Item Int 상수 대신 열거 타입을 사용하라

Item Int 상수 대신 열거 타입을 사용하라

열거 타입(Enum) 지원 전에는 정수 상수를 한 묶음 선언해서 사용하곤 하였다

```
public class Test {
    public static final int APPLE_FUJI = 0;
    public static final int APPLE_PIPPIN = 1;
    public static final int APPLE_GRANNY_SMITH = 2;

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

Item Int 상수 대신 열거 타입을 사용하라

열거 타입(Enum) 지원 전에는 정수 상수를 한 묶음 선언해서 사용하곤 하였다

```
public class Test {
    public static final int APPLE_FUJI = 0;
    public static final int APPLE_PIPPIN = 1;
    public static final int APPLE_GRANNY_SMITH = 2;

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

이런 기법에는 단점이 굉장히 많은데…

Item Int 상수 대신 열거 타입을 사용하라

열거 타입(Enum) 지원 전에는 정수 상수를 한 묶음 선언해서 사용하곤 하였다

```
public class Test {
    public static final int APPLE_FUJI = 0;
    public static final int APPLE_PIPPIN = 1;
    public static final int APPLE_GRANNY_SMITH = 2;

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

이런 기법에는 단점이 굉장히 많은데…

1. 타입 안정성을 보장할 수 없다

Item Int 상수 대신 열거 타입을 사용하라

열거 타입(Enum) 지원 전에는 정수 상수를 한 묶음 선언해서 사용하곤 하였다

```
public class Test {
    public static final int APPLE_FUJI = 0;
    public static final int APPLE_PIPPIN = 1;
    public static final int APPLE_GRANNY_SMITH = 2;

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

이런 기법에는 단점이 굉장히 많은데…

1. 타입 안정성을 보장할 수 없다
2. APPLE_FUJI == ORANGE_NAVEL true?

Item Int 상수 대신 열거 타입을 사용하라

열거 타입(Enum) 지원 전에는 정수 상수를 한 묶음 선언해서 사용하곤 하였다

```
public class Test {
    public static final int APPLE_FUJI = 0;
    public static final int APPLE_PIPPIN = 1;
    public static final int APPLE_GRANNY_SMITH = 2;

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

이런 기법에는 단점이 굉장히 많은데…

1. 타입 안정성을 보장할 수 없다
2. APPLE_FUJI == ORANGE_NAVEL true?
3. Int I = (APLE_FUJI - ORANGE_TEMPLE) / APPLE_PIPPIN 향긋한 오렌지 향의 사과 소스?

Item Int 상수 대신 열거 타입을 사용하라

열거 타입(Enum) 지원 전에는 정수 상수를 한 묶음 선언해서 사용하곤 하였다

```
public class Test {
    public static final int APPLE_FUJI = 0;
    public static final int APPLE_PIPPIN = 1;
    public static final int APPLE_GRANNY_SMITH = 2;

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

이런 기법에는 단점이 굉장히 많은데…

1. 타입 안정성을 보장할 수 없다
2. APPLE_FUJI == ORANGE_NAVEL true?
3. Int I = (APLE_FUJI - ORANGE_TEMPLE) / APPLE_PIPPIN 향긋한 오렌지 향의 사과 소스?
4. 네임스페이스를 지원하지 않는다 MERCURY(원소), MERCURY(수성) 동시에 등록할수가 없어서 ELEMENT_MERCURY, PLANET_MERCUR

Item Int 상수 대신 열거 타입을 사용하라

열거 타입(Enum) 지원 전에는 정수 상수를 한 묶음 선언해서 사용하곤 하였다

```
public class Test {
    public static final int APPLE_FUJI = 0;
    public static final int APPLE_PIPPIN = 1;
    public static final int APPLE_GRANNY_SMITH = 2;

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

이런 기법에는 단점이 굉장히 많은데…

1. 타입 안정성을 보장할 수 없다
2. APPLE_FUJI == ORANGE_NAVEL true?
3. Int I = (APLE_FUJI - ORANGE_TEMPLE) / APPLE_PIPPIN 향긋한 오렌지 향의 사과 소스?
4. 네임스페이스를 지원하지 않는다 MERCURY(원소), MERCURY(수성) 동시에 등록할수가 없어서 ELEMENT_MERCURY, PLANET_MERCUR
5. 상수값이 바뀌면 클라이언트에서 재 컴파일이 필요하다

Item Int 상수 대신 열거 타입을 사용하라

이 모든걸 해결하기 위해 나타난것이 바로

1. 타입 안정성을 보장할 수 없다
2. APPLE_FUJI == ORANGE_NAVEL
3. Int I = (APLE_FUJI - ORANGE_TEMPLE) / APPLE_PIPPIN
4. 네임스페이스를 지원하지 않는다
5. 상수값이 바뀌면 클라이언트에서 재 컴파일이 필요하다

Item Int 상수 대신 열거 타입을 사용하라

Enum type

1. ~~타입 안정성을 보장할 수 없다~~
2. ~~APPLE_FUJI == ORANGE_NAVEL~~
3. ~~Int I = (APLE_FUJI - ORANGE_TEMPLE) / APPLE_PIPPIN~~
4. ~~네임스페이스를 지원하지 않는다~~
5. ~~상수값이 바뀌면 클라이언트에서 재 컴파일이 필요하다~~

```
public class Test {
   public enum Apple { FUJI, PIPPIN, GRANNY_SMITH }
   public enum Orange { NAVEL, TEMPLE, BLOOD }

}
```

Enum type

```
public class Test {
   public enum Apple { FUJI, PIPPIN, GRANNY_SMITH }
   public enum Orange { NAVEL, TEMPLE, BLOOD }

}
```

자바의 Enum은 C, C++, C#의 열거타입과 비슷한듯 보이지만
완전한 형태의 클래스라서 **다른 언어의 열거타입보다 훨씬 강력**하다

Item Int 상수 대신 열거 타입을 사용하라

자바의 열거타입 특징

```
public class Test {
   public enum Apple { FUJI, PIPPIN, GRANNY_SMITH }
   public enum Orange { NAVEL, TEMPLE, BLOOD }

}
```

자바의 Enum은 C, C++, C#의 열거타입과 비슷한듯 보이지만
완전한 형태의 클래스라서 **다른 언어의 열거타입보다 훨씬 강력**하다

Item Int 상수 대신 열거 타입을 사용하라

자바의 열거타입 특징

완전한 형태의 클래스로써, 단순 정수값 기반인 타 언어 열거타입보다 강력함

```
public class Test {
   public enum Apple { FUJI, PIPPIN, GRANNY_SMITH }
   public enum Orange { NAVEL, TEMPLE, BLOOD }

}
```

자바의 Enum은 C, C++, C#의 열거타입과 비슷한듯 보이지만
완전한 형태의 클래스라서 **다른 언어의 열거타입보다 훨씬 강력**하다

Item Int 상수 대신 열거 타입을 사용하라

자바의 열거타입 특징

완전한 형태의 클래스로써, 단순 정수값 기반인 타 언어 열거타입보다 강력함
상수 하나당 자신의 인스턴스를 만들어 public static final 필드로 공개한다

```
public class Test {
   public enum Apple { FUJI, PIPPIN, GRANNY_SMITH }
   public enum Orange { NAVEL, TEMPLE, BLOOD }

}
```

자바의 Enum은 C, C++, C#의 열거타입과 비슷한듯 보이지만
완전한 형태의 클래스라서 **다른 언어의 열거타입보다 훨씬 강력**하다

Item Int 상수 대신 열거 타입을 사용하라

자바의 열거타입 특징

완전한 형태의 클래스로써, 단순 정수값 기반인 타 언어 열거타입보다 강력함
상수 하나당 자신의 인스턴스를 만들어 public static final 필드로 공개한다
생성자를 제공하지 않으므로 final(불변) 이다

```
public class Test {
   public enum Apple { FUJI, PIPPIN, GRANNY_SMITH }
   public enum Orange { NAVEL, TEMPLE, BLOOD }

}
```

자바의 Enum은 C, C++, C#의 열거타입과 비슷한듯 보이지만
완전한 형태의 클래스라서 **다른 언어의 열거타입보다 훨씬 강력**하다

Item Int 상수 대신 열거 타입을 사용하라

자바의 열거타입 특징

완전한 형태의 클래스로써, 단순 정수값 기반인 타 언어 열거타입보다 강력함
상수 하나당 자신의 인스턴스를 만들어 public static final 필드로 공개한다
생성자를 제공하지 않으므로 final(불변) 이다
클라이언트가 인스턴스를 직접 생성하거나 확장할 수 없으므로 선언한 인스턴스 자체로서 딱 하나만 존재한다 (= 싱글턴)

```
public class Test {
   public enum Apple { FUJI, PIPPIN, GRANNY_SMITH }
   public enum Orange { NAVEL, TEMPLE, BLOOD }

}
```

자바의 Enum은 C, C++, C#의 열거타입과 비슷한듯 보이지만
완전한 형태의 클래스라서 **다른 언어의 열거타입보다 훨씬 강력**하다

Item Int 상수 대신 열거 타입을 사용하라

자바의 열거타입 특징

완전한 형태의 클래스로써, 단순 정수값 기반인 타 언어 열거타입보다 강력함
상수 하나당 자신의 인스턴스를 만들어 public static final 필드로 공개한다
생성자를 제공하지 않으므로 final(불변) 이다
클라이언트가 인스턴스를 직접 생성하거나 확장할 수 없으므로 선언한 인스턴스 자체로서 딱 하나만 존재한다 (= 싱글턴)
컴파일 타임 안정성을 제공하며 -> 파라미터로 전달받은 Enum은 선언된 Enum 타입 중 무조건 한개가 매핑된다

```
public class Test {
   public enum Apple { FUJI, PIPPIN, GRANNY_SMITH }
   public enum Orange { NAVEL, TEMPLE, BLOOD }

}
```

자바의 Enum은 C, C++, C#의 열거타입과 비슷한듯 보이지만
완전한 형태의 클래스라서 **다른 언어의 열거타입보다 훨씬 강력**하다

Item Int 상수 대신 열거 타입을 사용하라

자바의 열거타입 특징

완전한 형태의 클래스로써, 단순 정수값 기반인 타 언어 열거타입보다 강력함
상수 하나당 자신의 인스턴스를 만들어 public static final 필드로 공개한다
생성자를 제공하지 않으므로 final(불변) 이다
클라이언트가 인스턴스를 직접 생성하거나 확장할 수 없으므로 선언한 인스턴스 자체로서 딱 하나만 존재한다 (= 싱글턴)
컴파일 타임 안정성을 제공하며 -> 파라미터로 전달받은 Enum은 선언된 Enum 타입 중 무조건 한개가 매핑된다
네임스페이스가 제공되어 이름이 같은 상수도 공존가능하고

```
public class Test {
   public enum Apple { FUJI, PIPPIN, GRANNY_SMITH }
   public enum Orange { NAVEL, TEMPLE, BLOOD }

}
```

자바의 Enum은 C, C++, C#의 열거타입과 비슷한듯 보이지만
완전한 형태의 클래스라서 **다른 언어의 열거타입보다 훨씬 강력**하다

Item Int 상수 대신 열거 타입을 사용하라

자바의 열거타입 특징

완전한 형태의 클래스로써, 단순 정수값 기반인 타 언어 열거타입보다 강력함
상수 하나당 자신의 인스턴스를 만들어 public static final 필드로 공개한다
생성자를 제공하지 않으므로 final(불변) 이다
클라이언트가 인스턴스를 직접 생성하거나 확장할 수 없으므로 선언한 인스턴스 자체로서 딱 하나만 존재한다 (= 싱글턴)
컴파일 타임 안정성을 제공하며 -> 파라미터로 전달받은 Enum은 선언된 Enum 타입 중 무조건 한개가 매핑된다
네임스페이스가 제공되어 이름이 같은 상수도 공존가능하고
새로운 상수를 추가하거나 순서를 바꿔도 다시 컴파일 하지 않아도 되고..

```
public class Test {
   public enum Apple { FUJI, PIPPIN, GRANNY_SMITH }
   public enum Orange { NAVEL, TEMPLE, BLOOD }

}
```

자바의 Enum은 C, C++, C#의 열거타입과 비슷한듯 보이지만
완전한 형태의 클래스라서 **다른 언어의 열거타입보다 훨씬 강력**하다

Item Int 상수 대신 열거 타입을 사용하라

자바의 열거타입 특징

완전한 형태의 클래스로써, 단순 정수값 기반인 타 언어 열거타입보다 강력함
상수 하나당 자신의 인스턴스를 만들어 public static final 필드로 공개한다
생성자를 제공하지 않으므로 final(불변) 이다
클라이언트가 인스턴스를 직접 생성하거나 확장할 수 없으므로 선언한 인스턴스 자체로서 딱 하나만 존재한다 (= 싱글턴)
컴파일 타임 안정성을 제공하며 -> 파라미터로 전달받은 Enum은 선언된 Enum 타입 중 무조건 한개가 매핑된다
네임스페이스가 제공되어 이름이 같은 상수도 공존가능하고
새로운 상수를 추가하거나 순서를 바꿔도 다시 컴파일 하지 않아도 되고..
toString메서드를 아름다운 형태로 지원해주고…

```
public class Test {
   public enum Apple { FUJI, PIPPIN, GRANNY_SMITH }
   public enum Orange { NAVEL, TEMPLE, BLOOD }

}
```

자바의 Enum은 C, C++, C#의 열거타입과 비슷한듯 보이지만
완전한 형태의 클래스라서 **다른 언어의 열거타입보다 훨씬 강력**하다

Item Int 상수 대신 열거 타입을 사용하라

자바의 열거타입 특징

완전한 형태의 클래스로써, 단순 정수값 기반인 타 언어 열거타입보다 강력함
상수 하나당 자신의 인스턴스를 만들어 public static final 필드로 공개한다
생성자를 제공하지 않으므로 final(불변) 이다
클라이언트가 인스턴스를 직접 생성하거나 확장할 수 없으므로 선언한 인스턴스 자체로서 딱 하나만 존재한다 (= 싱글턴)
컴파일 타임 안정성을 제공하며 -> 파라미터로 전달받은 Enum은 선언된 Enum 타입 중 무조건 한개가 매핑된다
네임스페이스가 제공되어 이름이 같은 상수도 공존가능하고
새로운 상수를 추가하거나 순서를 바꿔도 다시 컴파일 하지 않아도 되고..
toString메서드를 아름다운 형태로 지원해주고…
메서드나 필드를 추가할수도있고…

```
public class Test {
   public enum Apple { FUJI, PIPPIN, GRANNY_SMITH }
   public enum Orange { NAVEL, TEMPLE, BLOOD }

}
```

자바의 Enum은 C, C++, C#의 열거타입과 비슷한듯 보이지만
완전한 형태의 클래스라서 **다른 언어의 열거타입보다 훨씬 강력**하다

Item Int 상수 대신 열거 타입을 사용하라

자바의 열거타입 특징

완전한 형태의 클래스로써, 단순 정수값 기반인 타 언어 열거타입보다 강력함
상수 하나당 자신의 인스턴스를 만들어 public static final 필드로 공개한다
생성자를 제공하지 않으므로 final(불변) 이다
클라이언트가 인스턴스를 직접 생성하거나 확장할 수 없으므로 선언한 인스턴스 자체로서 딱 하나만 존재한다 (= 싱글턴)
컴파일 타입 안정성을 제공하며 -> 파라미터로 전달받은 Enum은 선언된 Enum 타입 중 무조건 한개가 매핑된다
네임스페이스가 제공되어 이름이 같은 상수도 공존가능하고
새로운 상수를 추가하거나 순서를 바꿔도 다시 컴파일 하지 않아도 되고..
toString메서드를 아름다운 형태로 지원해주고…
메서드나 필드를 추가할수도있고…
인터페이스를 구현하게 할 수도 있다… (전에 배운 Comparable, Serializable처럼)

```
public class Test {
   public enum Apple { FUJI, PIPPIN, GRANNY_SMITH }
   public enum Orange { NAVEL, TEMPLE, BLOOD }

}
```

자바의 Enum은 C, C++, C#의 열거타입과 비슷한듯 보이지만
완전한 형태의 클래스라서 **다른 언어의 열거타입보다 훨씬 강력**하다

Item Int 상수 대신 열거 타입을 사용하라

자바의 열거타입 특징

완전한 형태의 클래스로써, 단순 정수값 기반인 타 언어 열거타입보다 강력함
상수 하나당 자신의 인스턴스를 만들어 public static final 필드로 공개한다
생성자를 제공하지 않으므로 final(불변) 이다
클라이언트가 인스턴스를 직접 생성하거나 확장할 수 없으므로 선언한 인스턴스 자체로서 딱 하나만 존재한다 (= 싱글턴)
컴파일 타입 안정성을 제공하며 -> 파라미터로 전달받은 Enum은 선언된 Enum 타입 중 무조건 한개가 매핑된다
네임스페이스가 제공되어 이름이 같은 상수도 공존가능하고
새로운 상수를 추가하거나 순서를 바꿔도 다시 컴파일 하지 않아도 되고..
toString메서드를 아름다운 형태로 지원해주고…
메서드나 필드를 추가할수도있고…
인터페이스를 구현하게 할 수도 있다… (전에 배운 Comparable, Serializable처럼)

그냥.. 너무 좋으니까 이걸써야만 한다 수준

```
public class Test {
   public enum Apple { FUJI, PIPPIN, GRANNY_SMITH }
   public enum Orange { NAVEL, TEMPLE, BLOOD }

}
```

자바의 Enum은 C, C++, C#의 열거타입과 비슷한듯 보이지만
완전한 형태의 클래스라서 **다른 언어의 열거타입보다 훨씬 강력**하다

Item Int 상수 대신 열거 타입을 사용하라

기존 상수를 사용하는 시나리오를 살펴보면…

```
public static void main(String[] args) {
    System.out.println("나는....FUJI야.. 내 값은.." + APPLE_FUJI);
    System.out.println("나는.. PIPPIN이야... " + APPLE_PIPPIN);
    System.out.println("나는..... GRANNY_SMITH야..." + APPLE_GRANNY_SMITH);
}
```

Item Int 상수 대신 열거 타입을 사용하라

기존 상수를 사용하는 시나리오를 살펴보면…

```
public static void main(String[] args) {
    System.out.println("나는....FUJI야.. 내 값은.." + APPLE_FUJI);
    System.out.println("나는.. PIPPIN이야... " + APPLE_PIPPIN);
    System.out.println("나는..... GRANNY_SMITH야..." + APPLE_GRANNY_SMITH);
}
```

이넘을 사용하면…

```
public enum Apple {
    FUJI(0), PIPPIN(1), GRANNY_SMITH(2);
    private final int value;

    Apple(int value) {
        this.value = value;
    }
}
```

Item Int 상수 대신 열거 타입을 사용하라

```
public static void main(String[] args) {
    for (Apple apple : Apple.values()) {
        System.out.println("나는~~~~ " + apple.name() + "이다~~~~ 내 값은~~~ " + apple.value);
    }
}
```

이넘을 사용하면…

```
나는~~~~ FUJI이다~~~~ 내 값은~~~ 0
나는~~~~ PIPPIN이다~~~~ 내 값은~~~ 1
나는~~~~ GRANNY_SMITH이다~~~~ 내 값은~~~ 2
```

```
public enum Apple {
    FUJI(0), PIPPIN(1), GRANNY_SMITH(2);
    private final int value;

    Apple(int value) {
        this.value = value;
    }
}
```

Item Int 상수 대신 열거 타입을 사용하라

```
public static void main(String[] args) {
    for (Apple apple : Apple.values()) {
        System.out.println("나는~~~~ " + apple.name() + "이다~~~~ 내 값은~~~ " + apple.value);
    }
}
```

이넘을 사용하면…

```
public enum Apple {
    FUJI(0), PIPPIN(1);
    private final int value;

    Apple(int value) {
        this.value = value;
    }
}
```

상수에 대한 직접적인 참조가 없는이상
상수가 하나 제거 되더라도 문제되지 않고…

Item Int 상수 대신 열거 타입을 사용하라

```
public static void main(String[] args) {
    for (Apple apple : Apple.values()) {
        System.out.println("나는~~~~ " + apple.name() + "이다~~~~ 내 값은~~~ " + apple.value);
    }
}
```

이넘을 사용하면…

```
나는~~~~ FUJI이다~~~~ 내 값은~~~ 0
나는~~~~ PIPPIN이다~~~~ 내 값은~~~ 1
```

```
public enum Apple {
    FUJI(0), PIPPIN(1);
    private final int value;

    Apple(int value) {
        this.value = value;
    }
}
```

상수에 대한 직접적인 참조가 없는이상
상수가 하나 제거 되더라도 문제되지 않고…

Item Int 상수 대신 열거 타입을 사용하라

```
public static void main(String[] args) {
    for (Apple apple : Apple.values()) {
        System.out.println("나는~~~~ " + apple.name() + "이다~~~~ 내 값은~~~ " + apple.value);
    }
}
```

이넘을 사용하면…

```
나는~~~~ FUJI이다~~~~ 내 값은~~~ 0
나는~~~~ PIPPIN이다~~~~ 내 값은~~~ 1
```

```
public enum Apple {
    FUJI(0), PIPPIN(1);
    private final int value;

    Apple(int value) {
        this.value = value;
    }
}
```

상수에 대한 직접적인 참조가 없는이상
상수가 하나 제거 되더라도 문제되지 않고…

만약 직접 참조가 있더라도, 컴파일 타임에 이를 잡아낼 수 있다!!

```
Error:(31, 41)  java: cannot find symbol
                  symbol:   variable GRANNY_SMITH
                  location: class Test.Apple
```

Item Int 상수 대신 열거 타입을 사용하라

만약 각자 상수별로 다른 기능을 수행해야하는 요구사항이 있다면?

Item Int 상수 대신 열거 타입을 사용하라

만약 각자 상수별로 다른 기능을 수행해야하는 요구사항이 있다면?

각 상수를 케이스로 구분하는것보다 우아한 방식이 있다. 추상 메소드를 선언해놓고
선언 바로 옆에 붙어있도록 한다면, 새로운 요구사항이 들어와도 까먹을 일도 없으며, 깔끔하게 구현가능하다!

Item Int 상수 대신 열거 타입을 사용하라

만약 각자 상수별로 다른 기능을 수행해야하는 요구사항이 있다면?

각 상수를 케이스로 구분하는것보다 우아한 방식이 있다. 추상 메소드를 선언해놓고
선언 바로 옆에 붙어있도록 한다면, 새로운 요구사항이 들어와도 까먹을 일도 없으며, 깔끔하게 구현가능하다!

```
public enum Operation {
     PLUS {
          public double test(double x, double y) {
               return x + y;
          }
     },
     MINUS {
          public double test(double x, double y) {
               return x + y;
          }
     },
     TIMES {
          public double test(double x, double y) {
               return x + y;
          }
     },
     DIVIDE {
          public double test(double x, double y) {
               return x + y;
          }
     };
     public abstract double test(double x, double y);
```

Item Int 상수 대신 열거 타입을 사용하라

만약 각자 상수별로 다른 기능을 수행해야하는 요구사항이 있다면?

각 상수를 케이스로 구분하는것보다 우아한 방식이 있다. 추상 메소드를 선언해놓고
선언 바로 옆에 붙어있도록 한다면, 새로운 요구사항이 들어와도 까먹을 일도 없으며, 깔끔하게 구현가능하다!

이를 잘 사용하면, 복잡한 비즈니스로직의 코드들도 Enum을 사용하는것 만으로도 많은 부분을 깔끔하게 구현 가능하다

```
public enum PayrollDay {
    MONDAY(PayType.WEEKDAY), TUESDAY(PayType.WEEKDAY), WENDSDAY(PayType.WEEKDAY), THURSDAY(PayType.WEEKDAY), FRIDAY(PayType.WEEKDAY),
    SATURDASY(PayType.WEEKEND), SUNDAY(PayType.WEEKEND);

    private final PayType payType;

    PayrollDay(PayType payType) {
        this.payType = payType;
    }

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

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

Item ordinal 메서드 대신 인스턴스 필드를 사용하라

Item ordinal 메서드 대신 인스턴스 필드를 사용하라

ordinal 메서드가 무엇인지부터…

열거 타입에서 몇번째 위치 인가를 알려주는 메소드!

Item ordinal 메서드 대신 인스턴스 필드를 사용하라

ordinal 메서드가 무엇인지부터…

열거 타입에서 몇번째 위치 인가를 알려주는 메소드!

```
/**
 * Returns the ordinal of this enumeration constant (its position
 * in its enum declaration, where the initial constant is assigned
 * an ordinal of zero).
 *
 * Most programmers will have no use for this method.  It is
 * designed for use by sophisticated enum-based data structures, such
 * as {@link java.util.EnumSet} and {@link java.util.EnumMap}.
 *
 * @return the ordinal of this enumeration constant
 */
@Range(from = 0, to = java.lang.Integer.MAX_VALUE )
public final int ordinal() {
    return ordinal;
}
```

Item ordinal 메서드 대신 인스턴스 필드를 사용하라

ordinal 메서드가 무엇인지부터…

열거 타입에서 몇번째 위치 인가를 알려주는 메소드!

```
/**
 * Returns the ordinal of this enumeration constant (its position
 * in its enum declaration, where the initial constant is assigned
 * an ordinal of zero).
 *
 * Most programmers will have no use for this method.  It is
 * designed for use by sophisticated enum-based data structures, such
 * as {@link java.util.EnumSet} and {@link java.util.EnumMap}.
 *
 * @return the ordinal of this enumeration constant
 */
@Range(from = 0, to = java.lang.Integer.MAX_VALUE )
public final int ordinal() {
    return ordinal;
}
```

Item ordinal 메서드 대신 인스턴스 필드를 사용하라

그럼 어떻게 쓰면 되나요?

```
/**
 * Returns the ordinal of this enumeration constant (its position
 * in its enum declaration, where the initial constant is assigned
 * an ordinal of zero).
 *
 * Most programmers will have no use for this method.  It is
 * designed for use by sophisticated enum-based data structures, such
 * as {@link java.util.EnumSet} and {@link java.util.EnumMap}.
 *
 * @return the ordinal of this enumeration constant
 */
@Range(from = 0, to = java.lang.Integer.MAX_VALUE )
public final int ordinal() {
    return ordinal;
}
```

Item ordinal 메서드 대신 인스턴스 필드를 사용하라

그럼 어떻게 쓰면 되나요?

필드에 선언해서 사용합시다!

```
public enum Number {
    ONE(0), TWO(1), THREE(2);

    private final int index;

    Number(int index) {
        this.index = index;
    }
}
```

```
/**
 * Returns the ordinal of this enumeration constant (its position
 * in its enum declaration, where the initial constant is assigned
 * an ordinal of zero).
 *
 * Most programmers will have no use for this method.  It is
 * designed for use by sophisticated enum-based data structures, such
 * as {@link java.util.EnumSet} and {@link java.util.EnumMap}.
 *
 * @return the ordinal of this enumeration constant
 */
@Range(from = 0, to = java.lang.Integer.MAX_VALUE )
public final int ordinal() {
    return ordinal;
}
```

Item 비트 필드 대신 EnumSet을 사용하라

Item 비트 필드 대신 EnumSet을 사용하라

글씨체 같은 Enum이 있다고 생각해보자…

STYLE_BOLD, STYLE_ITALIC, STYLE_UNDERLINE…

Item 비트 필드 대신 EnumSet을 사용하라

글씨체 같은 Enum이 있다고 생각해보자…

STYLE_BOLD, STYLE_ITALIC, STYLE_UNDERLINE…

이런애들은 중복해서 표현하고 싶을수가 있다.
STYLE_BOLD | STYLE_ITALIC 요런 식으로…

과거에는 이런 경우 비트필드를 사용해서 이런 요구사항을 해결했다. (비트연산은 합집합, 교집합 등에 대한 연산이 효율적!)

Item 비트 필드 대신 EnumSet을 사용하라

글씨체 같은 Enum이 있다고 생각해보자…

STYLE_BOLD, STYLE_ITALIC, STYLE_UNDERLINE…

이런애들은 중복해서 표현하고 싶을수가 있다.
STYLE_BOLD | STYLE_ITALIC 요런 식으로…

과거에는 이런 경우 비트필드를 사용해서 이런 요구사항을 해결했다. (비트연산은 합집합, 교집합 등에 대한 연산이 효율적!)

STYLE_BOLD = 0001
STYLE_ITALIC = 0010

STYLE_BOLD | STYLE_ITALIC = 0011

Item 비트 필드 대신 EnumSet을 사용하라

글씨체 같은 Enum이 있다고 생각해보자…

STYLE_BOLD, STYLE_ITALIC, STYLE_UNDERLINE…

이런애들은 중복해서 표현하고 싶을수가 있다.
STYLE_BOLD | STYLE_ITALIC 요런 식으로…

과거에는 이런 경우 비트필드를 사용해서 이런 요구사항을 해결했다. (비트연산은 합집합, 교집합 등에 대한 연산이 효율적!)

STYLE_BOLD = 0001
STYLE_ITALIC = 0010

STYLE_BOLD | STYLE_ITALIC = 0011

성능은 우수하지만… 몇가지 아쉬운점이있다.

1. 비트 필드값으로 그대로 출력되는 상황에서 해석이 어렵다
2. 비트 필드 값을 순회하는 상황도 구현하기 까다롭다
3. 최대 몇비트가 필요한지 API작성시 미리 예측해서 적정 타입을 선택해야한다

Item 비트 필드 대신 EnumSet을 사용하라

글씨체 같은 Enum이 있다고 생각해보자…

STYLE_BOLD, STYLE_ITALIC, STYLE_UNDERLINE…

이런얘들은 중복해서 표현하고 싶을수가 있다.
STYLE_BOLD | STYLE_ITALIC 요런 식으로…

과거에는 이런 경우 비트필드를 사용해서 이런 요구사항을 해결했다. (비트연산은 합집합, 교집합 등에 대한 연산이 효율적!)

STYLE_BOLD = 0001
STYLE_ITALIC = 0010

STYLE_BOLD | STYLE_ITALIC = 0011

성능은 우수하지만… 몇가지 아쉬운점이있다.

1. 비트 필드값으로 그대로 출력되는 상황에서 해석이 어렵다
2. 비트 필드 값을 순회하는 상황도 구현하기 까다롭다
3. 최대 몇비트가 필요한지 API작성시 미리 예측해서 적정 타입을 선택해야한다

이런 문제를 해결해주는 녀석이 바로 **EnumSet**!!

Item 비트 필드 대신 EnumSet을 사용하라

글씨체 같은 Enum이 있다고 생각해보자…

STYLE_BOLD, STYLE_ITALIC, STYLE_UNDERLINE…

이런얘들은 중복해서 표현하고 싶을수가 있다.
STYLE_BOLD | STYLE_ITALIC 요런 식으로…

과거에는 이런 경우 비트필드를 사용해서 이런 요구사항을 해결했다. (비트연산은 합집합, 교집합 등에 대한 연산이 효율적!)

STYLE_BOLD = 0001
STYLE_ITALIC = 0010

STYLE_BOLD | STYLE_ITALIC = 0011

성능은 우수하지만… 몇가지 아쉬운점이있다.

1. 비트 필드값으로 그대로 출력되는 상황에서 해석이 어렵다
2. 비트 필드 값을 순회하는 상황도 구현하기 까다롭다
3. 최대 몇비트가 필요한지 API작성시 미리 예측해서 적정 타입을 선택해야한다

이런 문제를 해결해주는 녀석이 바로 **EnumSet**!!
내부적으로는 **비트벡터로 구현**되어있어 성능이 우수하고
Set 인터페이스를 완벽히 구현하였고
순회하는 상황, 삭제해야하는 상황 또한 **메소드로 지원**해준다!!

Item 비트 필드 대신 EnumSet을 사용하라

```
public static class Text {
   public enum Style {BOLD, ITALIC, UNDERLINE;}

   public void applyStyles(Set<Style> styles) {
      // ...
   }
}
```

API

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

Client

Item 비트 필드 대신 EnumSet을 사용하라

```
public static class Text {
   public enum Style {BOLD, ITALIC, UNDERLINE;}

   public void applyStyles(Set<Style> styles) {
      // ...
   }
}
```

API

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

Client

Item ordinal 인덱싱 대신 EnumMap을 사용하라

Item ordinal 인덱싱 대신 EnumMap을 사용하라

```
public class Plant {
     enum LifeCycle {ANNUAL, PERMERNIAL, BIENNIAL}

     final String name;
     final LifeCycle lifeCycle;

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

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

Item ordinal 인덱싱 대신 EnumMap을 사용하라

```
public class Plant {
     enum LifeCycle {ANNUAL, PERMERNIAL, BIENNIAL}

     final String name;
     final LifeCycle lifeCycle;

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

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

```
public static void main(String[] args) {

     List<Plant> garden = new ArrayList();
     garden.add(new Plant("test1", Plant.LifeCycle.ANNUAL));
     garden.add(new Plant("test2", Plant.LifeCycle.BIENNIAL));

     Set<Plant>[] plantsByLifeCycle = (Set<Plant>[])new Set[Plant.LifeCycle.values().length];

     for (int i = 0; i < plantsByLifeCycle.length; i++) {
          plantsByLifeCycle[i] = new HashSet<>();
     }

     for (Plant p : garden) {
          plantsByLifeCycle[p.lifeCycle.ordinal()].add(p);
     }

     for(int i= 0; i<plantsByLifeCycle.length;i++) {
          System.out.printf("%s: %s%n", Plant.LifeCycle.values()[i], plantsByLifeCycle[i]);
     }
}
```

Item ordinal 인덱싱 대신 EnumMap을 사용하라

```
public class Plant {
     enum LifeCycle {ANNUAL, PERMERNIAL, BIENNIAL}

     final String name;
     final LifeCycle lifeCycle;

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

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

```
public static void main(String[] args) {

     List<Plant> garden = new ArrayList();
     garden.add(new Plant("test1", Plant.LifeCycle.ANNUAL));
     garden.add(new Plant("test2", Plant.LifeCycle.BIENNIAL));

     Set<Plant>[] plantsByLifeCycle = (Set<Plant>[])new Set[Plant.LifeCycle.values().length];

     for (int i = 0; i < plantsByLifeCycle.length; i++) {
          plantsByLifeCycle[i] = new HashSet<>();
     }

     for (Plant p : garden) {
          plantsByLifeCycle[p.lifeCycle.ordinal()].add(p);
     }

     for(int i= 0; i<plantsByLifeCycle.length;i++) {
          System.out.printf("%s: %s%n", Plant.LifeCycle.values()[i], plantsByLifeCycle[i]);
     }
}
```

배열은 제너릭과 호환되지 않으므로 비검사 형변환(아이템 28에 나오는)을 수행해야되고…

Item ordinal 인덱싱 대신 EnumMap을 사용하라

```
public class Plant {
     enum LifeCycle {ANNUAL, PERMERNIAL, BIENNIAL}

     final String name;
     final LifeCycle lifeCycle;

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

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

```
public static void main(String[] args) {

     List<Plant> garden = new ArrayList();
     garden.add(new Plant("test1", Plant.LifeCycle.ANNUAL));
     garden.add(new Plant("test2", Plant.LifeCycle.BIENNIAL));

     Set<Plant>[] plantsByLifeCycle = (Set<Plant>[])new Set[Plant.LifeCycle.values().length];

     for (int i = 0; i < plantsByLifeCycle.length; i++) {
          plantsByLifeCycle[i] = new HashSet<>();
     }

     for (Plant p : garden) {
          plantsByLifeCycle[p.lifeCycle.ordinal()].add(p);
     }

     for(int i= 0; i<plantsByLifeCycle.length;i++) {
          System.out.printf("%s: %s%n", Plant.LifeCycle.values()[i], plantsByLifeCycle[i]);
     }
}
```

배열은 제너릭과 호환되지 않으므로 비검사 형변환(아이템 28에 나오는)을 수행해야되고…
배열은 각 인덱스의 의미를 모르니 출력 결과에 직접 레이블을 달아야하고…

Item ordinal 인덱싱 대신 EnumMap을 사용하라

```
public class Plant {
     enum LifeCycle {ANNUAL, PERMERNIAL, BIENNIAL}

     final String name;
     final LifeCycle lifeCycle;

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

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

```
public static void main(String[] args) {

     List<Plant> garden = new ArrayList();
     garden.add(new Plant("test1", Plant.LifeCycle.ANNUAL));
     garden.add(new Plant("test2", Plant.LifeCycle.BIENNIAL));

     Set<Plant>[] plantsByLifeCycle = (Set<Plant>[])new Set[Plant.LifeCycle.values().length];

     for (int i = 0; i < plantsByLifeCycle.length; i++) {
          plantsByLifeCycle[i] = new HashSet<>();
     }

     for (Plant p : garden) {
          plantsByLifeCycle[p.lifeCycle.ordinal()].add(p);
     }

     for(int i= 0; i<plantsByLifeCycle.length;i++) {
          System.out.printf("%s: %s%n", Plant.LifeCycle.values()[i], plantsByLifeCycle[i]);
     }
}
```

배열은 제너릭과 호환되지 않으므로 비검사 형변환(아이템 28에 나오는)을 수행해야되고…
배열은 각 인덱스의 의미를 모르니 출력 결과에 직접 레이블을 달아야하고…
정확한 정수값을 사용한다는 것을 보장할 수 없으니까(제일 치명적인 상황) 런타임에 버그로 등장할 수 있고,
운이 좋아야 ArrayIndexOutOfBounds로 트래킹 되는.. 상황

Item ordinal 인덱싱 대신 EnumMap을 사용하라

```
public class Plant {
     enum LifeCycle {ANNUAL, PERMERNIAL, BIENNIAL}

     final String name;
     final LifeCycle lifeCycle;

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

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

```
public static void main(String[] args) {

     List<Plant> garden = new ArrayList();
     garden.add(new Plant("test1", Plant.LifeCycle.ANNUAL));
     garden.add(new Plant("test2", Plant.LifeCycle.BIENNIAL));

     Set<Plant>[] plantsByLifeCycle = (Set<Plant>[])new Set[Plant.LifeCycle.values().length];

     for (int i = 0; i < plantsByLifeCycle.length; i++) {
          plantsByLifeCycle[i] = new HashSet<>();
     }

     for (Plant p : garden) {
          plantsByLifeCycle[p.lifeCycle.ordinal()].add(p);
     }

     for(int i= 0; i<plantsByLifeCycle.length;i++) {
          System.out.printf("%s: %s%n", Plant.LifeCycle.values()[i], plantsByLifeCycle[i]);
     }
}
```

배열은 제너릭과 호환되지 않으므로 비검사 형변환(아이템 28에 나오는)을 수행해야되고…
배열은 각 인덱스의 의미를 모르니 출력 결과에 직접 레이블을 달아야하고…
정확한 정수값을 사용한다는 것을 보장할 수 없으니까(제일 치명적인 상황) 런타임에 버그로 등장할 수 있고,
운이 좋아야 ArrayIndexOutOfBounds로 트래킹 되는.. 상황

을 해결하는 대안이 바로 **EnumMap**!

Item ordinal 인덱싱 대신 EnumMap을 사용하라

```
public class Plant {
     enum LifeCycle {ANNUAL, PERMERNIAL, BIENNIAL}

     final String name;
     final LifeCycle lifeCycle;

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

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

```
public static void main(String[] args) {

     List<Plant> garden = new ArrayList();
     garden.add(new Plant("test1", Plant.LifeCycle.ANNUAL));
     garden.add(new Plant("test2", Plant.LifeCycle.BIENNIAL));

     Set<Plant>[] plantsByLifeCycle = (Set<Plant>[])new Set[Plant.LifeCycle.values().length];

     for (int i = 0; i < plantsByLifeCycle.length; i++) {
          plantsByLifeCycle[i] = new HashSet<>();
     }

     for (Plant p : garden) {
          plantsByLifeCycle[p.lifeCycle.ordinal()].add(p);
     }

     for(int i= 0; i<plantsByLifeCycle.length;i++) {
          System.out.printf("%s: %s%n", Plant.LifeCycle.values()[i], plantsByLifeCycle[i]);
     }
}
```

```
List<Plant> garden = new ArrayList();
garden.add(new Plant("test1", Plant.LifeCycle.ANNUAL));
garden.add(new Plant("test2", Plant.LifeCycle.BIENNIAL));

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

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);
```

배열은 제너릭과 호환되지 않으므로 비검사 형변환(아이템 28에 나오는)을 수행해야되고…
배열은 각 인덱스의 의미를 모르니 출력 결과에 직접 레이블을 달아야하고…
정확한 정수값을 사용한다는 것을 보장할 수 없으니까(제일 치명적인 상황) 런타임에 버그로 등장할 수 있고,
운이 좋아야 ArrayIndexOutOfBounds로 트래킹 되는.. 상황

을 해결하는 대안이 바로 **EnumMap**!

Item ordinal 인덱싱 대신 EnumMap을 사용하라

```
public class Plant {
     enum LifeCycle {ANNUAL, PERMERNIAL, BIENNIAL}

     final String name;
     final LifeCycle lifeCycle;

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

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

```
public static void main(String[] args) {

     List<Plant> garden = new ArrayList();
     garden.add(new Plant("test1", Plant.LifeCycle.ANNUAL));
     garden.add(new Plant("test2", Plant.LifeCycle.BIENNIAL));

     Set<Plant>[] plantsByLifeCycle = (Set<Plant>[])new Set[Plant.LifeCycle.values().length];

     for (int i = 0; i < plantsByLifeCycle.length; i++) {
          plantsByLifeCycle[i] = new HashSet<>();
     }

     for (Plant p : garden) {
          plantsByLifeCycle[p.lifeCycle.ordinal()].add(p);
     }

     for(int i= 0; i<plantsByLifeCycle.length;i++) {
          System.out.printf("%s: %s%n", Plant.LifeCycle.values()[i], plantsByLifeCycle[i]);
     }
}
```

```
List<Plant> garden = new ArrayList();
garden.add(new Plant("test1", Plant.LifeCycle.ANNUAL));
garden.add(new Plant("test2", Plant.LifeCycle.BIENNIAL));

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

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);
```

코드도 단순해졌고

Map 객체에서 제공하는 메소드들을 활용하므로

Item ordinal 인덱싱 대신 EnumMap을 사용하라

```
public class Plant {
     enum LifeCycle {ANNUAL, PERMERNIAL, BIENNIAL}

     final String name;
     final LifeCycle lifeCycle;

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

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

```
public static void main(String[] args) {

     List<Plant> garden = new ArrayList();
     garden.add(new Plant("test1", Plant.LifeCycle.ANNUAL));
     garden.add(new Plant("test2", Plant.LifeCycle.BIENNIAL));

     Set<Plant>[] plantsByLifeCycle = (Set<Plant>[])new Set[Plant.LifeCycle.values().length];

     for (int i = 0; i < plantsByLifeCycle.length; i++) {
          plantsByLifeCycle[i] = new HashSet<>();
     }

     for (Plant p : garden) {
          plantsByLifeCycle[p.lifeCycle.ordinal()].add(p);
     }

     for(int i= 0; i<plantsByLifeCycle.length;i++) {
          System.out.printf("%s: %s%n", Plant.LifeCycle.values()[i], plantsByLifeCycle[i]);
     }
}
```

```
List<Plant> garden = new ArrayList();
garden.add(new Plant("test1", Plant.LifeCycle.ANNUAL));
garden.add(new Plant("test2", Plant.LifeCycle.BIENNIAL));

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

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);
```

코드도 단순해졌고

Map 객체에서 제공하는 메소드들을 활용하므로

Item ordinal 인덱싱 대신 EnumMap을 사용하라

```
public class Plant {
     enum LifeCycle {ANNUAL, PERMERNIAL, BIENNIAL}

     final String name;
     final LifeCycle lifeCycle;

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

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

```
public static void main(String[] args) {

     List<Plant> garden = new ArrayList();
     garden.add(new Plant("test1", Plant.LifeCycle.ANNUAL));
     garden.add(new Plant("test2", Plant.LifeCycle.BIENNIAL));

     Set<Plant>[] plantsByLifeCycle = (Set<Plant>[])new Set[Plant.LifeCycle.values().length];

     for (int i = 0; i < plantsByLifeCycle.length; i++) {
          plantsByLifeCycle[i] = new HashSet<>();
     }

     for (Plant p : garden) {
          plantsByLifeCycle[p.lifeCycle.ordinal()].add(p);
     }

     for(int i= 0; i<plantsByLifeCycle.length;i++) {
          System.out.printf("%s: %s%n", Plant.LifeCycle.values()[i], plantsByLifeCycle[i]);
     }
}
```

```
List<Plant> garden = new ArrayList();
garden.add(new Plant("test1", Plant.LifeCycle.ANNUAL));
garden.add(new Plant("test2", Plant.LifeCycle.BIENNIAL));

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

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);
```

코드도 단순해졌고, 성능도 비슷하고

내부적으로 배열을 사용하므로(내부 구현방식을 안으로 숨긴것이 핵심 포인트)

Item ordinal 인덱싱 대신 EnumMap을 사용하라

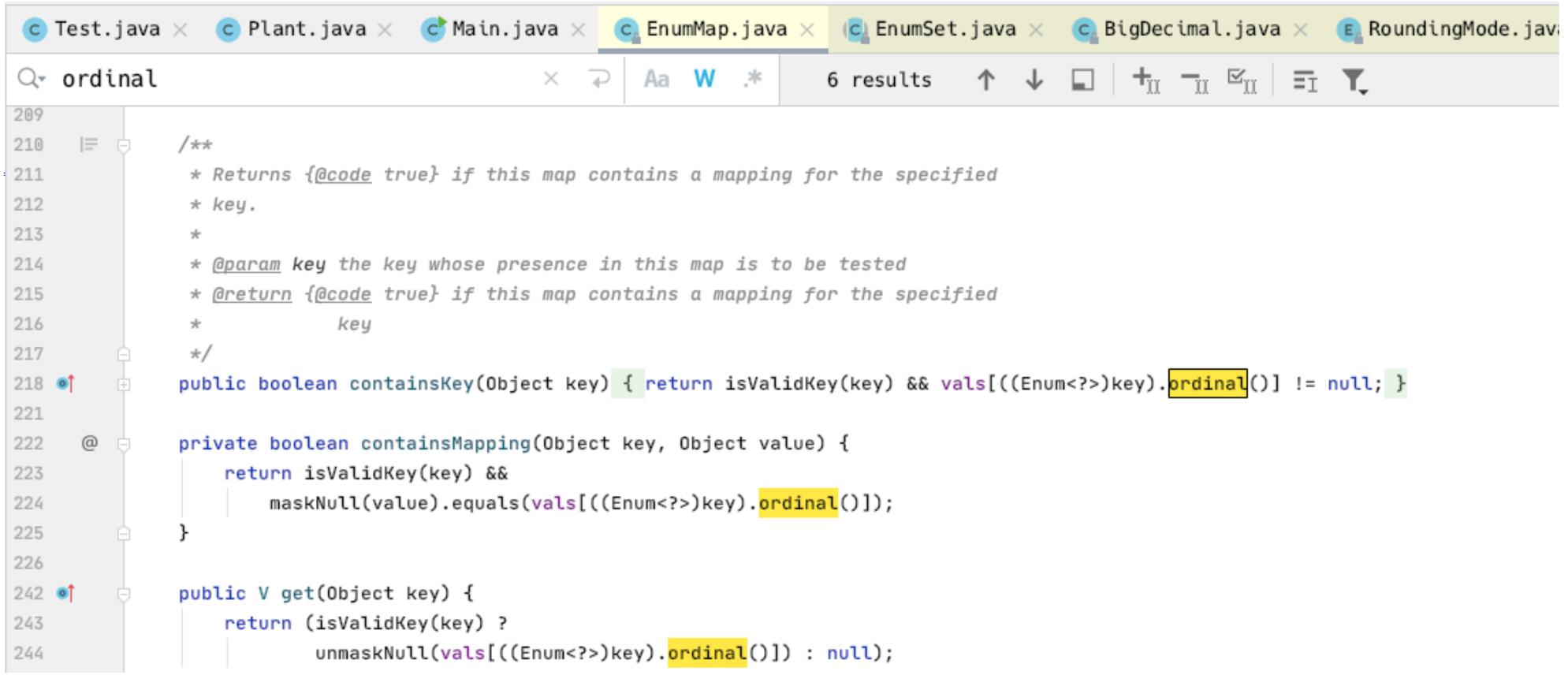
```
public class Plant {
     enum LifeCycle {ANNUAL, PERMERNIAL, BIENNIAL}

     final String name;
     final LifeCycle lifeCycle;

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

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

```
public static void main(String[] args) {

     List<Plant> garden = new ArrayList();
     garden.add(new Plant("test1", Plant.LifeCycle.ANNUAL));
     garden.add(new Plant("test2", Plant.LifeCycle.BIENNIAL));

     Set<Plant>[] plantsByLifeCycle = (Set<Plant>[])new Set[Plant.LifeCycle.values().length];

     for (int i = 0; i < plantsByLifeCycle.length; i++) {
          plantsByLifeCycle[i] = new HashSet<>();
     }

     for (Plant p : garden) {
          plantsByLifeCycle[p.lifeCycle.ordinal()].add(p);
     }

     for(int i= 0; i<plantsByLifeCycle.length;i++) {
          System.out.printf("%s: %s%n", Plant.LifeCycle.values()[i], plantsByLifeCycle[i]);
     }
}
```

```
List<Plant> garden = new ArrayList();
garden.add(new Plant("test1", Plant.LifeCycle.ANNUAL));
garden.add(new Plant("test2", Plant.LifeCycle.BIENNIAL));

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

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);
```

코드도 단순해졌고, 성능도 비슷하고, **안전하다**!

Item ordinal 인덱싱 대신 EnumMap을 사용하라



내부적으로 ordinal을 사용하고 있다!

Item

끝!