13. 열거형 (Enum)

enum 정의 및 사용

🟦 1. enum이란?

> `enum`(enumeration)은 관련된 **고정된 상수 값들의 집합**을 정의할 때 사용

```
public enum Direction {
    NORTH, EAST, SOUTH, WEST;
}
```

✅ `Direction` 타입은 오직 4개의 값만 가질 수 있어
✅ 타입 안정성이 생기고, 오타나 범위 오류 방지

🟧 2. 기본 사용 예제

```
public class EnumExample {
    public static void main(String[] args) {
        Direction dir = Direction.NORTH;
        System.out.println(dir);         // NORTH
        System.out.println(dir.name());  // "NORTH"
        System.out.println(dir.ordinal()); // 0
    }
}
```

🧩 3. `enum` 내부에 필드, 생성자, 메서드 정의 가능

```
public enum Planet {
    MERCURY(3.303e+23, 2.4397e6),
    EARTH(5.975e+24, 6.37814e6);

    private final double mass;   // 질량 (kg)
    private final double radius; // 반지름 (m)

    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }

    public double surfaceGravity() {
        final double G = 6.6730OE-11;
        return G * mass / (radius * radius);
    }
}
```

✅ 열거형도 **클래스처럼 필드와 생성자, 메서드 가짐**

🔁 4. 열거형의 핵심 메서드

메서드	설명
`name()`	상수 이름 (문자열) 반환
`ordinal()`	상수의 순서 (0부터 시작)
`valueOf(String)`	이름으로 enum 상수 반환
`values()`	모든 enum 상수 배열 반환

```
for (Direction d : Direction.values()) {
    System.out.println(d + ": " + d.ordinal());
}
```

🎭 5. 열거형에서 `switch` 사용

```
switch (dir) {
    case NORTH: System.out.println("위"); break;
    case SOUTH: System.out.println("아래"); break;
    default: System.out.println("기타");
}
```

✅ `enum`은 switch문과 아주 잘 어울림
✅ 타입 안정성을 유지한 채 직관적 제어 흐름 구성 가능

🛠️ 6. 열거형에 추상 메서드 정의도 가능

```
public enum Operation {
    PLUS {
        public int apply(int x, int y) { return x + y; }
    },
    MINUS {
        public int apply(int x, int y) { return x - y; }
    };

    public abstract int apply(int x, int y);
}
```

✅ 각 enum 상수가 메서드를 **오버라이딩** 가능
✅ 전략 패턴(Strategy Pattern)처럼 사용 가능

🔐 7. 열거형은 싱글톤과 비슷하다

- 각 enum 상수는 **JVM에서 단 하나의 인스턴스**
- 싱글톤 구현 대체용으로도 사용됨
- Java의 `Enum`은 `Serializable`, `Cloneable`, `final` 등의 특성을 자동으로 가짐

```
public enum Singleton {
    INSTANCE;

    public void doSomething() {
        System.out.println("Doing...");
    }
}
```

❗ 8. enum은 내부적으로 클래스로 처리됨

```
// enum Color { RED, GREEN, BLUE }
// 내부적으로 다음과 유사하게 변환됨:

final class Color extends Enum<Color> {
    public static final Color RED = new Color("RED", 0);
    public static final Color GREEN = new Color("GREEN", 1);
    ...
}
```

🔚 요약

항목	설명
키워드	`enum`
타입 안정성	✅ 매우 강함
필드/메서드	✅ 정의 가능
추상 메서드	✅ 가능 (상수마다 오버라이딩)
`switch`문	✅ 지원
싱글톤 용도	✅ 가능
클래스처럼 작동	✅ 컴파일 시 `Enum` 상속한 클래스 생성

메서드, 생성자 포함 enum

🟦 1. enum에 생성자, 필드, 메서드 추가하기

`enum`은 내부적으로 클래스로 동작하므로 **생성자, 필드, 메서드, 인터페이스 구현까지 모두 가능**하다.

```
public enum Status {
    READY(1, "준비됨"),
    RUNNING(2, "실행 중"),
    DONE(3, "완료됨");

    private final int code;
    private final String description;

    // 생성자 (private이 기본)
    Status(int code, String description) {
        this.code = code;
        this.description = description;
    }

    // Getter 메서드
    public int getCode() {
        return code;
    }

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

✅ 각 상수는 생성자 인자를 넘겨서 **필드를 초기화**함
✅ enum 생성자는 항상 **private** 또는 생략됨 (public 불가)

🧪 사용 예

```
public class Main {
    public static void main(String[] args) {
        Status s = Status.RUNNING;

        System.out.println(s);                      // RUNNING
        System.out.println(s.getCode());            // 2
        System.out.println(s.getDescription());     // 실행 중
    }
}
```

🛠️ 2. toString() 오버라이드

```
@Override
public String toString() {
    return code + " - " + description;
}
```

→ 출력 결과가 `RUNNING` 대신 `"2 - 실행 중"` 처럼 커스터마이징 가능

🧩 3. 값으로 enum 찾기 (정적 메서드 활용)

```
public static Status fromCode(int code) {
    for (Status s : values()) {
        if (s.code == code) return s;
    }
    throw new IllegalArgumentException("Invalid code: " + code);
}
```

→ `Status.fromCode(2)` → `Status.RUNNING` 반환

✅ `Map`으로 캐싱하면 성능 개선 가능 (정적 블록 사용)

⚙️ 4. enum + 메서드 오버라이딩 (열거 상수별로 다른 동작)

```
public enum Operation {
    PLUS {
        public int apply(int x, int y) { return x + y; }
    },
    MINUS {
        public int apply(int x, int y) { return x - y; }
    };

    public abstract int apply(int x, int y);
}
```

각 상수가 **메서드를 개별로 구현**함 (전략 패턴과 유사)

💡 실전 패턴 예시: 상태머신 정의

```
public enum OrderState {
    CREATED {
        public boolean canCancel() { return true; }
    },
    SHIPPED {
        public boolean canCancel() { return false; }
    };

    public abstract boolean canCancel();
}
```

→ 상수마다 동작이 다르고 `switch` 없이도 확장성 있게 설계 가능

📌 5. 생성자 주의사항

- 생성자는 enum 선언의 **맨 마지막**에 위치해야 함
- `enum` 상수는 **반드시 생성자보다 먼저 선언**
- **public 생성자 불가**, 컴파일 에러 발생함

✅ 정리 요약

기능	지원 여부
필드 추가	✅ 가능
생성자 정의	✅ 가능 (private or 생략)
getter/setter 메서드	✅ 가능
`toString`, `equals` 등 오버라이드	✅ 가능
상수별 다른 구현	✅ 추상 메서드 사용 가능
값 → enum 변환	✅ 정적 메서드 작성

🧷 예제 구조 요약

```
public enum Type {
    A(1, "alpha"), B(2, "beta");

    private final int code;
    private final String label;

    Type(int code, String label) {
        this.code = code;
```

```
        this.label = label;
    }

    public String getLabel() {
        return label;
    }

    public static Type fromCode(int code) {
        for (Type t : values()) {
            if (t.code == code) return t;
        }
        throw new IllegalArgumentException("invalid");
    }
}
```

`values()`, `ordinal()`, `valueOf()`

🟦 1. `values()` – 모든 enum 상수를 배열로 반환

```
public enum Direction {
    NORTH, EAST, SOUTH, WEST;
}

Direction[] directions = Direction.values();
```

🔧 동작 설명

- `values()`는 컴파일러가 자동 생성하는 **정적(static)** 메서드이다.
- 각 enum 타입마다 자동으로 **모든 열거 상수를 순서대로 저장한 배열**을 만들어 리턴한다.

```
// 실제로 이런 식으로 컴파일됨
public static final Direction[] values() {
    return new Direction[] { NORTH, EAST, SOUTH, WEST };
}
```

✅ 활용 예

```
for (Direction d : Direction.values()) {
    System.out.println(d);
}
```

→ `NORTH`, `EAST`, `SOUTH`, `WEST` 출력

🟧 2. `ordinal()` - enum 상수의 순서 (인덱스)를 반환

```
Direction.NORTH.ordinal();  // 0
Direction.EAST.ordinal();   // 1
```

🔧 동작 설명

- `Enum` 클래스의 **인스턴스 메서드**
- 각 enum 상수는 **선언 순서대로 인덱스(0부터 시작)** 가 자동 부여됨

```
public final int ordinal() {
    return this.ordinal;
}
```

> `ordinal`은 컴파일 시 enum 내부에서 정해지는 `final` 필드임

⚠️ 주의

- `ordinal()` 값은 **순서 기반이므로 하드코딩된 인덱스로 로직을 짜면 위험**함
- 나중에 enum 순서 바뀌면 버그 발생할 수 있음

🟥 3. `valueOf(String name)` - 문자열 → enum 변환

```
Direction d = Direction.valueOf("NORTH");
```

🔧 동작 설명

- `Enum` 클래스의 정적 메서드
- 매개변수 문자열과 `name()` 값이 **정확히 일치**해야 함

```
public static <T extends Enum<T>> T valueOf(Class<T> enumType, String name)
```

```
Direction.valueOf("NORTH") // OK
Direction.valueOf("north") // ❌ 예외 발생 (대소문자 구분)
```

❗ 예외 처리

```
try {
    Direction dir = Direction.valueOf("WRONG");
} catch (IllegalArgumentException e) {
    System.out.println("존재하지 않는 상수");
}
```

📋 4. 보너스: `name()`과 `toString()`의 차이

메서드	설명
`name()`	enum 상수 이름 그대로 반환 (불변, final)
`toString()`	`name()`과 동일 (기본적으로), 하지만 오버라이드 가능

```
Direction.NORTH.name();     // "NORTH"
Direction.NORTH.toString(); // "NORTH"

@Override
public String toString() {
    return "방향: " + name();
}
```

✅ 요약 비교

메서드	반환값	용도
`values()`	enum[]	모든 enum 상수 배열 반환
`ordinal()`	int	enum 상수의 선언 순서 반환
`valueOf()`	enum	문자열로 enum 상수 반환 (정확히 일치해야 함)

💡 실전 팁

- `values()`는 `for` 반복에 자주 쓰임
- `valueOf()`는 주로 **문자열 → enum 매핑**에 사용 (ex. DB, JSON 파싱)
- `ordinal()`은 되도록 **비즈니스 로직에서 직접 사용 ❌**