

---

# 이펙티브 자바

1장 ~ 2장(~item 2)

24/01/03

---

# 1장 : 들어가기

# 01장 들어가기

---

이 책은 총 90개의 아이템과, 아이템들을 주제별로 묶은 11개의 장으로 구성된다.

또한 많은 디자인 패턴과, 피해야할 안티패턴들을 소개한다.

앞으로의 자바8용 언어 명세를 따른다.  
자바가 지원하는 타입은 다음 네가지이다.

- 인터페이스 -참조 타입
- 클래스 - 참조 타입
- 배열 (array) - 참조 타입
- 기본타입 (primitive)

즉, 클래스의 인스턴스와 배열은 객체이지만, 기본 타입은 그렇지 않다.

# 01장 들어가기

---

## 클래스의 멤버 종류?

- 필드
- 메서드
- 멤버 클래스
- 멤버 인터페이스

## API 요소

클래스, 인터페이스, 패키지를 통해 접근 가능한 모든

- 클래스
- 인터페이스
- 생성자
- 멤버
- 직렬화된 상태



API의 사용자 : API를 사용하는 프로그램 작성자(사람)  
API의 클라이언트: API를 사용하는 클래스(코드)

---

# 2장 : 객체 생성과 파괴

이번 장에서는 다음 내용들을 다룬다.

- 객체의 생성과 파괴
- 객체를 만들어야 할 때와 만들지 말아야 할 때를 구분하는 법
- 올바른 객체 생성 방법과 불필요한 생성을 피하는 법
- 객체를 관리하는 요령

## 02장 아이템 1 : 생성자 대신 정적 팩터리 메서드를 고려하라

---

클래스의 인스턴스를 얻는 방법 두가지

public 생성자를 사용하기.  
정적 팩터리 메서드를 사용하기.

정적 팩터리 메서드의 장점 다섯가지



- 이름을 가질 수 있다.
- 호출될 때마다 인스턴스를 새로 생성하지 않아도 된다.
- 리턴 타입의 하위 타입 객체를 반환할 수 있는 능력이 있다.
- 입력 매개변수에 따라 매번 다른 클래스의 객체를 반환할 수 있다.
- 정적 팩터리 메서드를 작성하는 시점에 반환할 객체의 클래스가 존재하지 않아도 된다.

## 02장 아이템 1 : 생성자 대신 정적 팩터리 메서드를 고려하라

---

첫 번째 장점 : 이름을 가질 수 있다.

한 클래스에 시그니처가 같은 생성자가 여러개 필요할 것 같을 때,  
생성자를 정적 팩터리 메서드로 바꾸고 각각의 차이를 드러내는 이름으로 지어주자.

★ 여기서 시그니처란, 메서드의 리턴값과 파라미터의 개수/종류이다.

# 02장 아이템 1 : 생성자 대신 정적 팩터리 메서드를 고려하라

---

```
<java />

public class Car {

    private String color;
    private int power;

    private Car(String color, int power) {
        this.color= color;
        this.power = power;
    }

    public static Car createRedCar(int power) {
        return new Car("red", power);
    }

    public static Car createBlueCar(int power) {
        return new Car("blue", power);
    }
}

/**
** 외부에서
** Car car = new Car("red", 3); 이렇게 생성자를 직접 호출하는 것보다
** Car car = Car.createRedCar(3); 이렇게 호출하는 방식이 훨씬 의미를 파악하기 쉽다.
**/
```



## 02장 아이템 1 : 생성자 대신 정적 팩터리 메서드를 고려하라

---

두 번째 장점 : 호출될 때마다 인스턴스를 새로 생성하지 않아도 된다.

호출될 때마다 인스턴스를 새로 생성하지 않는다는 말은, 즉  
인스턴스를 미리 만들어 놓거나 생성한 인스턴스를 계속해서 재활용하여  
불필요한 객체 생성을 피할 수 있다는 것이다.

특히, 생성 비용이 큰 같은 객체가 자주 요청되는 상황에 성능을 더 끌어올린다.

 **NOTE** flyweight pattern도 이와 비슷한 기법이다.

## 02장 아이템 1 : 생성자 대신 정적 팩터리 메서드를 고려하라

---

정적 팩토리 메서드는 언제 어느 인스턴스를 살아 있게 할지를 통제할 수 있다.  
인스턴스를 통제하면 다음 장점들이 있다.

- 클래스를 싱글턴으로 만들 수 있다. (아이템3)
- 클래스를 인스턴스화 불가로 만들 수 있다. (아이템4)
- 불변 값 클래스에서 동치인 인스턴스가 단 하나뿐임을 보장할 수 있다.(아이템17)

# 02장 아이템 1 : 생성자 대신 정적 팩터리 메서드를 고려하라

---

```
<java />

/**
 * 불필요한 객체 생성을 방지한다.
 * 불변 클래스immutable class의 경우,
 * 인스턴스를 미리 만들어 두거나,
 * 새로 생성한 인스턴스를 캐싱하여 재사용하는 식으로 불필요한 객체 생성을 피할 수 있다.
 */

public class Robot {
    private String name;

    private static final Robot alphaRobot = new Robot("alpha");
    private static final Robot betaRobot = new Robot("beta");

    public Robot(String name) {
        this.name = name;
    }

    public static Robot getInstanceAlphaRobot() {
        return alphaRobot;
    }

    public static Robot getInstanceBetaRobot() {
        return betaRobot;
    }
}

Robot alphaRobot = Robot.getInstanceAlphaRobot();
```

## 02장 아이템 1 : 생성자 대신 정적 팩터리 메서드를 고려하라

---

세 번째 장점 : 리턴 타입의 하위 타입 객체를 반환할 수 있는 능력이 있다.

반환 타입의 하위 타입 객체를 반환한다는 것은, 엄청난 유연성을 제공한다.

## 02장 아이템 1 : 생성자 대신 정적 팩터리 메서드를 고려하라

---

```
<java />

public List<String> foo1() {
    return new ArrayList<String>();
}

public List<String> foo2() {
    return new Vector<String>();
}
```

List의 하위 클래스인 ArrayList와 Vector로 선택하여 반환하는 모습. (Stack과 LinkedList도 마찬가지로 가능)

클라이언트는 내부 구조는 모른채, List<String>으로 반환 받는다.

## 02장 아이템 1 : 생성자 대신 정적 팩터리 메서드를 고려하라

---

네 번째 장점 : 입력 매개변수에 따라 매번 다른 클래스의 객체를 반환할 수 있다.

우리가 EnumSet 클래스를 사용할 때,  
사실 EnumSet의 정적 팩터리 메서드는 원소 개수의 따라  
각기 다른 클래스의 인스턴스를 반환한다.

(원소가 64개 이하면 RegularEnumSet, 65개 이상이면 JumboEnumSet)

그러나 클라이언트는 두 클래스의 존재를 모르고 사용해도 괜찮다.  
EnumSet의 하위 클래스이기만 하면 상관없기 때문이다.


## 02장 아이템 1 : 생성자 대신 정적 팩터리 메서드를 고려하라

---

다섯 번째 장점 : 정적 팩터리 메서드를 작성하는 시점에 반환할 객체의 클래스가 존재하지 않아도 된다.

서비스 제공자 프레임워크를 만드는 근간으로,  
서비스 제공자 프레임워크는 다음 3개의 핵심 컴포넌트로 이루어져있다.

- 서비스 인터페이스
- 제공자 등록 API
- 서비스 접근 API

 조금 더 공부하고 다시 복기하기

## 02장 아이템 1 : 생성자 대신 정적 팩터리 메서드를 고려하라

---

첫 번째 단점 : 정적 팩터리 메서드만 제공하면 하위 클래스를 만들 수 없다.

상속을 하려면 public 혹은 protected 생성자가 필요하므로,  
정적 팩터리 메서드만 제공하면 하위 클래스를 만들 수 없다.

그러나 이 제약은 상속 대신 컴포지션을 사용하도록 유도하는 것일 수도 있다. (발상의 전환?)

\*상위 클래스와 하위 클래스의 관계가 is a 라면 상속을, has a 라면 컴포지션을 사용하자.



두 번째 단점 : 정적 팩터리 메서드는 프로그래머가 찾기 어렵다.

정적 팩터리 메서드 방식 클래스를 인스턴스화할 방법을 프로그래머가 알아놓아야한다.

생성자처럼 자바독을 사용한 API 설명에 명확히 들어나지 않기 때문.

API 문서를 잘 써두는 방식으로 문제를 완화할 수 있다.



# 02장 아이템 1 : 생성자 대신 정적 팩터리 메서드를 고려하라

---

## 정적 팩터리 메서드의 흔한 명명 방식들

- from : 매개변수를 하나 받아서 해당 타입의 인스턴스를 반환하는 형변환 메서드
  - `Data d = Date.from(instant);`
- of : 여러 매개변수를 받아 적합한 타입의 인스턴스를 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 / getInstance : ( 매개변수를 받는다면 ) 매개변수로 명시한 인스턴스를 반환하지만, 같은 인스턴스임을 보장하지는 않는다.
  - `StackWalker luke = StackWalker.getInstance(options);`
- create 혹은 newInstance : instance 혹은 getInstance와 같지만, 매번 새로운 인스턴스를 생성해 반환함을 보장한다.
  - `Object newArray = Array.newInstance(classObject, arrayLen);`
- getType : getInstance와 같으나, 생성할 클래스가 아닌 다른 클래스에 팩터리 메서드를 정의할 때 쓴다. "Type"은 팩터리 메서드가 반환할 객체의 타입이다.
  - `FileStore fs = Files.getFileStore(path)`

## 02장 아이템 2 : 생성자에 매개변수가 많다면 빌더를 고려하라.

---

정적 팩토리 메서드와 생성자의 공통된 고민  
➡ 선택적 매개변수가 많을 때 적절히 대응하기 어렵다!

이에 대해 발전해 온 해결책  
점층적 생성자 패턴 ➡ 자바 빈즈 패턴 ➡ 빌더 패턴

## 02장 아이템 2 : 생성자에 매개변수가 많다면 빌더를 고려하라.

---

점층적 생성자 패턴 : 원하는 생성자를 골라쓰는 방식

➡ 원하는 매개변수만 골라 쓸 수 있지만, 보통 원치 않는 매개변수도 포함하기 쉽다.  
더불어 매개변수 개수가 많아지면 클린코드 x

자바 빈즈 패턴 : 빈 생성자를 만들고, setter로 매개변수 값을 채워나가는 방식

➡ 메서드를 여러개 호출해야 하고,  
객체가 완성되기 전 일관성이 무너진다.

빌더 패턴: 빌더의 세터 메서드들이 빌더 자신을 반환하기 때문에 메서드 호출이 물 흐르듯 연결된다.

➡ 읽고 쓰기가 쉽다.

# 02장 아이템 2 : 생성자에 매개변수가 많다면 빌더를 고려하라.

---

점층적인 생성자 패턴

```
<java />
Person person = new Person("탱", 29, "010-1234-1234", "hello@gmail.com");
```

자바빈 패턴

```
<java />
Person person = new Person();
person.setName("탱");
person.setAge(29);
person.setPhoneNumber("010-1234-1234");
person.setEmail("hello@gmail.com");
```

빌더 패턴을 사용하면 점층적인 생성자 패턴의 안정성과 자바빈 패턴의 가독성을 함께할 수 있다.

```
Person person = new Person().Builder("탱", 29).phoneNumber("010-1234-1234").email("hello@gmail.com").build();
```

```
<java />
Person person = new Person().Builder("탱", 29)
    .phoneNumber("010-1234-1234")
    .email("hello@gmail.com")
    .build();
```

## 02장 아이템 2 : 생성자에 매개변수가 많다면 빌더를 고려하라.

---

책의 Pizza 예제가 계층적인 빌더 패턴을 아주 잘 보여주고 있다!  
아직 자바와 빌드 패턴에 미숙하다면 해당 예제를 반복 학습하자.

빌더 패턴에 장점만 있는 것은 아니다.  
객체를 만들려면, 그에 앞서 빌더부터 만들어야 한다.  
코드가 장황해서 매개변수가 4개 이상은 되어야 값어치를 한다.

하지만 API는 시간이 지날수록 매개변수가 많아지는 경향이 있음을 명심하자.  
매개변수가 많다면, 초기부터 생성자와 정적 팩토리 메서드 대신 빌더 패턴을 사용하자.