생성자에 매개변수가 많다면 빌더를 고려하라

발표자료

정적 팩토리 메소드 & 생성자의 제약

영양정보	총 내용량(355 ml) 11 kcal
나트륨 275mg	14%
탄수화물 4g	1%
당류 0g	0%
에리스리톨 2g	
단백질 1g	2%
지방 0.1g	0%
트랜스지방 0g	
포화지방 0g	0%
콜레스테롤 0mg	0%
나이아신(비타민 B3) 30 mg NE	200%
판토텐산(비타민 B5) 15.0 mg	300%
비타민 B6 3.0 mg	200%
비타민 B12 9μg	375%

선택적 매개변수

선택적 매개변수 많을 때 적절히 대응 어렵다 !

선택적 매개변수 많은 클래스 구현 방법

1. 점층적 생성자 패턴
2. 자바빈즈 패턴
3. 빌더 패턴

이 중 선택적 매개변수가 많다면 빌더 패턴 고려!

방법 1. 점층적 생성자 패턴

```
public class Car{
        private final String name;
        private final int km;
        private final int cc;

        public Car (String name){~}
        public Car (String name, int km){~}
        public Car (String name, int km, int cc){~}
        public Car (int km){~};
        public Car (int km, int cc){~}
        .....
}
```

생성자를 여러 개 만드는 방법

방법 2. 자바빈즈 패턴

```
public class Car{
        private String name = "";
        private int km = 0;
        private int cc = 0;

        public Car (){~}

        public void setName (String name){~}
        public void setKm (int km){~}
        public void setCC (int cc){~}
        .....
}
```

필드마다 setter를 가지고 설정하는 방법

방법 3. 빌더 패턴

```java
public class Car{
        private final String name;
        private final int km;
        private final int cc;
        private final int year;

        //빌더 클래스 정적 멤버 클래스로 가짐
        public static class Builder{
                //필수 매개변수
                private final String name;
                private final int km;

                //선택 매개변수 - 기본 값으로 초기화
                private int cc = 0;
                private int year = 0;

                //필수 매개변수 가지는 생성자
                public Builder(String name, int km){~~}

                //선택 매개변수 setter
                public Builder cc(int cc){~}
                public Builder year(int year){~}

                //build 메소드
                public Car build(){
                        return new Car(this);
                }
        }

        //builder를 받는 private 생성자
        private Car(Builder builder){~}
}
```

점층적 생성자 패턴 안전성 + 자바빈즈 패턴 가독성 = 빌더 패턴

동작 방식

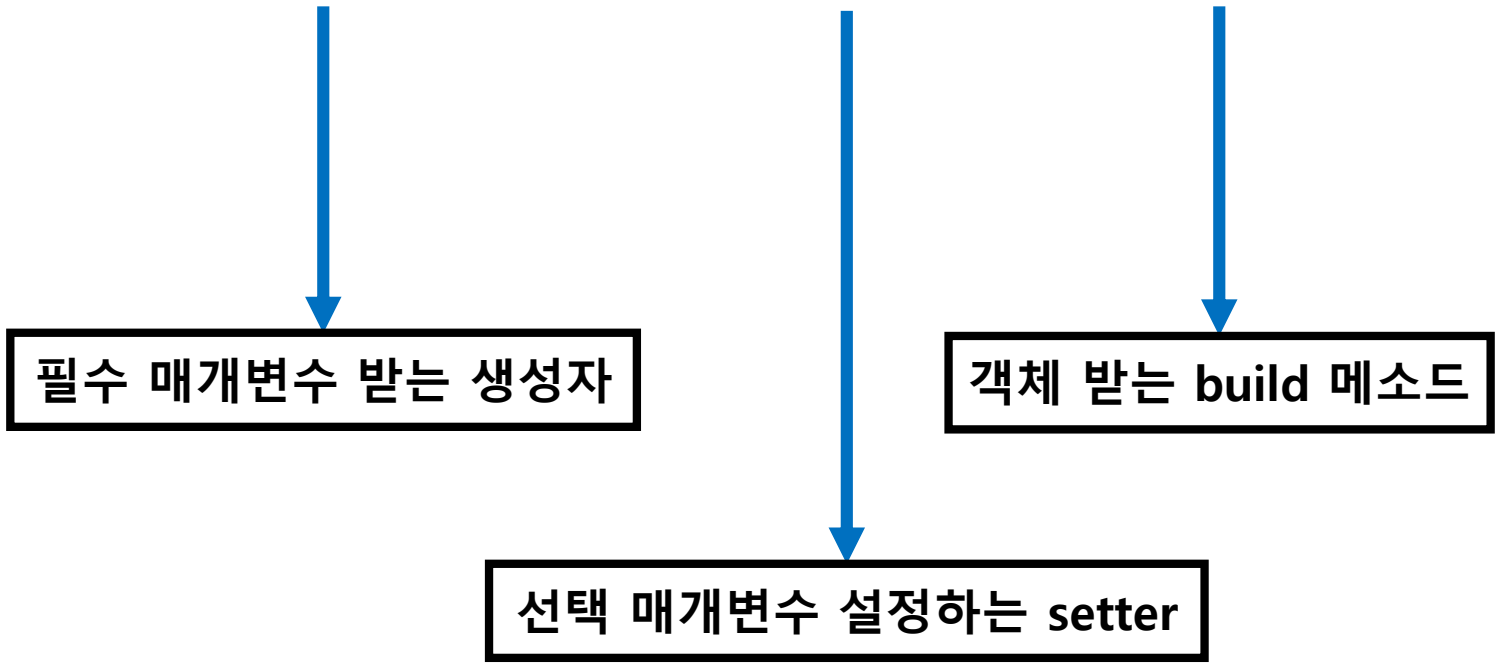
1. 필수 매개변수만으로 생성자 호출해 빌더 객체 획득
2. 빌더 객체의 setter로 선택 매개변수 설정
3. Build 메소드로 필요한 객체 획득

빌더 패턴의 장점

1. 코드를 사용하기, 읽기 쉽다.
2. 계층적으로 설계된 클래스와 함께 쓰기 좋다.
3. 상당히 유연하다.

장점 1. 코드를 사용하기 쉽고 읽기 쉽다.

```
Car sonata = new Car.Builder("YF", 200).cc(2000).year(2).build();
```



장점 2. 계층적으로 설계된 클래스와 함께 쓰기 좋다.

상위 클래스

```
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();

        protected abstract T self();
    }

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

상위 클래스는 제네릭 타입으로 선언

추상 메소드인 self 추가해 하위 클래스에서
형변환 하지 않고도 메소드 연쇄 지원

장점 2. 계층적으로 설계된 클래스와 함께 쓰기 좋다.

하위 클래스

```java
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 = size;
        }

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

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

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

하위 클래스는 build 메소드 하위 클래스 반환하도록 함

self 메소드 구현해 자기 Builder 반환하도록 함

```java
public class Calzone extends Pizza {
    private final boolean sauceInside;

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

        public Builder sauceInside() {
            this.sauceInside = true;
            return this;
        }

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

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

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

장점 2. 계층적으로 설계된 클래스와 함께 쓰기 좋다.

```
NyPizza pizza = new NyPizza.Builder(SMALL)
        .addTopping(SAUSAGE).addTopping(ONION).build();
Calzone calzone = new Calzone.Builder()
        .addTopping(HAM).sauceInside().build();
```

장점 2. 계층적으로 설계된 클래스와 함께 쓰기 좋다.

```
public T addTopping(Topping topping) {
    toppings.add(Objects.requireNonNull(topping));
    return self();
}

abstract Pizza build();

protected abstract T self();
```

장점 2. 계층적으로 설계된 클래스와 함께 쓰기 좋다.

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

장점 3. 상당히 유연하다.

1. 빌더 하나로 여러 객체 순회하면서 생성 가능
2. 매개변수에 따라 다른 객체 생성 가능
3. 특정 필드들은 빌더가 알아서 채우기 가능

빌더 패턴의 단점

1. 객체 만들려면 빌더 먼저 만들어야 함
2. 코드가 장황해 매개변수 4개 이상이어야 이득

결론

선택적 매개변수가 많고 같은 타입의 매개변수가 여러 개라면?

-> 코드 읽고 쓰기 더 편하고 안전한 빌더 패턴 선택하자!