

# 이펙티브 자바 CP.1

🕒 작성 일시	@2022년 12월 30일 오후 8:23
🕒 최종 편집 일시	@2023년 1월 2일 오후 8:42
🏷️ 유형	이펙티브 자바
👤 작성자	
👥 참석자	

## 1 객체 생성과 파괴

1. 생성자 대신 정적 팩토리 메소드를 고려해라
2. 생성자에 매개 변수가 많다면 빌더를 고려해라
3. `private` 생성자나 열거 타입으로 싱글턴임을 보증해라
4. 인스턴스화를 막으려거든 `private` 생성자를 사용해라
5. 자원을 직접 명시하지 말고 의존 객체를 주입해라
6. 불 필요한 객체 생성을 피해라
7. 다 쓴 객체 참조를 해제해라
8. `finalizer`, `cleaner` 사용을 피해라
9. `try-finally` 보다는 `try-with-resources`를 사용해라

## 1 객체 생성과 파괴

### ▼ 1. 생성자 대신 정적 팩토리 메소드를 고려해라

`public ClassName()`, `static getWhat()`

클래스는 `public` 생성자 대신 정적 팩토리 메소드를 제공할 수 있다.

정적 팩토리 메소드의 장점

1. 이름을 가질 수 있다.  
명확화 할 수 있다.
2. 호출할 때마다 인스턴스를 새로 생성하지 않아도 된다.  
`lib` 같이 자주 쓰는 파일에 활용 가능해보인다.
3. 반환 타입을 하위 타입 객체로 반환할 수 있는 능력이 있다.
4. 입력 매개변수에 따라 매번 다른 클래스의 객체를 반환할 수 있다.

3, 4 항목 안에서

이건 단점이 아닌가 생각 했지만, `Map` 을 예로 생각해보면 편하다. `TreeMap`, `LinkedMap`, `HashMap` 등을 입력 변수에 따라 변경할 수 있다는건, 유연한 개발이 가능 할 것으로 보인다.

5. 정적 팩토리 메소드는 작성하는 시점에는 반환할 객체의 클래스가 존재하지 않아도 된다.

```
// Order 라는 interface 안에 구현체가 없어도 반환 가능하다.
public static List<Order> getOrders() {
    return new ArrayList<>();
}

interface Order {
}
```

## 6. 서비스 제공자 프레임워크를 만드는 배경이된다. (ex. JDBC)

### 정적 팩토리 메소드의 단점

1. 상속을 하려면 public 이나 protected 생성자가 필요하니, 정적 팩터리 메소드만 제공하면 하위 클래스를 만들 수 없다.

어쩌면, 이 제약은 상속보다 컴포지션(3-4)을 사용한다면, 오히려 장점일 수 있다.

2. 정적 팩터리 메소드는 프로그래머가 찾기 어렵다.

흔히 사용하는 명명 방식

```
// from 매개수를 하나 받아서 해당 타입의 인스턴스를 반환하는 형변환 메소드
Data d = Date.from(instant);

// of 여러 매개변수를 받아 적합한 타입의 인스턴스를 반환하는 집계 메소드
Set<Rank> faceCards = EnumSet.of(JACK, QUEEN, KING);

// instance 혹은 getInstance 매개변수로 명시한 인스턴스를 반환하지만, 같은 인스턴스임을 보장하지 않음
StackWalker luke = StackWalker.getInstance(option);

// create 혹은 newInstance, 위와 같지만, 매번 새로운 인스턴스를 반환함을 보장함
Object newArray = Array.newInstance(classObject, arrayLen);

// getType getInstance 와 같으나, 생성할 클래스가 아닌 다른 클래스에 팩터리 메소드를 정의할 때 사용한다.
FileStore fs = Files.getFileStore(path);

// newType newInstance 와 같으나, 위와 같음
BufferedReader br = Files.newBufferedReader(paht);

// type 위 getType, newType의 간결한 버전
List<Complaint> litany = Collections.list(legacyLitany);
```

## ▼ 2. 생성자에 매개 변수가 많다면 빌더를 고려해라

정적 팩토리 메소드든, 생성자든 선택적 매개 변수가 많다면, 점층적 생성자 패턴, 자비빈즈 패턴, 빌더 패턴 가 있다.

### 1. 점층적 생성자 패턴

```
public class Unit {
    private final int hp;
    private final int mp;
    private final int moveSpeed;
    private final int damage;
    private final int armor;
    private final int shield;
```

```

public Unit(int hp, int moveSpeed, int damage, int armor) {
    Unit(hp, 0, moveSpeed, damage, armor, 0);
}

public Unit(int hp, int mp, int moveSpeed, int damage, int armor) {
    Unit(hp, mp, moveSpeed, damage, armor, 0);
}

public Unit(int hp, int moveSpeed, int damage, int armor) {
    Unit(hp, 0, moveSpeed, damage, armor, 0);
}

public Unit(int hp, int mp, int moveSpeed, int damage, int armor, int shield) {
    this.hp = hp;
    this.mp = mp;
    this.moveSpeed = moveSpeed;
    this.damage = damage;
    this.armor = armor;
    this.shield = shield;
}
}

```

- 이 클래스의 인스턴스를 만들려면 원하는 매개변수를 모두 포함한 생성자 중 가장 짧은 것을 골라 호출하면 된다. 여기서 몇몇 오버로딩된 생성자들은 0이라는 값을 넘기는 매개 변수들이 있다. 지금은 매개 변수 수가 많이 없어서 괜찮게? 보일 수 있지만 수가 100개 이상이라면 해당 코드를 변경하기 어려울 것이다.
- 요약, 사용은 할 수 있겠으나, 매개변수 개수가 많아지면 생성자를 늘려가면서 코드를 작성해야하고, 해당 코드의 변경, 읽기 어려워 진다.

## 2. 자바빈즈 패턴

```

public class Unit {
    private final int hp;
    private final int mp;
    private final int moveSpeed;
    private final int damage;
    private final int armor;
    private final int shield;

    public Unit () {}
    public void setHp(int hp) { this.hp = hp; }
    public void setMp(int mp) { this.mp = mp; }
    public void setMoveSpeed(int moveSpeed) { this.moveSpeed = moveSpeed; }
    public void setDamage(int damage) { this.damage = damage; }
    public void setArmor(int armor) { this.armor = armor; }
    public void setShield(int shield) { this.shield = shield; }
}

Unit zealot = new Unit();
zealot.setHp(100);
zealot.setMoveSpeed(8);
zealot.setDamage(16);
zealot.setArmor(0);
zealot.setShield(100);

```

- 매개변수가 없는 생성자로 객체를 생성 후 setter 메소드를 통해 매개변수의 값을 설정하는 방식이다.

- 자바빈즈 패턴으로는 생성자 수를 늘리지 않아도 되어 점층적 생성자 패턴의 단점이 더이상 보이지 않는다. 하지만, 단점으로 객체 하나를 만들기 위해 메소드를 여러 개 호출해야 하고, 객체가 완전히 생성되기 전까지는 **일관성(멀티 스레드 환경이라고 가정)**이 무너진 상태이다. 즉, 클래스를 불변(3-3)으로 만들 수 없으며, 스레드 안정성을 얻기 위해선, 추가 작업을 해야 한다.
- 요약, 점층적 생성자 패턴의 단점을 극복할 수 있겠으나, 다른 단점들이 크다.

### 3. 빌더 패턴

```
public class Unit {
    private final int hp;
    private final int mp;
    private final int moveSpeed;
    private final int damage;
    private final int armor;
    private final int shield;

    public Unit(Bulider builder) {
        hp = builder.hp;
        mp = builder.mp;
        moveSpeed = builder.moveSpeed;
        damage = builder.damage;
        armor = builder.armor;
        shield = builder.shield;
    }

    public static class Builder {
        // 필수 매개변수
        private final int hp;
        private final int moveSpeed;
        private final int damage;
        private final int armor;

        // 선택 매개 변수 기본값으로 초기화
        private int mp = 0;
        private int shield = 0;

        public Builder(int hp, int moveSpeed, int damage, int armor) {
            this.hp = hp;
            this.moveSpeed = moveSpeed;
            this.damage = damage;
            this.armor = armor;
        }
        public Builder mp(int val) {
            this.mp = val;
            return this;
        }
        public Builder shield(int val) {
            this.shield = val;
            return this;
        }

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

Unit unit = new Unit.Builder(100, 6, 16, 0).shield(100).build();
```

- Unit 클래스는 불변(어떤 변경도 허용하지 않겠다는 뜻 - private, set 메소드 존재 X)이며, 모든 매개 변수의 기본값을 한 곳에 모아 둔다. 빌더의 setter 메서드는 빌더 자신을 반환하기 때문에, 연쇄적 호출이 가능하다. (aka. 플루언트 API, 멍세드 연쇄)
- 빌더 패턴은 계층적으로 설계된 클래스와 함께 사용하기 좋다.
  - 각 계층의 클래스에 관련 빌더를 멤버로 정의하고 추상 클래스는 추상 빌더를, 구체 클래스는 구체 빌더를 갖게 한다.

```
public abstract class Unit {

    public enum Skill { CLOCKING, STORM, BURROW, LOCKDOWN }
    final Set<Skill> skills;
    final int hp;
    final int moveSpeed;
    final int damage;
    final int armor;

    abstract static class Builder<T extends Builder<T>> {
        EnumSet<Skill> skills = EnumSet.noneOf(Skill.class);
        public T addSkill(Skill skill) {
            skills.add(Objects.requireNonNull(skill));
            return self();
        }

        abstract Unit build();

        // 반드시, 하위 클래스는 이 메서드를 오버라이딩 하여 this 를 반환하도록 해야한다.
        protected abstract T self();
    }

    Unit(Builder<?> builder, int hp, int moveSpeed, int damage, int armor) {
        skills = builder.skills.clone(); // item 50 에서 다뤄볼 내용
        this.hp = hp;
        this.moveSpeed = moveSpeed;
        this.damage = damage;
        this.armor = armor;
    }
}

public class Ghost extends Unit {
    private final int mp;

    public static class Builder extends Unit.Builder<Builder> {
        private final int mp;

        public Builder(int mp) {
            this.mp = mp;
        }

        // 오버라이딩에서 반환 타입을 해당 하는 하위 클래스로 반환하도록 한다.
        // aka. 공변 반환 타이핑 - 객체 생성시 캐스팅하지 않아도 된다.
        @Override
        public Ghost build() {
            return new Ghost(this);
        }

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

```

private Ghost(Builder builder) {
    super(builder, 45, 5, 10, 0);
    mp = builder.mp;
}

}

Ghost ghost = new Ghost.Builder(50)
    .addSkill(Unit.Skill.CLOCKING)
    .addSkill(Unit.Skill.LOCKDOWN)
    .build();

```

- 더 나아가가기, 유효성 검사의 타이밍은 빌더 생성자와 메서드에서 입력 매개변수를 검사하고, build 메서드가 호출하는 생성자에서 여러 매개변수에 걸친 불변식을 검사한다.

### ▼ 3. private 생성자나 열거 타입으로 싱글턴임을 보증해라

- 싱글턴이란 인스턴스를 오직 하나만 생성 할 수 있는 클래스를 뜻 함

주로 stateless 객체나 설계상 유일해야 하는 시스템 컴포넌트

단점으로는, 이를 사용하는 클라이언트 코드를 테스트하기 어렵다.

- 싱글턴을 만드는 방식

#### 1. public static 멤버 변수가 final 인 경우

- 생성자는 private 로 감쳐두고, 유일한 인스턴스를 접근할 수단 public static final 멤버 변수를 하나 둔다.
- private 생성자는 Singleton.INSTANCE 를 초기화할 때, 딱 한 번 호출된다.
- public 이나 protected 생성자가 없음으로 클래스가 초기화 될 때 만들어진 인스턴스가 전체 시스템에서 하나뿐이 보장된다.\
- 장점
  - 해당 클래스가 싱글턴임이 API에 명백히 드러난다.public static 필드가 final이니 절대로 다른 객체를 참조할 수 없다.
  - 간결하다.

```

public class Singleton {
    public static final Singleton INSTANCE = new Singleton();
    private Singleton() {...}
}

```

#### 2. 정적 팩터리 메소드를 public static 멤버 변수를 제공하는 경우

- Singleton.getInstance 는 항상 같은 객체의 참조를 반환하므로, 인스턴스를 결코 만들어 지지 않는다.
- 장점
  - 언제든지 싱글턴에서 여러 인스턴스를 반환하는 방식으로 변경 가능하다.

- 원한다면 정적 팩터리를 제네릭 싱글톤 팩터리로 만들 수 있다.
- 정적 팩터리의 메서드 참조를 공급자(supplier)로 사용할 수 있다.

```
Supplier<Singleton> singletonSupplier = Singleton::getInstance;
Singleton singleton = singletonSupplier.get();
```

```
public class Singleton {
    private static final Singleton INSTANCE = new Singleton();
    private Singleton() {...}
    public static Singleton getInstance() { return INSTANCE; }
}
```

- 위 두 방식 모두의 단점
  - 권한이 있는 클라이언트에서 리플렉션 API 인 `AccessibleObject.setAccessible` 을 이용해 `private` 생성자를 호출 할 수 있다.
    - 해결방안 - `private` 생성자에 해당 코드를 추가한다.

```
private Singleton() {
    if(INSTANCE != null){
        throw new RuntimeException("생성자 호출이 불가능합니다.");
    }
}
```

- 직렬화된 인스턴스를 역직렬화할 때 새로운 인스턴스를 만들어서 반환한다. 역직렬화는 기본 생성자를 호출하지 않고 값을 복사해서 새로운 인스턴스를 반환한다. 그때 사용되는 것이 `readResolve` 라는 메소드이다.
  - 해결방안 - 아래 코드를 추가한다.

```
private Object readResolve() {
    return INSTANCE;
}
```

- 번외. (백기선님의 싱글톤 강의)
  - 만약, 위 싱글톤 내용(이른 초기화)들이 무거운 객체들이고 언제 사용될지 모를 경우 비효율적인 객체를 생성해서 보관 할 수 있다. 이를 방지해 맨 처음 사용될 때, 사용하는 방식이 있다.
    - double checked locking - 복잡한 방식

```
public class Singleton {
    public static volatile Singleton INSTANCE;
    private Singleton() {}
    public static Singleton getInstance() {
        if (INSTANCE == null) {
            // 동시 접근시 막기 위함.
            synchronized (Singleton.class) {

```

```

        if (INSTANCE == null) {
            INSTANCE = new Singleton();
        }
    }
}
return INSTANCE;
}
}

```

- static inner class - 비교적 간단한 방식

```

public class Singleton {
    private Singleton() {}

    private static class SingletonHolder {
        private static final Singleton INSTANCE = new Singleton();
    }

    public static Singleton getInstance() {
        return SingletonHolder.INSTANCE;
    }
}

```

- ▼ 4. 인스턴스화를 막으려거든 private 생성자를 사용해라
- ▼ 5. 자원을 직접 명시하지 말고 의존 객체를 주입해라
- ▼ 6. 불 필요한 객체 생성을 피해라
- ▼ 7. 다 쓴 객체 참조를 해제해라
- ▼ 8. finalizer, cleaner 사용을 피해라
- ▼ 9. try-finally 보다는 try-with-resources를 사용해라