다양한 객체 생성 방법 (2/13)

- 참고: Effective Java 3rd Edition, 2장. 객체 생성과 파괴
- 객체 생성 방법: 생성자, 정적 팩토리 메서드, 빌더 등
  - 기본 생성자

```
public class OlympicPlayer {
    private int age;
    private String name;

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

    public void addAge(int adder) {
        this.age += adder;
    }
}
```

```
OlympicPlayer olympicPlayer = new OlympicPlayer("Minjeong", 23);

//동일 변수를 가진 Person를 통해 OlympicPlayer로 변환시 경우 캡슐화 위반
Person person = new Person("Daeheon", 22);
OlympicPlayer olympicPlayer = new OlympicPlayer(person.getName(), person.getAge());
```

- 주생성자/부생성자 추가로 캡슐화 위반 문제 해결

```
public class OlympicPlayer {
    private int age;
    private String name;

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

    public OlympicPlayer(Person person) {
        this(person.getName(), person.getAge());
    }
  }
```

  - score 추가시, 주생성자와 부생성자 활용하는 this를 이용해 코드 중복 피할 수도 있음.

```
public OlympicPlayer(String name, int age, int score) {
        this(name, age);
        this.score = score;
    }
```

  - 생성자 추가시 객체 사용함에 있어 유연성과 확장성 커짐. 하지만, score에 null 값인 경우 NullPointerException 발생할 수도 있음.

Item 1: 생성자 대신 static 팩토리 메소드를 고려하라

- **정적 팩터리 메서드 (static factory method)**: 클래스의 인스턴스를 반환하는 단순한 정적 메서드

```
//p.8
public static Boolean valueOf(boolean b) {//boolean 기본 타입의 박싱 클래스인 Boolean
        return b ? Boolean.TRUE : Boolean.FALSE;
    }
```

- 참고 : private static method가 필요한 이유? 호출되는 scope 때문에 private static한 것. (밖으로 노출 될 필요 없음)

장점 #1. 이름을 가질 수 있다.

```
public static OlympicPlayer withScore(String name, int age, int score) {
        return new OlympicPlayer(name, age, score);
    }
```

- 생성자: BigInteger(int, int Random) vs 정적 팩터리 매소드: BigInteger.probablePrime

  → 값이 소수(Prime Number)인 BigInteger를 반환한다.

- 하나의 시그니처로 생성자 하나만 만들 수 있으나 정적 팩터리 메서드는 제약 없음.

  → 한 클래스에 시그니처 같은 생성자 여러 개 필요한 경우 정적 팩터리 메서드 사용 추천

장점 #2. 호출될 때마다 인스턴스를 새로 생성하지 않아도 됨.

- 불변 클래스는 인스턴스 미리 만들어 놓거나 생성한 인스턴스를 캐싱하여 재활용하는 식으로 불필요한 객체 생성 피할 수 있음.
- Boolean.valueOf(boolean)는 아예 객체 생성 하지 않음. Flyweight pattern도 비슷한 방법.
  - valueOf: from과 of의 더 자세한 버전
  - BigInteger prime = BigInteger.valueOf(Integer.MAX_VALUE);

- 인스턴스 통제 클래스. 인스턴스 통제 이유?
  - 1. 싱글톤으로도 생성 가능 (item 3 참고)

    ```
    ApplicationContext applicationContext = new AnnotationConfigApplicationContext(Config.class);
            UserService userService1 = applicationContext.getBean(UserService.class);
    ```

  - 2. 인스턴스화 불가로 만들 수 있음
  - 3. 불변 값 클래스에서 동치인 인스턴스가 단 하나뿐임 보장. (a == b일 때만 a.equals(b) 성립)
  - 4. 플라이웨이트 패턴의 근간이 됨. 열거 타입은 인스턴스가 하나만 만들어짐 보장.

- Flyweight Pattern: 클래스의 인스턴스 한 개만 가지고 여러 개의 가상 인스턴스 제공하고 싶을 때 사용하는 패턴. 인스턴스 공유 최대화로 new 연산자 통한 메모리 낭비 줄임.

장점 #3. 반환 타입의 하위 타입 객체 반환할 수 있는 능력 있음.

- API 만들 때 유연성 응용하면 구현 클래스 공개하지 않고 그 객체 반환 가능. API 작게 유지.

  인터페이스 기반 프레임워크 (return 타입에는 인터페이스만 노출하고 실제 인터페이스의 구현체 리턴 (Client에게 구현제 노출 안시킨다는 장점)

- java.util.Collections 자바 8 부터 인터페이스 public static 메소드 추가, 자바 9부터 private static 메소드 추가 (Collections 추가없이 interface 구현하면 됨)

장점 #4. 입력 매개변수에 따라 매번 다른 클래스의 객체 반환할 수 있음.

- EnumSet 클래스. RegularEnumSet (long 변수 하나 관리), JumboEnumSet (long 배열 관리)
- Client 클래스 존재 모름.

```
enum Color {
        RED, BLUE, WHITE

    }

EnumSet<Color> colors = EnumSet.allOf(Color.class);

EnumSet<Color> blueAndWhite = EnumSet.of(BLUE, WHITE);
```

장점 #5. 정적 팩터리 메서드를 작성하는 시점에는 반환할 객체의 클래스가 존재하지 않아도 됨.

- 서비스 제공자 프레임워크 JDBC (Java Database Connectivity): provider는 서비스 구현체
  - 핵심 컴포넌트 3개로 이뤄짐. 1) 서비스 인터페이스, 2) 제공자 등록 API, 3) 서비스 접근 API

  → 서비스 접근 API, 유연한 정적 팩터리의 실체

  ```
  DriverManager.registerDriver//제공자 등록 API 역할
  DriverManager.getConnnection()... //서비스 접근 API 역할
  ```

- DI도 강력한 서비스 제공자, Java 6, ServiceLoader 범용 서비스 제공자 프레임워크.

단점 #1. 상속 하려면 public이나 protected 생성자 필요. 정적 팩터리 메서드만 제공하면 하위 클래스 만들 수 없음.

- Collection Framework의 유틸리티 구현 클래스들 상속 불가.
- 상속보다 컴포지션 사용하도록 유도하고 불변 타입 만들려면 지켜야 하는 점. → 단점이자 장점

단점 #2. 정적 팩터리 메서드는 프로그래머가 찾기 어려움.

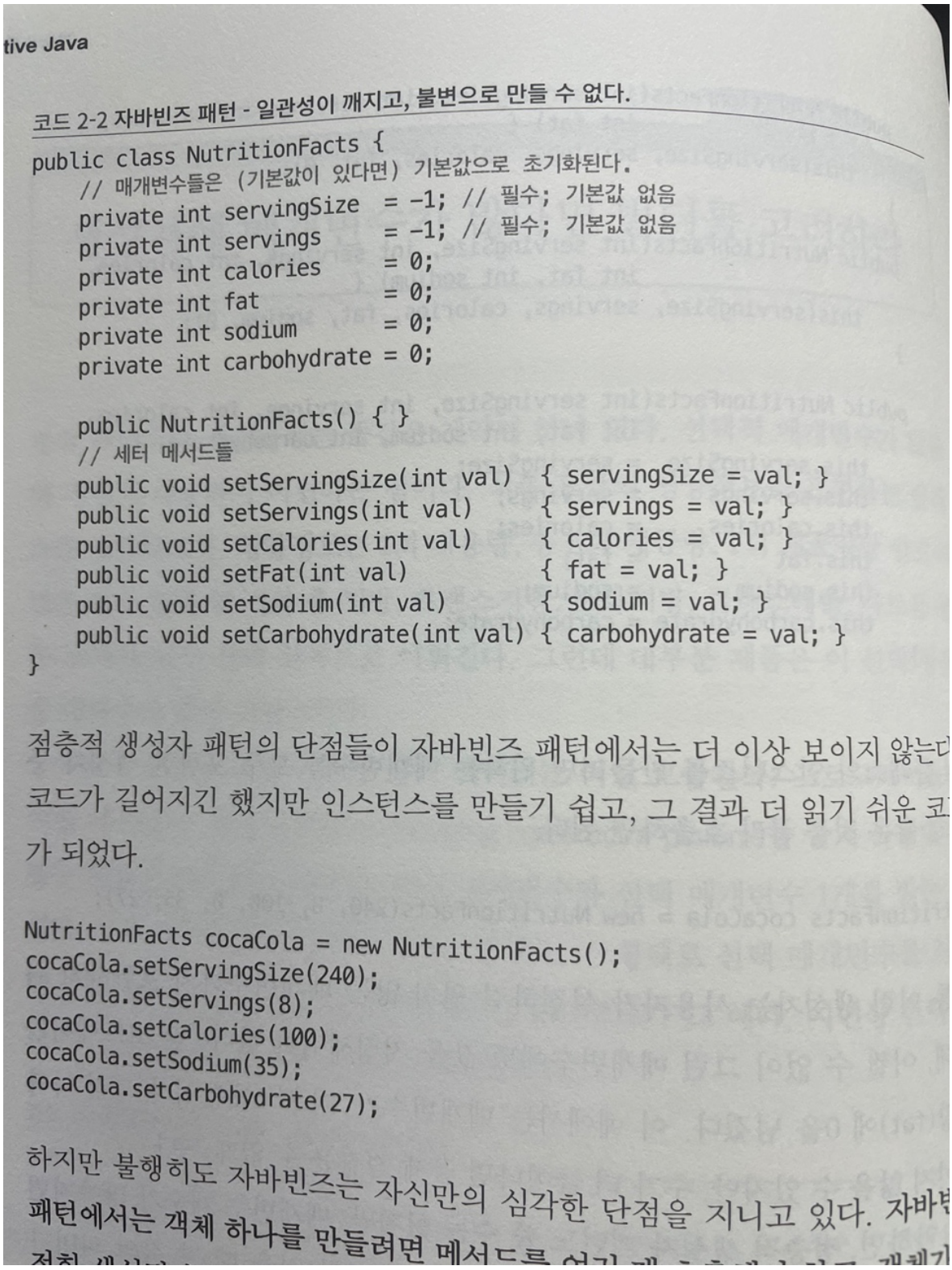
- 명확히 드러나지 않아 User는 정적 팩터리 메서드 방식 클래스 인스터스화할 방법 알아야함.
- 정적 메소드는 생성자와 달리 API 문서에서 따로 다루지 않으므로 클래스나 인터페이스 상단에 문서 제공.

**정적 팩터리 메서드에 흔히 사용하는 명명 방식

- from: 매개변수 하나 받아서 해당 타입의 인스턴스 반환하는 형변환 메서드
  - Date d = Date.from(instant);
- of: 여러 매개 변수 받아 적합한 타입의 인스턴 반환하는 집계 메서드
  - Set<Rank> faceCards = EnumSet.of(JACK, QUEEN, KING);
- valueOf: from과 of의 더 자세한 버전
  - BigInteger prime = BigInteger.valueOf(Integer.MAX_VALUE);
- instance 혹은 getInstance: 같은 인스턴스임 보장은 하지 않음.
  - StackWalker luke = StackWalker.getInstance(options);
- create 혹은 newInstance: 매번 새로운 인스턴스 생성 반환 보장
  - Object newArray = Array.new Instance(classObject, arrayLen);
- getType: getInstance와 같으나, 생성할 클래스가 아닌 다른 클래스에 팩터리 메서드 정의시 사용. 반환객체 type
  - FileStore fs = Files.getFileStore(path)
- newType: newInstance와 같으나, 생성할 클래스가 아닌 다른 클래스에 팩터리 메서드 정의시 반환 객체 type
  - BufferedReader br = Files.newBufferedReader(path);
- type: getType과 new Type 간결한 버전
  - List<Complaint> litany = Collections.list(legacyLitany);

Item 2 생성자에 매개 변수가 많다면 빌더를 고려하라

- 점층적 생성자 패턴 - 확장 어려움
- 자바빈즈 점층적 생성자보다 더 읽기 쉬우나 객체 하나 위해 여러 개 호출, 객체 완전히 생성되기 전까지 비일관성.



tive Java

코드 2-2 자바빈즈 패턴 - 일관성이 깨지고, 불변으로 만들 수 없다.

```
public class NutritionFacts {
    // 매개변수들은 (기본값이 있다면) 기본값으로 초기화된다.
    private int servingSize  = -1; // 필수; 기본값 없음
    private int servings     = -1; // 필수; 기본값 없음
    private int calories     = 0;
    private int fat          = 0;
    private int sodium       = 0;
    private int carbohydrate = 0;

    public NutritionFacts() { }
    // 세터 메서드들
    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; }
}
```

점층적 생성자 패턴의 단점들이 자바빈즈 패턴에서는 더 이상 보이지 않는다
코드가 길어지긴 했지만 인스턴스를 만들기 쉽고, 그 결과 더 읽기 쉬운 코
가 되었다.

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

하지만 불행히도 자바빈즈는 자신만의 심각한 단점을 지니고 있다. 자바
패턴에서는 객체 하나를 만들려면 메서드를 여러

```
//빌더는 점층적 생성자보다 클라이언트 코드를 읽고 쓰기가 훨씬 간결, 자바빈즈보다 훨씬 안전함.
@Builder
```

```
public class NutritionFacts {

    @Builder.Default private int servingSize =10;
    private int sodium;
    private int carbonhydrate;
    private int servings;
    @Singular private List<String> names;

/*
//점층적 패턴
public NutritionFacts(int servingSize, int sodium...)
{
this.servingSzie =servingSize;
this.sodium = sodium;....
}
*/

/*
    public NutritionFacts(int servingSize, int sodium, int carbonhydrate, int servings) {
        this.servingSize = servingSize;
        this.sodium = sodium;
        this.carbonhydrate = carbonhydrate;
        this.servings = servings;
    }
*/
    public static void main(String[] args) {
        //NutritionFacts nutritionFacts = new NutritionFacts(1, 6, 10, 20); //zmffotm dlstmxjstmghk
        NutritionFacts nutritionFacts = NutritionFacts.builder()
                .servings(10)
                .carbonhydrate(100)
                .name("Hello")
                .name("World")
                .clearNames()
               // .servingSize(1)
                .build();

       // NutritionFacts facts = new NutritionFacts();

    }
}
```

- 생성자나 정적 팩터리가 처리해야 할 매개변수가 많다면 빌더 패턴 선택하는게 더 나음.
    - 매개변수 4개 이상 되어야 값어치 함.

References:

- 자바 객체 생성 https://tecoble.techcourse.co.kr/post/2021-05-17-constructor/
- Flyweight Patter https://lee1535.tistory.com/106
- 백기선님 요약본 https://github.com/keesun/study/blob/master/effective-java/item1.md