[item 2] 생성자의 매개변수가 많을 때는 빌더(builder)를 고려하자

✏️ 생성자의 매개변수가 많아질 경우

(1) 텔리스코핑 생성자 패턴, (2) 자바빈즈 패턴, (3) 빌더 패턴을 고려 해 볼 수 있다.

📎 1. Telescoping constructor pattern

```
필수 매개변수들만 갖는 생성자
필수 매개변수들 + 1개의 선택 매개변수를 갖는 생성자
필수 매개변수들 + 2개의 선택 매개변수를 갖는 생성자
...
```

위와 같은 형태로 모든 선택 매개변수를 생성자가 가질 수 있도록 여러 개의 생성자를 겹겹이 만드는 방식

Disadvantages

매개변수가 많을 때는 클라이언트 코드 작성이 힘들고, 코드의 가독성이 떨어진다

📎 2. JavaBeans pattern

```
NutritionFacts cocaCola = new NutritionFacts();
cocaCola.setServingSize(240);
cocaCola.setServings(8);
cocaCola.setCalories(100);
cocaCola.setSodium(35);
cocaCola.setCarbohydrate(27);
```

매개변수가 없는 생성자를 호출해서 객체를 생성한 후 setter 메소드를 호출해서 각각 필수 필드와 선택 필드 값을 지정

Advantages

코드가 조금 길어지기는 하지만 인스턴스 생성이 간단하고 코드의 가독성이 좋다.

Disadvantages

1. 여러번의 메소드(constructor, setters methods) 호출로 나누어져 인스턴스가 생성되므로, 생성 과정을 거치는 동안 자바빈 객체가 일관된 상태를 유지하지 못할 수 있다.

   객체 생성이 완전하게 끝났을 때 그 객체를 동결하고 완전하게 되기 전까지는 사용할 수 없도록 함으로써 그런 단점을 줄일 수 있다 → 적용이 어려워 거의 사용되지 않는다.

2. 불변 클래스를 만들 수 있는 가능성을 배제하므로 thread 에서 안전성을 유지하려면 프로그래머의 추가적인 노력이 필요하다

📎 3. Builder pattern

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

Advantages

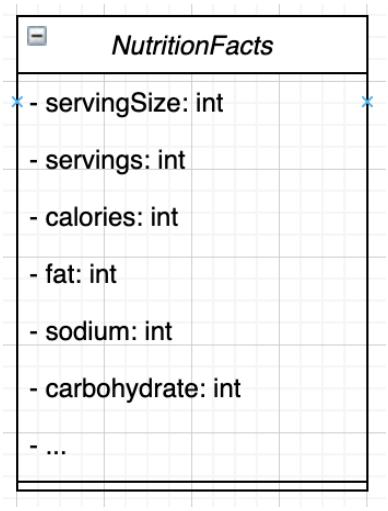
1. 작성이 쉽고 가독성이 좋다
2. 빌더에 매개변수에 불변 규칙을 적용할 수 있고, build 메소드는 그런 불변 규칙을 검사할 수 있다 (→항목39)
3. 여러 개의 가변인자 매개변수를 가질 수 있다
4. 유연하다
5. 매개변수들의 값이 설정된 빌더는 훌륭한 추상 팩토리를 만든다

Disadvantages

1. 성능이 매우 중요한 상황에서는 문제가 될 수 있다.
2. 텔리스코핑 패턴보다 코드가 길어질 수 있다.

생성자의 매개변수가 많아질 경우

어떤 클래스의 필드값이 엄청 많은 경우 생성자를 어떻게 만들어야 하는가?



포장 식품에 붙은 영양 정보 라벨을 나타내는 클래스의 경우, 식품의 양, 개수, 칼로리 - 여기까지는 필수 필드- 총 지방, 포화 지방, 트랜스 지방, 콜레스테롤, 나트륨 등 20개 이상의 선택 필드를 가질 수 있다. 대부분의 식품은 선택 필드 중 몇 개만 값을 가진다고 할 때 생성자를 어떻게 만들어야 할까?

Telescoping constructor pattern

필요한 모든 케이스에 대해서 생성자를 만들어내자!

```
public class NutritionFacts {
    private final int servingSize;  // (mL)            required
    private final int servings;     // (per container) required
    private final int calories;     // (per serving)   optional
    private final int fat;          // (g/serving)     optional
    private final int sodium;       // (mg/serving)    optional
    private final int carbohydrate; // (g/serving)     optional

    public NutritionFacts(int servingSize, int servings) {
        this(servingSize, servings, 0);
    }

    public NutritionFacts(int servingSize, int servings,
                          int calories) {
        this(servingSize, servings, calories, 0);
    }

    public NutritionFacts(int servingSize, int servings,
                          int calories, int fat) {
```

```
        this(servingSize, servings, calories, fat, 0);
    }

    public NutritionFacts(int servingSize, int servings,
                          int calories, int fat, int sodium) {
        this(servingSize, servings, calories, fat, sodium, 0);
    }

    public NutritionFacts(int servingSize, int servings,
                          int calories, int fat, int sodium, int carbohydrate) {
        this.servingSize  = servingSize;
        this.servings     = servings;
        this.calories     = calories;
        this.fat          = fat;
        this.sodium       = sodium;
        this.carbohydrate = carbohydrate;
    }
}
```

```
NutritionFacts cocaCola = new NutritionFacts(240, 8, 100, 0, 35, 27);
```

위와 같이 생성자를 만드는 방식을 텔레스코핑 패턴이라고 한다. 오버로딩을 이용한 방식이며, 보통 원하지 않는 매개변수에도 초기값을 주어야 한다

[오버로딩] 메소드 매개변수들의 개수와 타입 또는 순서를 달리하면 하나의 클래스에 같은 이름의 메소드를 여러 개 정의할 수 있다. 이를 오버로딩이라고 하며 생성자도 오버로딩이 가능하다

Telescoping constructor pattern 의 단점

매개변수가 많아질수록 코드 작성이 힘들고 - 귀찮게 뻔한 코드를 많이 작성 - 코드의 가독성도 떨어진다.

```
NutritionFacts cocaCola = new NutritionFacts(240, 8, 100, 0, 35, 27);
```

해당 코드를 처음 본 사람 입장에서 이 코드를 딱 봤을 때 240 이 무슨 값인지, 8이 무슨 값인지 모르겠고

해당 코드를 작성하는 입장에서는 내가 몇번째 파라매터를 입력하고 있는지 주의깊게 살펴보면서 작성해야 한다.

만약 어떤 두 파라매터의 순서를 바꿔 입력하면 컴파일 에러는 생기지 않겠지만 프로그램의 실행이 엉뚱해질 것

JavaBeans pattern

```
NutritionFacts cocaCola = new NutritionFacts();
cocaCola.setServingSize(240);
cocaCola.setServings(8);
cocaCola.setCalories(100);
cocaCola.setSodium(35);
cocaCola.setCarbohydrate(27);
```

위와 같이 기본 생성자를 이용해서 객체를 하나 찍어 내고 setter 를 이용해서 필드 값들을 채우는 방식

> 💡 **[getter, setter method]** 객체의 캡슐화와 정보은닉을 위해 인스턴스 변수는 private 으로 지정하고, 각 인스턴스 변수의 값은 public 메소드를 통해 다른 객체가 읽거나 변경하도록 한다. 이 때 값을 읽어주는 메소드를 getter, 값을 변경하는 메소드를 setter 메소드라고 한다.

```
// JavaBeans Pattern - allows inconsistency, mandates mutability  (pages 11-12)
public class NutritionFacts {
    // Parameters initialized to default values (if any)
    private int servingSize  = -1; // Required; no default value
    private int servings     = -1; // Required; no default value
    private int calories     = 0;
    private int fat          = 0;
    private int sodium       = 0;
    private int carbohydrate = 0;

    public NutritionFacts() { }

    // Setters
    public void setServingSize(int val)  { servingSize = val; }
    public void setServings(int val)     { servings = val; }
    public void setCalories(int val)     { calories = val; }
    public void setFat(int val)          { fat = val; }
```

```
    public void setSodium(int val)       { sodium = val; }
    public void setCarbohydrate(int val) { carbohydrate = val; }
}
```

필수 필드는 -1 로, 선택 필드는 0 으로 값을 주고, 기본 생성자와 setter 들을 정의한다.

JavaBeans pattern 의 단점

허나 이 방식은 여러 줄에 나눠서 메소드를 호출하며 인스턴스를 생성하게 되기 때문에, 생성 과정을 거치는 동안 자바빈 객체가 일관된 상태를 유지하지 못할 수 있다

객체 생성이 완전히 끝났을 때 그 객체를 동결하고 완전하게 되기 전까지는 사용할 수 없도록 함으로써 그런 단점을 줄일 수 있다고는 하는데 적용이 어려워서 거의 사용되지 않는다고 한다.

불변 클래스를 만들 수 있는 가능성을 배제하므로 thread 에서 안전성을 유지하려면 프로그래머의 추가적인 노력이 필요하다

→ 생성중에 객체의 상태가 변할 수 있기 때문에 생성 중에 쓰레드 1에서 해당 객체를 가져가서 사용하고 나머지 생성 메소드를 호출 한 후 쓰레드 2에서 가져가 사용하면 두 객체가 다를 수 있다?고 이해했음

객체의 필드값들이 언제 어떻게 바껴있을지 보장 할 수가 없다!

> 💡 thread-safety or thread-safe code in Java refers to code which can safely be used or shared in concurrent or multi-threading environment and they will behave as expected. any code, class, or object which can behave differently from its contract on the concurrent environment is not thread-safe.

Builder pattern

텔레스코핑 생성자 패턴의 **안전성과** 자바빈즈 패턴의 **가독성**을 결합한 세번째 방법

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

1. 빌더 객체 얻기

   빌더 객체는 모든 필수 매개변수를 설정함

   생성자나 static 팩토리 메소드 사용 가능

   (여기서는 NutritionFacts.Builder 의 생성자를 호출하여 빌더 객체를 얻음)
2. 빌더 객체의 세터 메소드를 호출

   (calories, sodium, carbohydrate 세터들을 호출)
3. build() 를 호출하여 불변 객체 생성

```
public class NutritionFacts {
    private final int servingSize;
    private final int servings;
    private final int calories;
    private final int fat;
    private final int sodium;
    private final int carbohydrate;

    public static class Builder {
        // Required parameters
        private final int servingSize;
        private final int servings;

        // Optional parameters - initialized to default values
        private int calories = 0;
        private int fat = 0;
        private int sodium = 0;
        private int carbohydrate = 0;

        public Builder(int servingSize, int servings) {
            this.servingSize = servingSize;
            this.servings = servings;
        }

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

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

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

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

        public NutritionFacts build() {
            return new NutritionFacts(this);
        }
    }

    private NutritionFacts(Builder builder) {
        servingSize = builder.servingSize;
        servings = builder.servings;
        calories = builder.calories;
        fat = builder.fat;
        sodium = builder.sodium;
        carbohydrate = builder.carbohydrate;
    }
}
```

Builder pattern 장점

- 작성이 쉽고 가독성이 좋다

  롬복을 이용하면 더 쉬워진다!

  ```
  @Builder
  public class NutritionFactsLombok {
      private final int servingSize;
      private final int servings;
      private final int calories;
      private final int fat;
      private final int sodium;
      private final int carbohydrate;

      public static void main(String[] args) {
          NutritionFactsLombok cocaCola = NutritionFactsLombok.builder()
                  .servingSize(240).servings(8)
                  .calories(100).sodium(35).carbohydrate(27).build();
      }
  }
  ```

- 빌더에 매개변수에 불변 규칙을 적용할 수 있고, build 메소드는 그런 불변 규칙을 검사할 수 있다 (→항목39)

여기서 말하는 불변규칙이란 매개변수나 필드가 가질 수 있는 값의 약속된 범위나 타입 및 형태이다.

이런 불변규칙에 맞지 않는 경우 build 메소드에서 IllegalStateException 예외를 발생시켜 검사할 수 있다.

→ 실제로 이런 식으로 써본 적이 없어서 자세한 설명은 생략하고 넘어가겠습니다 😐 .. 누군가 좀 알려주시길 ..

Builder pattern 단점

1. 어떤 객체를 생성하기 위해 빌더를 만들어야 생성할 수 있기 때문에 성능이 매우 중요한 상황에서는 문제가 될 수 있다.
2. 빌더 패턴은 텔리스코핑 패턴보다 코드가 길어진다.

   → 매개변수가 4개 이상쯤 됐을 때 사용하는 것이 좋다

   → 그러나 코드는 언제든 수정 될 수 있는 것이고 언제 매개변수가 추가 될지 모른다

   → 여러번 수정을 통해서 매개변수가 굉장히 많아졌을 때 원래 있던 생성자나 static 팩토리 메소드를 지우려고 하면 아까울 것이다

   → 그러니 매개변수 개수가 적더라도 처음부터 빌더 패턴를 염두는 해라 (..?)

적절한 예제코드

Member 클래스는 이름, 나이 필드가 반드시 필요하고 생일, 취미, mbti 필드는 옵셔널한 필드값라고 할 때 Member 를 빌더 패턴으로 만들어보자! 🥁🎸🥁🎸🥁🎸

▼ 시작

```
public class Member {
    // required
    private String name;
    private Integer age;

    // optional
    private LocalDate birth;
    private String hobby;
    private String mbti;
}
```

▼ 컨닝 페이퍼

```
import java.time.LocalDate;

public class Member {
    // required
    private String name;
    private Integer age;

    // optional
    private LocalDate birth;
    private String hobby;
    private String mbti;

    public static void main(String[] args) {
        Member hjeong = new Member.Builder("유효정", 28)
                .birth(LocalDate.now()).hobby("눕기").mbti("INTP")
                .build();
    }

    static public class Builder {
        // required
        private String name;
        private Integer age;

        // optional
        private LocalDate birth;
        private String hobby;
        private String mbti;

        public Builder(String name, int age) {
            this.name = name;
            this.age = age;
        }

        public Builder birth(LocalDate birth) {
            this.birth = birth;
            return this;
        }

        public Builder hobby(String hobby) {
            this.hobby = hobby;
            return this;

        }

        public Builder mbti(String mbti) {
            this.mbti = mbti;
            return this;
        }

        public Member build() {
            return new Member(this);
        }
```

```
    }

    public Member(Member.Builder builder) {
        this.age = builder.age;
        this.birth = builder.birth;
        this.hobby = builder.hobby;
        this.mbti  = builder.mbti;
        this.name = builder.name;
    }
}
```