금액: " + diamond);
    }
}
```

실행 결과

```
BASIC 등급의 할인 금액: 1000
GOLD 등급의 할인 금액: 2000
DIAMOND 등급의 할인 금액: 3000
```

실행 결과는 기존과 같다.

열거형 - 리팩토링3

다음 코드를 리팩토링 하고 나니, 단순한 할인율 계산만 남았다.

```
public class DiscountService {

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

- 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.

`Grade` 클래스 안으로 `discount()` 메서드를 이동시키자. 일부 코드 수정이 필요하다.

```
package enumeration.ref3;

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

    private final int discountPercent;

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

    public int getDiscountPercent() {
        return discountPercent;
    }

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

}
```

- `Grade` 내부에 `discount()` 메서드를 만들어서, 이제 할인율을 스스로 계산한다.

```
package enumeration.ref3;

public class DiscountService {
```

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

- 할인율 계산은 이제 `Grade`가 스스로 처리한다. 따라서 `DiscountService.discount()` 메서드는 단순히 `Grade.discount()`를 호출하기만 하면 된다.

```
package enumeration.ref3;

public class EnumRefMain3_1 {
    public static void main(String[] args) {
        int price = 10000;
        DiscountService discountService = new DiscountService();
        int basic = discountService.discount(Grade.BASIC, price);
        int gold = discountService.discount(Grade.GOLD, price);
        int diamond = discountService.discount(Grade.DIAMOND, price);

        System.out.println("BASIC 등급의 할인 금액: " + basic);
        System.out.println("GOLD 등급의 할인 금액: " + gold);
        System.out.println("DIAMOND 등급의 할인 금액: " + diamond);
    }
}
```

실행 결과

```
BASIC 등급의 할인 금액: 1000
GOLD 등급의 할인 금액: 2000
DIAMOND 등급의 할인 금액: 3000
```

실행 결과는 기존과 같다.

DiscountService 제거

`Grade`가 스스로 할인율을 계산하면서 `DiscountService` 클래스가 더는 필요하지 않게 되었다.

```
package enumeration.ref3;

public class EnumRefMain3_2 {
    public static void main(String[] args) {
        int price = 10000;
        System.out.println("BASIC 등급의 할인 금액: " +
Grade.BASIC.discount(price));
```

```
        System.out.println("GOLD 등급의 할인 금액: " + Grade.GOLD.discount(price));
        System.out.println("DIAMOND 등급의 할인 금액: " +
Grade.DIAMOND.discount(price));
    }
}
```

실행 결과

```
BASIC 등급의 할인 금액: 1000
GOLD 등급의 할인 금액: 2000
DIAMOND 등급의 할인 금액: 3000
```

- 각각의 등급별로 자신의 `discount()`를 직접 호출하면 할인율을 구할 수 있다.
- `DiscountService`를 제거해도 되지만, 앞의 예제에서 사용되므로 복습을 위해 남겨두자.

중복 제거

출력 부분의 중복을 제거하자.

```
package enumeration.ref3;

public class EnumRefMain3_3 {
    public static void main(String[] args) {
        int price = 10000;
        printDiscount(Grade.BASIC, price);
        printDiscount(Grade.GOLD, price);
        printDiscount(Grade.DIAMOND, price);
    }

    private static void printDiscount(Grade grade, int price) {
        System.out.println(grade.name() + " 등급의 할인 금액: " +
grade.discount(price));
    }
}
```

실행 결과

```
BASIC 등급의 할인 금액: 1000
GOLD 등급의 할인 금액: 2000
DIAMOND 등급의 할인 금액: 3000
```

`grade.name()`을 통해서 ENUM의 상수 이름을 사용할 수 있다.

ENUM 목록

이후에 새로운 등급이 추가되더라도 `main()` 코드의 변경 없이 모든 등급의 할인을 출력해보자.

```
package enumeration.ref3;

public class EnumRefMain3_4 {
    public static void main(String[] args) {
        int price = 10000;
        Grade[] grades = Grade.values();
        for (Grade grade : grades) {
            printDiscount(grade, price);
        }
    }

    private static void printDiscount(Grade grade, int price) {
        System.out.println(grade.name() + " 등급의 할인 금액: " +
grade.discount(price));
    }
}
```

실행 결과

```
BASIC 등급의 할인 금액: 1000
GOLD 등급의 할인 금액: 2000
DIAMOND 등급의 할인 금액: 3000
```

`Grade.values()`를 사용하면 `Grade` 열거형의 모든 상수를 배열로 구할 수 있다.

문제와 풀이1

문제1 - 인증 등급 만들기

문제 설명

패키지의 위치는 `enumeration.test`를 사용하자.

회원의 인증 등급을 `AuthGrade`라는 이름의 열거형으로 만들어라.

인증 등급은 다음 3가지이고, 인증 등급에 따른 레벨과 설명을 가진다. 레벨과 설명을 `getXxx()` 메서드로 조회할 수

있어야 한다.

- GUEST(손님)
  - level=1
  - description=손님
- LOGIN(로그인 회원)
  - level=2
  - description=로그인 회원
- ADMIN(관리자)
  - level=3
  - description=관리자

정답

```
package enumeration.test.ex1;

public enum AuthGrade {
    GUEST(1, "손님"),
    LOGIN(2, "로그인 회원"),
    ADMIN(3, "관리자");

    private final int level;
    private final String description;

    AuthGrade(int level, String description) {
        this.level = level;
        this.description = description;
    }

    public int getLevel() {
        return level;
    }

    public String getDescription() {
        return description;
    }
}
```

문제2 - 인증 등급 열거형 조회하기

문제 설명

- `AuthGradeMain1`이라는 클래스를 만들고 다음 결과가 출력되도록 코드를 작성해라.
- 앞서 만든 `AuthGrade`을 활용하자

```
grade=GUEST, level=1, 설명=손님
grade=LOGIN, level=2, 설명=로그인 회원
grade=ADMIN, level=3, 설명=관리자
```

정답

```
package enumeration.test.ex1;

public class AuthGradeMain1 {

    public static void main(String[] args) {
        AuthGrade[] values = AuthGrade.values();
        for (AuthGrade value : values) {
            System.out.println("grade=" + value.name() + ", level=" +
value.getLevel() + ", 설명=" + value.getDescription());
        }
    }
}
```

문제3 - 인증 등급 열거형 활용하기

문제 설명

- `AuthGradeMain2` 클래스에 코드를 작성하자.
- 인증 등급을 입력 받아서 앞서 만든 `AuthGrade` 열거형으로 변환하자.
- 인증 등급에 따라 접근할 수 있는 화면이 다르다.
  - 예를 들어 GUEST 등급은 메인 화면만 접근할 수 있고, ADMIN 등급은 모든 화면에 접근할 수 있다.
  - 각각의 등급에 따라서 출력되는 메뉴 목록이 달라진다.
- 다음 출력 결과를 참고해서 코드를 완성하자.

GUEST 입력 예

```
당신의 등급을 입력하세요[GUEST, LOGIN, ADMIN]: GUEST
당신의 등급은 손님입니다.
==메뉴 목록==
- 메인 화면
```

LOGIN 입력 예

```
당신의 등급을 입력하세요[GUEST, LOGIN, ADMIN]: LOGIN
당신의 등급은 로그인 회원입니다.
==메뉴 목록==
- 메인 화면
- 이메일 관리 화면
```

ADMIN 입력 예

```
당신의 등급을 입력하세요[GUEST, LOGIN, ADMIN]: ADMIN
당신의 등급은 관리자입니다.
==메뉴 목록==
- 메인 화면
- 이메일 관리 화면
- 관리자 화면
```

잘못된 값이 입력되는 경우

```
당신의 등급을 입력하세요[GUEST, LOGIN, ADMIN]: x
Exception in thread "main" java.lang.IllegalArgumentException: No enum constant
enumeration.test.AuthGrade.X
    at java.base/java.lang.Enum.valueOf(Enum.java:293)
    at enumeration.test.AuthGrade.valueOf(AuthGrade.java:3)
    at enumeration.test.AuthGradeMain2.main(AuthGradeMain2.java:12)
```

참고: `Enum.valueOf()`를 사용할 때 잘못된 값이 입력되면 예와 같이 `IllegalArgumentException` 예외가 발생한다. 해당 예외를 잡아서 복구하는 방법은 예외 처리에서 학습한다.

정답

```
package enumeration.test.ex1;

import java.util.Scanner;

public class AuthGradeMain2 {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("당신의 등급을 입력하세요[GUEST, LOGIN, ADMIN]: ");
        String grade = scanner.nextLine();

        AuthGrade authGrade = AuthGrade.valueOf(grade.toUpperCase());
        System.out.println("당신의 등급은 " + authGrade.getDescription() + "입니
```

```
다.");

        System.out.println("==메뉴 목록==");
        if (authGrade.getLevel() > 0) {
            System.out.println("- 메인 화면");
        }
        if (authGrade.getLevel() > 1) {
            System.out.println("- 이메일 관리 화면");
        }
        if (authGrade.getLevel() > 2) {
            System.out.println("- 관리자 화면");
        }
    }
}
```

문제와 풀이2

문제 설명

- `enumeration.test.http` 패키지를 사용하자.
- `HttpStatus` 열거형을 만들어라.
- HTTP 상태 코드 정의
  - `OK`
    - code: 200
    - message: "OK"
  - `BAD_REQUEST`
    - code: 400
    - message: "Bad Request"
  - `NOT_FOUND`
    - code: 404
    - message: "Not Found"
  - `INTERNAL_SERVER_ERROR`
    - code: 500
    - message: "Internal Server Error"
- **참고**: HTTP 상태 코드는 200 ~ 299사이의 숫자를 성공으로 인정한다.

다음 `HttpStatus` 열거형을 완성해라. `HttpStatusMain` 코드와 실행 결과를 참고하자.

```java
package enumeration.test.http;

public enum HttpStatus {
    // 코드 작성
}
```

```java
package enumeration.test.http;

import java.util.Scanner;

public class HttpStatusMain {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("HTTP CODE: ");
        int httpCodeInput = scanner.nextInt();

        HttpStatus status = HttpStatus.findByCode(httpCodeInput);
        if (status == null) {
            System.out.println("정의되지 않은 코드");
        } else {
            System.out.println(status.getCode() + " " + status.getMessage());
            System.out.println("isSuccess = " + status.isSuccess());
        }
    }
}
```

실행 결과

```
HTTP CODE: 200
200 OK
isSuccess = true
```

```
HTTP CODE: 400
400 Bad Request
isSuccess = false
```

```
HTTP CODE: 404
404 Not Found
isSuccess = false
```

```
HTTP CODE: 500
500 Internal Server Error
isSuccess = false
```

정답

```java
package enumeration.test.http;

public enum HttpStatus {
    OK(200, "OK"),
    BAD_REQUEST(400, "Bad Request"),
    NOT_FOUND(404, "Not Found"),
    INTERNAL_SERVER_ERROR(500, "Internal Server Error");

    private final int code;
    private final String message;

    HttpStatus(int code, String message) {
        this.code = code;
        this.message = message;
    }

    public static HttpStatus findByCode(int code) {
        for (HttpStatus status : values()) {
            if (status.getCode() == code) {
                return status;
            }
        }
        return null;
    }

    public int getCode() {
        return code;
    }

    public String getMessage() {
        return message;
    }

    public boolean isSuccess() {
        return code >= 200 && code <= 299;
    }
```

```
}
```

정리