Ch6. 열거 타입과 에너테이션

태그 3주차

Item34 int 상수 대신 열거 타입을 사용하라

int 상수 방식

```
public static final int APPLE_FUJI = 0;
public static final int APPLE_PIPPIN = 1;
public static final int APPLE_SMITH = 2;

public static final int ORANGE_NAVEL = 0;
public static final int ORANGE_TEMPLE = 1;
public static final int ORANGE_BLOOD = 2;
```

단점: 서로 다른 종류끼리 동등 연산자(==)로 비교해도 컴파일 에러가 나지 않음.

```
int i = (APPLE_FUJI - ORANGE_TEMPLE) / APPLE_PIPPIN;
```

열거 타입

- 자바의 **열거 타입은 완전한 형태의 클래스**이며, 상수 하나당 자신의 인스턴스를 하나씩 만들어 `public static final` 필드로 공개한다.
- 클라이언트가 인트선스를 직접 생성하거나 확장할 수 없으니, 인스턴스들은 딱 하나씩만 존재한다. 열거타입은 싱글턴을 일반화한 형태인 것이다.
- 열거타입은 컴파일타임 타입 안전성을 제공한다. Apple 열거 타입을 매개변수로 받는 메서드를 선언하고 다른 타입을 넘기려하면 컴파일 오류가 난다.
- 열거 타입에는 각자의 이름공간이 있어 이름이 같은 상수도 공존가능하다.
  열거타입에 새로운 상수를 추가하거나 순서를 바꿔도 다시 컴파일 하지 않아도 된다.

```
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
    }
}
```

- **열거 타입 상수 각각을 특정 데이터와 연결지으려면 생성자에서 데이터를 받아 인스턴스 필드에 저장하면 된다.**

```
// 어떤 객체의 지구에서의 무게를 입력받아 여덟 행성에서의 무게를 출력한
다. (212쪽)
public class WeightTable {
   public static void main(String[] args) {
```

```
        double earthWeight = Double.parseDouble(args[0]);
        double mass = earthWeight / Planet.EARTH.surfaceGravi
ty();
        for (Planet p : Planet.values())
           System.out.printf("%s에서의 무게는 %f이다.%n",
                             p, p.surfaceWeight(mass));
    }
}
```

- `toString` 메서드는 상수 이름을 문자열을 반환한다.
- 열거타입에서 상수를 제거해도 참조하지 않는 클라이언트에는 아무 영향이 없다.

하나의 메서드가 상수별로 다르게 동작할때

Switch문을 사용하여 작성하는 것은 위험하다!

```
enum PayrolLDay {
    MONDAY, TUESDAY, WEDSENDAY, THURSDAY, FRIDAY, SATURDAY, S

    private static final int MINS_PER_SHIFT = 8 * 60;

    int pay(int minutesWorked, int payRate) {
        int basePay = minuitesWorked * payRate;

        int overtimePay;
        switch(this) {
            case SATURDAY: case SUNDAY:
                overtimePay = basePay /2;
                break;
            default:
                overtimePay = minuutesWorked <= MINS_PER_SHIF
        }
        return basePay + overtimePay;
    }
}
```

확장 및 오류 가능성이 높아진다 → **전략 열거 타입 패턴**을 사용하자

```
enum PayrollDay {
    MONDAY(WEEKDAY), TUESDAY(WEEKDAY), WEDNESDAY(WEEKDAY),
    THURSDAY(WEEKDAY), FRIDAY(WEEKDAY),
    SATURDAY(WEEKEND), SUNDAY(WEEKEND);
    // (역자 노트) 원서 1~3쇄와 한국어판 1쇄에는 위의 3줄이 아래처럼 인쇄
    //
    // MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
    // SATURDAY(PayType.WEEKEND), SUNDAY(PayType.WEEKEND);
    //
    // 저자가 코드를 간결하게 하기 위해 매개변수 없는 기본 생성자를 추가했
    // 열거 타입에 새로운 값을 추가할 때마다 적절한 전략 열거 타입을 선택
    // 이 패턴의 의도를 잘못 전달할 수 있어서 원서 4쇄부터 코드를 수정할

    private final PayType payType;

    PayrollDay(PayType payType) { this.payType = payType; }
    // PayrollDay() { this(PayType.WEEKDAY); } // (역자 노트) 원

    int pay(int minutesWorked, int payRate) {
        return payType.pay(minutesWorked, payRate);
    }

    // 전략 열거 타입
    enum PayType {
        WEEKDAY {
            int overtimePay(int minsWorked, int payRate) {
                return minsWorked <= MINS_PER_SHIFT ? 0 :
                        (minsWorked - MINS_PER_SHIFT) * payRa
            }
        },
        WEEKEND {
            int overtimePay(int minsWorked, int payRate) {
                return minsWorked * payRate / 2;
            }
        };
```

```
        abstract int overtimePay(int mins, int payRate);
        private static final int MINS_PER_SHIFT = 8 * 60;

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

    public static void main(String[] args) {
        for (PayrollDay day : values())
            System.out.printf("%-10s%d%n", day, day.pay(8 * 6
    }
}
```

→ 상수별로 메서드를 구현하는 것으로 대체

⭐핵심 정리

> 💡 열거 타입은 정수 상수보다 읽기 쉽고 안전하고 강력하다. 대다수 열거 타입이 명시적 생성자나 메서드 없이 쓰이지만, 각 상수를 특정 데이터와 연결짓거나 상수마다 다르게 동작하게 할 때는 필요하다. 드물게는 하나의 메서드가 상수별로 다르게 동작해야 할 때도 있다. 이런 열거 타입에서는 switch문 대신 상수별 메서드 구현을 사용하자

Item35 ordinal 메서드 대신 인스턴스 필드를 사용하라

열거 타입은 해당 상수가 그 열거 타입에서 몇 번째 위치인지를 반환하는 ordinal이라는 메서드를 반환한다.

ordinal을 사용하니 실제 숫자와 다르다 (ordinal은 0부터 시작). 그래서 `ordinal+1`과 같이 가독성이 떨어지는 코드가 나온다.

열거 타입 상수에 연결된 값은 ordinal 메서드로 얻지 말고, 인스턴스 필드에 저장하자.

```
public enum Ensemble {
    SOLO(1), DUET(2), TRIO(3), QUARTET(4), QUINTET(5),
    SEXTET(6), SEPTET(7), OCTET(8), DOUBLE_QUARTET(8),
    NONET(9), DECTET(10), TRIPLE_QUARTET(12);

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

Item36 비트 필드 대신 EnumSet을 사용하라

열거 값들을 집합으로 사용

- 예시 → 굵게, 기울게, 밑줄 등 여러 요소가 적용되는 경우

비트 필드를 사용하는 안좋은 예시

```
public class Text{
    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) {
        System.out.printf("Applying styles %s to text%n", sty
   }
}
```

EnumSet을 사용한 좋은 예시

`java.util` 패키지의 **EnumSet 클래스**는 **열거 타입 상수의 값으로 구성된 집합을 효과적으로 표현**해준다.

```
public class Text {
    public enum Style {BOLD, ITALIC, UNDERLINE, STRIKETHROUGH
```

```
    public void applyStyles(Set<Style> styles) {
        System.out.printf("Applying styles %s to text%n",
                Objects.requireNonNull(styles));
    }
}

public static void main(String[] args) {
     Text text = new Text();
     text.applyStyles(EnumSet.of(Style.BOLD, Style.ITALIC));
}
```

Item37 ordinal 인덱싱 대신 EnumMap을 사용하라

배열이나 리스트에서 원소를 꺼낼 때와 같이 인덱싱이 필요할때는 EnumMap을 사용하자

```
lass Plant {
    enum LifeCycle { ANNUAL, PERENNIAL, BIENNIAL }  // 생애
주기

    final String name;
    final LifeCycle lifeCycle;

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

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

생애주기별로 총 3개의 집합을 만들고, 각 식물을 해당 집합에 넣자.

```
Map<Plant.LifeCycle, Set<Plant>> plantsByLifeCycle =
                new EnumMap<>(Plant.LifeCycle.class);  // Enu
```

```
for (Plant.LifeCycle lc : Plant.LifeCycle.values())
      plantsByLifeCycle.put(lc, new HashSet<>());

for (Plant p : garden)
      plantsByLifeCycle.get(p.lifeCycle).add(p);

System.out.println(plantsByLifeCycle);
```

⭐핵심 정리

💡 EnumMap을 사용하라. 다차원 관계는 `EnumMap<..., EnumMap<...>>` 으로 표현하라.

Item38 확장할 수 있는 열거 타입이 필요하면 인터페이스를 사용하라

열거 타입은 확장할 수 없다. 대부분 상황에서 열거 타입을 확장하는 것은 좋지 않은 생각이다. 단 연산 코드는 예외이다.

- **인터페이스를 이용해 확장 가능 열거 타입을 흉내낸 예시**

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

```
public enum BasicOperation implements Operationn {
    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 * y
    },
    DIVIDE("/") {
        public double apply(double x, double y) { return x / y
    };

    private final String symbol;

    BasicOperation(String symbol) {
        this.symbol = symbol;
    }

    @Override
    public String toString() {
        return symbol;
    }
}
```

Operation을 구현한 또 다른 열거타입을 정의해 대체할 수 있다.

```
public enum ExtendedOperation implements Operation {
    EXP("^") {
        public double apply(double x, double y) {
            return Math.pow(x, y);
        }
    },
    REMAINDER("%") {
        public double apply(double x, double y) {
            return x % y;
        }
    };

    private final String symbol;
    ...
}
```

이처럼 inteface를 구현하는 여러 enum들에 접근할 수 있다.

Item39 명명 패턴보다 애너테이션을 사용하라

전통적으로, 도구나 프레임워크가 특별이 다뤄야 할 프로그램 요소에는 구분되는 **명명 패턴**을 적용해왔다. 예를 들어, JUnit은 버전 3까지 테스트 메서드 이름을 test로 시작하게끔 하였다. 하지만 이는 전부 어노테이션으로 대체할 수 있다.

```
import java.lang.annotation.*;

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

`@Test` 에너테이션 타입 선언 자체에 두 가지의 다른 애너테이션이 달려 있는데, 이를 **메타 애너테이션(meta-annotation)**이라 한다.

1) `@Retention(RetentionPolicy.RUNTIME)` 메타 애너테이션은 `@Test`가 런타임에도 유지되어야 한다는 의미이며, 만약 이를 생략하면 테스트 도구는 `@Test`를 인식할 수 없게 된다.

2) `@Target(ElementType.METHOD)` 메타 애너테이션은 `@Test`가 반드시 메서드 선언에만 사용되어야 한다고 알려주는 것이다. (*클래스 선언, 필드 선언 등 다른 프로그램 요소에는 달 수 없음*)

이처럼 `@Test`와 같은 애너테이션을, "아무 매개변수 없이 단순한 대상에 마킹한다"라는 뜻에서 **마커 애너테이션**이라 한다. 즉 프로그래머가 Test이름에 오타를 내거나 메서드 선언 외의 프로그램 요소에 달면 컴파일 오류를 내준다.

⭐핵심정리

> 💡 어노테이션으로 할 수 있는 일을 명명 패턴으로 처리할 이유는 없다!
> 도구 제작자를 제외하고는 일반 프로그래머가 어노테이션 타입을 직접 정의할 일은 없다. 하지만 자바 프로그래머라면 예외 없이 자바가 제공하는 애너테이션 타입들은 사용해야 한다.

Item40 @Override 어노테이션을 일관되게 사용하라

오버라이딩이 아닌 매개변수를 실수하여 오버로딩이 되는 등 실수를 할 가능성이 있다.

상위 클래스의 메서드를 재정의하려는 모든 메서드에 @Override 애너테이션을 달자.

상위 클래스의 추상 메서드를 구체 클래스에서 재정의할 때는 `@Override` 안달아도 되지만 일관성 위해 붙이는 것도 괜찮다.

인터페이스의 메서드를 구현할 때도 붙이는 습관을 들이면 시그니처가 올바른지 확인할 수 있다.

만약 구현하려는 인터페이스에 디폴트 메서드가 없음을 확신한다면 안붙여서 깔끔하게 유지해도 좋다.

⭐핵심 정리

> 💡 추상 클래스나 인터페이스에서는 상위 클래스나 상위 인터페이스의 메서드를 재정의하는 모든 메서드에 @Override를 다는 것이 좋다. Set 인터페이스는 Collection 인터페이스를 확장했지만 새로 추가한 메서드는 없다. 따라서 모든 선언에 `@Override`를 붙여서 **실수로 추가한 메서드가 없음을 보장하자**

Item41 정의하려는 것이 타입이라면 마커 인터페이스를 사용하라

마커 인터페이스란, 아무 메서드도 담고 있지 않고 단지 자신을 구현하는 클래스가 **특정 속성**을 가짐을 표시해주는 인터페이스를 의미한다.

대표적인 예로, `Serializable` 인터페이스가 있다. 해당 인터페이스는 단순히 자신을 구현한 클래스의 인스턴스는 `ObjectOutputstream` 을 통해 쓸 수 있다고(직렬화) 알려주는 역할을 한다.

마커 어노테이션이란 역할은 같지만 이를 어노테이션으로 표현한 것

- 마커 어노테이션과 마커 인터페이스를 사용하는 상황

1. **클래스와 인터페이스 외**의 프로그램 요소(모듈/패키지/필드/지역변수 등)에 마킹해야 할때는 **애너테이션**을 사용한다. **(사용할 수 밖에 없다. 클래스와 인터페이스만이 인터페**

이스를 구현하거나 확장할 수 있기 때문)

2. **마킹이 된 객체를 매개변수로 받는 메서드**를 작성할 일이 있다면 마커 인터페이스를 사용해야 한다. 마커 인터페이스를 해당 메서드의 매개변수 타입으로 사용하여 컴파일 타임에 오류를 잡아낼 수 있기 때문이다.

예를 들어 `@Test`는 `Test` 인터페이스를 매개변수로 받는 메서드를 작성할 일이 없으므로 마커 애노테이션을 사용하는게 맞다.

⭐핵심정리

> 💡 새로 추가하는 메서드 없이 단지 타입 정의가 목적이라면 마커 인터페이스를 정의하자. 클래스나 인터페이스 외의 프로그램 요소에 마킹해야 하거나, 애너테이션을 적극 활용하는 프레임워크의 일부로 마커를 편입시키고자 한다면 마커 애너테이션을 사용하자.