Builder Pattern

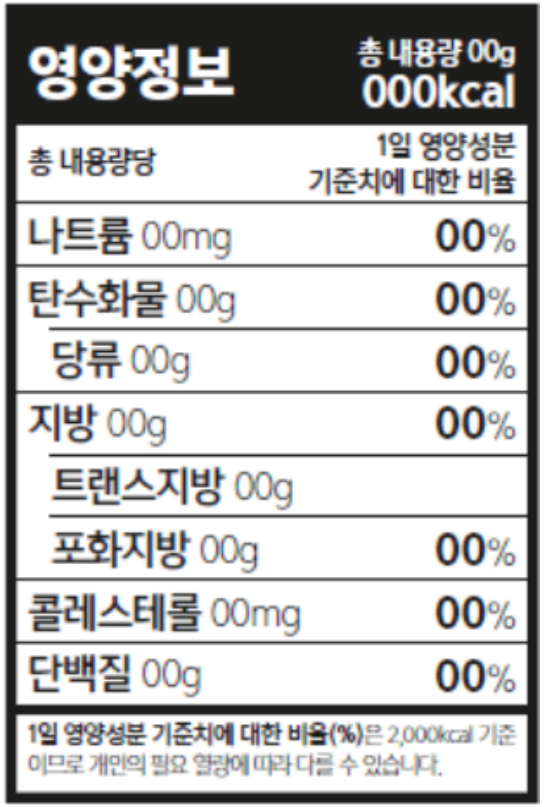
“ 객체 생성을 깔끔하고 유연하게 하기 위한 기법 ”

“ 빌더 패턴을 사용하는 이유..? ”

› **코드 가독성**

› **객체 일관성**

“ 인자값이 많은 경우, 선택적 인자들이 많은 경우, 추가될 인자가 많은 경우… ”



영양정보	총 내용량 00g 000kcal
총 내용량당	1일 영양성분 기준치에 대한 비율
나트륨 00mg	00%
탄수화물 00g	00%
당류 00g	00%
지방 00g	00%
트랜스지방 00g	
포화지방 00g	00%
콜레스테롤 00mg	00%
단백질 00g	00%

1일 영양성분 기준치에 대한 비율(%)은 2,000kcal 기준이므로 개인의 필요 열량에 따라 다를 수 있습니다.

영양성분표 (출처: 식품의약품안전처)

필수 입력 정보 : **탄수화물, 단백질, 지방**

선택적 입력 정보 : **나트륨, 콜레스테롤**

점층적 생성자 패턴

생성자 오버로딩을 통해 조절
but, 영양소가 많아질때마다 매개변수가 많아지고 호출할때도 많은걸 호출해야함.

```java
    public NutritionFacts_overload(int carbohydrate, int protein, int fat) {
        this.carbohydrate = carbohydrate;
        this.protein = protein;
        this.fat = fat;
    }

    public NutritionFacts_overload(int carbohydrate, int protein, int fat, int sodium) {
        this.carbohydrate = carbohydrate;
        this.protein = protein;
        this.fat = fat;
        this.sodium = sodium;
    }

    public NutritionFacts_overload(int carbohydrate, int protein, int fat, int sodium, int cholesterol) {
        this.carbohydrate = carbohydrate;
        this.protein = protein;
        this.fat = fat;
        this.sodium = sodium;
        this.cholesterol = cholesterol;
    }
```

인자들이 많아질수록
생성자가 많아진다

```java
public static void main(String[] args) {
    NutritionFacts_overload nutritionFacts = new NutritionFacts_overload(200, 100, 300, 50, 32);
}
```

코드 가독성은 떨어짐

자바빈 패턴

setter를 활용해 인자값주기

```java
    public NutritionFacts_javaBean() {}

    public void setCarbohydrate(int carbohydrate) {
        this.carbohydrate = carbohydrate;
    }
    public void setProtein(int protein) {
        this.protein = protein;
    }
    public void setFat(int fat) {
        this.fat = fat;
    }
    public void setSodium(int sodium) {
        this.sodium = sodium;
    }
    public void setCholesterol(int cholesterol) {
        this.cholesterol = cholesterol;
    }
```

Setter를 사용하는 방식

객체 일관성(Consistency) 문제

setter는 언제 어디서든 내가 인자값을 바꿔줄 수 있는데
이는 곧 객체가 계속 바뀔 수 있는 여지가 있다.
=> 객체 일관성 문제

```java
public static void main(String[] args) {
    NutritionFacts_javaBean nutritionFacts = new NutritionFacts_javaBean();
    nutritionFacts.setCarbohydrate(200);
    nutritionFacts.setProtein(100);
    nutritionFacts.setFat(300);
    nutritionFacts.setSodium(50);
    nutritionFacts.setCholesterol(32);
}
```

코드 가독성 문제는 해결

Builder Pattern ..?

“ 점층적 생성자 패턴과 자바빈즈 패턴의 단점을 보완한 것”

빌더 패턴

```java
public class NutritionFacts {
    private final int carbohydrate;
    private final int protein;
    private final int fat;
    private final int sodium;
    private final int cholesterol;

    public static class Builder { // static nested class 생성
        // 필수 인자
        private final int carbohydrate;
        private final int protein;
        private final int fat;
        // 선택적 인자
        private int sodium      = 0;
        private int cholesterol = 0;

        // 필수 생성자
        public Builder(int carbohydrate, int protein, int fat) {
            this.carbohydrate = carbohydrate;
            this.protein = protein;
            this.fat = fat;
        }
        // 선택적 메소드
        public Builder sodium(int val) {
            sodium = val;
            return this;
        }

        public Builder cholesterol(int val) {
            cholesterol = val;
            return this;
        }
```

필수 인자를 final로 한 이유 :
private final은 "동결"의 의미.
바꾸지 않게 하기 위해.

나트륨과 콜레스트롤이 없는 제품을 위해
디폴트값을 0으로 주고 나트륨과 콜레스트롤이
있는 제품에 한해 값을 바꿔줌

method chaining을 위해 this로 return
얘를 호출한 객체 자체를 리턴한다

빌더 클래스를 static class로 생성

필수값들에 대한 생성자는 public으로 선언, 필수값을 파라미터로 받아준다.

선택 값들에 대해서는 각각의 속성마다 메소드 제공
메소드의 리턴 값이 빌더 객체 자신 (this)

빌더 패턴

```
36
37        public NutritionFacts build() {
38            return new NutritionFacts(this);
39        }
40    }
41
42    private NutritionFacts(Builder builder) {
43        carbohydrate    = builder.carbohydrate;
44        protein         = builder.protein;
45        fat             = builder.fat;
46        sodium          = builder.sodium;
47        cholesterol     = builder.cholesterol;
48    }
49 }
```

원하는 값들이 다 세팅되면 이 함수를 실행해 새로운 객체를 만들어준다.

빌더 클래스 내에 build() 메소드를 정의하여 클라이언트 에 반환

NutritionFacts 형을 반환하는 build() 메소드로 현재 만든 build 객체를 넘겨주어 NutritionFacts의 생성자를 호출한다.
NutritionFacts의 생성자 내에선 builder(this)를 매개로 builder 클래스의 멤버변수를 NutritionFacts의 멤버변수와 같게 만들어준다.

```
    NutritionFacts cocaCola = new NutritionFacts
        .Builder(200, 100, 300)      //필수값 입력
        .sodium(50)
        .cholesterol(32)
        .build();
}
```

메소드 체이닝을 이용한 호출

cocaCola.builder,
cocaCola.sodium,
cocaCola.cholesterol,
cocaCola.build()
메소드 호출을 간단히함

빌더 패턴

› 장점

점층적 생성자 패턴이 가진 코드 가독성 문제,
자바빈즈 패턴이 가진 객체 불변성 문제를 해결

› 단점

코드량이 줄어드는 것은 아니다.
Builder 객체를 추가로 만들면서 성능이 낮아질 수 있다.

```
NutritionFacts_builder cocaCola = new NutritionFacts_builder
        .Builder(200, 100, 300)      //필수값 입력
        .sodium(50)
        .cholesterol(32)
        .build();
```

```
public static void main(String[] args) {
    NutritionFacts_javaBean nutritionFacts = new NutritionFacts_javaBean();
    nutritionFacts.setCarbohydrate(200);
    nutritionFacts.setProtein(100);
    nutritionFacts.setFat(300);
    nutritionFacts.setSodium(50);
    nutritionFacts.setSodium(32);
}
```

```
public static void main(String[] args) {
    NutritionFacts_overload nutritionFacts = new NutritionFacts_overload(200, 100, 300, 50, 32);
}
```

그래서 Builder Pattern은 실무에서 쓰이는가…?

Introduction

Retrofit turns your HTTP API into a Java interface.

```
public interface GitHubService {
  @GET("users/{user}/repos")
  Call<List<Repo>> listRepos(@Path("user") String user);
}
```

The `Retrofit` class generates an implementation of the `GitHubService` interface.

```
Retrofit retrofit = new Retrofit.Builder()
    .baseUrl("https://api.github.com/")
    .build();

GitHubService service = retrofit.create(GitHubService.class);
```

THANK YOU

1.빌더 클래스를 Static Nested Class로 생성합니다. 이때, 관례적으로 생성하고자 하는 클래스 이름 뒤에 Builder를 붙입니다. 예를 들어, Computer 클래스에 대한 빌더 클래스의 이름은 ComputerBuilder 라고 정의합니다.

2.빌더 클래스의 생성자는 public으로 하며, 필수 값들에 대해 생성자의 파라미터로 받습니다.

3.옵셔널한 값들에 대해서는 각각의 속성마다 메소드로 제공하며, 이때 중요한 것은 메소드의 리턴 값이 빌더 객체 자신이어야 합니다. =〉 코드상에서 return this 로 넘겨준다.

4.마지막 단계로, 빌더 클래스 내에 build() 메소드를 정의하여 클라이언트 프로그램에게 최종 생성된 결과물을 제공합니다. 이렇듯 build()를 통해서만 객체 생성을 제공하기 때문에 생성 대상이 되는 클래스의 생성자는 private으로 정의해야 합니다.

빌더 패턴 정리해야할 사항

Q1. 생성자처럼 빌더는 자신의 매개변수에 불변규칙을 적용할 수 있고 build 메소드는 그런 불변규칙을 검사할 수 있다?

~~Q1-1. setter 메소드를 사용하면 변경가능하다? setter를 이용해서 마음대로 변수를 변경할 수 있다?~~

~~Q2. 유연성이 좋다. 하나의 빌더는 여러 개의 객체를 생성하는데 사용될 수 있으며, 이러한 과정 중에 빌더의 매개변수는 다양하게 조정될 수 있다? => 필수 인자 + 선택인자 개수 자유롭게 설정이 가능하다.~~

Q3. 매개변수들의 값이 설정된 빌더는 훌륭한 추상 팩토리를 만든다. 즉, 클라이언트 코드에서는 그런 빌더를 메소드로 전달하여 그 메소드에서 하나 이상의 객체를 생성하게 할 수 있다?

https://hashcode.co.kr/questions/887/%EC%9E%90%EB%B0%94%EC%97%90%EC%84%9C-builder%EB%A5%BC-%EC%93%B0%EB%8A%94-%EC%9D%B4%EC%9C%A0%EB%8A%94-%EB%AD%94%EA%B0%80%EC%9A%94