Emulate extensible enums with interface

책 초판에서 설명한 typesafe enum pattern에 비해 열거 타입이 대부분 우수하나, 단 하나의 단점을 가지고 있습니다.열거 타입은 확장하기 어렵다는 것이죠.

다시 말해서, 열거타입의 값을 그대로 가져와서 다른 목적으로 사용하는 방법을 말합니다.

보통 확장은 안좋은 경우이기 때문에 Enum 설계에 반영하지 않았습니다.

1. extention type이 base type의 원소는 맞으나, 그 역은 성립되지 않는다는건 이상하기 때문이죠.
2. base type 및 extention type 순회가 어렵고,
3. 확장성을 높히려면 고려할 요소가 늘어나서 설계가 어려워집니다.

그럼에도 확장 열거 타입이 적절한 예

하지만 확장할 수 있는 열거타입이 적절한 예는 다음과 같습니다. **opcode**가 바로 그 예이지요. API가 제공하는 기본 연산 외에도 사용자 확장 연산을 지원해줘야할 때가 있습니다.

확장 열거 타입을 구현하는 방법

열거타입이 인터페이스를 구현할 수 있다는 사실을 이용하면 됩니다.

1. 연산코드용 인터페이스를 정의하고
2. 열거 타입이 이 인터페이스를 구현하도록 합니다.

```
// Emulated extensible enum using an interface
public interface Operation {
  double apply(double x, double y);
}
public enum BasicOperation implements Operation {
  PLUS("+") {
    public double apply(double x, double y) { return x + y; }
  },
  MINUS("-") {
    public double apply(double x, double y) { return x - y; }
  },
  TIMES("*") {
    public double apply(double x, double y) { return x * y; }
  },
  DIVIDE("/") {
    public double apply(double x, double y) { return x / y; }
```

```
  };
```

BaseOperation은 확장하기 어려우나, Operation interface를 이용하면 확장 가능하지요.

이제 확장된 Enum을 작성해보겠습니다.

###

```
// Emulated extension enum
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;

  ExtendedOperation(String symbol) {
    this.symbol = symbol;
  }
  @Override public String toString() {
    return symbol;
  }
}
```

Operation을 구현하면 Subclass이기 때문에 BaseOperation을 대체할 수 있게 됩니다.

그리고 Item-34처럼 abstraction을 enum에 두지 않았으므로 따로 apply 추상메서드를 선언하지 않아도 됩니다.

그러면 확장된 열거타입으로 순회 및 연산 적용 방법을 말씀드리겠습니다.

방법 1

*개별 인스턴스 수준 뿐만 아니라 **타입 수준**에서도 기본 열거 타입 대신 확장된 열거 타입의 원소를 사용할 수 있습니다.*

```
public static void main(String[] args) {
    double x = Double.parseDouble(args[0]);
    double y = Double.parseDouble(args[1]);
    test(ExtendedOperation.class, x, y);
}

private static <T extends Enum<T> & Operation> void test(Class<T> opEnumType,
double x, double y) {
    for (Operation op : opEnumType.getEnumConstants())
        System.out.printf("%f %s %f = %f%n",
                x, op, y, op.apply(x, y));
}
```

1. 이 예제는 boundded type token을 넘겨서 **그 타입**으로 확장된 연산들을 알려줍니다.
2. <T extends Enum & Operation> Class객체가 Enum타입인 동시에 Operation의 subtype이여야 한다는 뜻입니다.

   열거 타입이여야 원소를 순회하고, Operation이어야 원소가 정의한 apply연산을 수행할 수 있기 때문이죠. 다시말해서 객체를 타입에 담고, 객체의 method를 실행하기 위해서라고 말할 수 있겠습니다.

방법 2

위 원소의 순회 및 연산 수행을 boundded wildcard를 이용할 수도 있습니다. 이를 이용하면 코드가 간결해지죠

```
public static void main(String[] args) {
    double x = Double.parseDouble(args[0]);
    double y = Double.parseDouble(args[1]);
    test(Arrays.asList(ExtendedOperation.values()), x, y);
}

private static void test(Collection<? extends Operation> opSet, double x,
double y) {
    for (Operation op : opSet)
        System.out.printf("%f %s %f = %f%n", x, op, y, op.apply(x, y));
}
```

1. 코드가 조금더 유연해졌습니다.

   test에 여러 확장된 연산을 조합해 호출할 수 있게 되었습니다.
   Arrays.asList(ExtendedOperation1.values(), ExtendedOperation2.values()) 이런식으로 사용할 수 있게 되었단 뜻이죠.
2. 한편으로는 특정 연산에서는 EnumSet과 EnumMap을 사용할 수 없습니다. Operation의 서브 클래스로 수행test를 수행하니까요.

두 대안으로 출력한 값은 다음과 같습니다.

```
4.000000 ^ 2.000000 = 16.000000
4.000000 % 2.000000 = 0.000000
```

인터페이스 확장 열거 타입의 문제점

열거 타입끼리 구현을 상속하기 어렵습니다.

아무 상태에도 의존하지 않는 경우에는 default 구현으로 인터페이스에 추가할 수 있고,

공유하는 기능이 많다면 별도 helper class나 helper method로 분리하여 코드 중복을 피할 수 있습니다.

java library에서 이번 아이템을 적용한 예