# Item 02 - Consider a builder when faced with many constructor parameters

## 개요

static factories나 constructors는 매우 큰 optional parameters에 대해 scale하기 어렵습니다.

## 종류

### 1. Telescoping constructor pattern

#### 방법 설명

아주 모든 parameter를 받는 constructor를 작성하는 것입니다.

https://github.com/jbloch/effective-java-3e-source-code/blob/master/src/effectivejava/chapter2/item2/telescopingconstructor/NutritionFacts.java

#### 단점

parameter가 거대해질 때 쓰거나 읽기가 어려워집니다.

### 2. JavaBeans pattern

#### 방법 설명

default constructor를 만들고 setter method를 통해 필요한 parameter와 상황에 따라 optional parameter를 설정해줍니다.

https://github.com/jbloch/effective-java-3e-source-code/blob/master/src/effectivejava/chapter2/item2/javabeans/NutritionFacts.java

telescoping의 단점은 해결되었으나, 아래와 같은 치명적인 문제가 발생합니다.

#### 단점

construction이 여러 call로 분리되어 있기 때문에 생성하는 동안 상태가 일정하지 않을 수 있습니다.

constructor parameters의 일관성을 검증하는 기능 또한 없지요. 따라서 이는 버그를 가지고 있지만 해결하기 어려운 코드를 만들 수 있습니다.

JavaBeanPattern은 immutable하는 것을 배제하죠. 따라서 thread safety를 위한 방법을 넣어야 합니다.

### 변형 - freezing

construction이 완료될 때까지 수동으로 object를 freezing해주는 것입니다.

하지만 이 방법은 잘 쓰이지 않을 뿐더러, object를 사용하기 전에 freeze method를 호출할 것이라고 compiler가 확신하지 못하기 때문에 runtime 중에 에러를 발생시킬 수 있습니다.

## 3. Builder pattern

### 방법 설명

원하는 object를 직접적으로 생성하지 않습니다. 대신에,

1. client가 필요한 parameter와 함께 constructor를 호출하고 builder object를 얻습니다.
2. client가 setter와 비슷한 method를 builder object에서 관심있는 parameter를 넣어 호출합니다.
3. 마지막으로 parameter가 없는 build method를 실행하여 object를 생성합니다.

일반적으로 builder는 생성해주려는 class의 static member class로 선언해줍니다.

https://github.com/jbloch/effective-java-3e-source-code/blob/master/src/effectivejava/chapter2/item2/builder/NutritionFacts.java

### 장점

1. class가 immutable하게 됩니다.
2. 모든 parameter의 default 값을 한 곳에 모을 수 있습니다.
3. chaining을 통해 유창한 API를 만들 수 있습니다.

**client code**

```
NutritionFacts cocaCola = new
NutritionFacts.Builder(240,8).calories(100).sodium(35).carbohydrate(27).build()
;
```

4. python이나 Scala에서 볼 수 있는 named optional parameters를 연상케 합니다.
5. Validity checks가 간결하게 생략됩니다.

   곧바로 invalid parameter를 확인하기 위해 빌더로 부터 parameter를 copy한 다음에 constructor나 method에서 검증하도록 합니다.

그리고 검증이 실패한다면 throw를 통해 어떤 파라미터가 유효하지 않은지 exception을 던지도록 합니다.

## Builder pattern은 hierarchies에 적합합니다.

builders의 parallel girachy of builder를 사용하세요.

Abstract class는 abstract builder를 가질 것이며, concrete class는 concrete buidler를 가지도록 합니다.

### 예시 - Pizza

**root class**

```
public abstract class Pizza {
    public enum Topping { HAM, MUSHROOM, ONION, PEPPER, SAUSAGE }
    final Set<Topping> toppings;

    abstract static class Builder<T extends Builder<T>> {
        EnumSet<Topping> toppings = EnumSet.noneOf(Topping.class);
        public T addTopping(Topping topping) {
            toppings.add(Objects.requireNonNull(topping));
            return self();
        }

        abstract Pizza build();

        // Subclasses must override this method to return "this"
        protected abstract T self();
    }

    Pizza(Builder<?> builder) {
        toppings = builder.toppings.clone(); // See Item 50
    }
}
```

**테크닉**

1. recursive type parameter를 generic type으로 둡니다. absract self method와 함께요.

   이경슨 casting없이 subclass와 method chaining을 가능하도록 합니다.

*simulated self-type idiom*

**parallel hirachy한 상황에서도 builder패턴은 유용하게 사용될 수 있습니다.**

두 개의 concrete한 Pizza의 subclass가 있습니다.

하나는 New-York-style pizza를 나타내고, 다른 하나는 calzone style pizza를 나타냅니다. 전자는 size parameter가 필요하며 후자는 sauce를 안에 넣을 것인지 말것인지를 결정해야 합니다.

```
// Subclass with hierarchical builder (Page 15)
public class NyPizza extends Pizza {
    public enum Size { SMALL, MEDIUM, LARGE }
    private final Size size;

    public static class Builder extends Pizza.Builder<Builder> {
        private final Size size;

        public Builder(Size size) {
            this.size = Objects.requireNonNull(size);
        }

        @Override public NyPizza build() {
            return new NyPizza(this);
        }

        @Override protected Builder self() { return this; }
    }

    private NyPizza(Builder builder) {
        super(builder);
        size = builder.size;
    }

    @Override public String toString() {
        return "New York Pizza with " + toppings;
    }
}
```

```
// Subclass with hierarchical builder (Page 15)
public class Calzone extends Pizza {
    private final boolean sauceInside;

    public static class Builder extends Pizza.Builder<Builder> {
        private boolean sauceInside = false; // Default

        public Builder sauceInside() {
            sauceInside = true;
```

```
            return this;
        }

        @Override public Calzone build() {
            return new Calzone(this);
        }

        @Override protected Builder self() { return this; }
    }

    private Calzone(Builder builder) {
        super(builder);
        sauceInside = builder.sauceInside;
    }

    @Override public String toString() {
        return String.format("Calzone with %s and sauce on the %s",
                toppings, sauceInside ? "inside" : "outside");
    }
}
```

**주목할 점**

subclass의 builder의 build method는 알맞은 subclass를 반환하도록 합니다. NyPizza.Builder의 build method는 NyPizz를 반환하죠.

이렇게 sub class method에서 super class에서 정의된 return type의 subtype을 반환하도록 하는 형태를 covariant return typing이라고 합니다. 이는 casting할 필요없이 이러한 builder를 사용하도록 하죠.

다른 소소한 장점으로는 다음과 같습니다. single builder는 여러번 재사용 될 수도 있습니다.

## 단점

### 1. 생성하기 위해서는 반드시 첫번째로 builder를 생성해야만 합니다.

performance가 중요한 상황에서는 문제가 될  수 있습니다.

### 2. telescope parameter pattern보단 수다스럽습니다.(이것저것 쓸게 많다)

parameter가 적으면 telescope parameter pattern을 생각할 수 있지요.

하지만 class가 진화하면서 parameter를 더 필요로 하게 된다면, 눈에 가시가 됩니다. 따라서 시작을 builder로 하는게 좋습니다.

## 요약

parameter가 많아질 경우 builder pattern은 매우 유용한 선택이 될 수 있습니다. 특히나 optional parameter가 있을 경우엔요.

이를 적용하면 telescoping constructors보단 Client code는 읽고 쓰기 쉬워질 것이며, JavaBeans보단 더욱 안전할 수 있습니다.