Effective Java

기본 원칙

- 명료성
- 단순성
- 컴포넌트는 정해진 동작이나 예측 가능한 동작만 수행해야 한다.
- 컴포넌트는 가능한 한 작되, 너무 작아서는 안된다.
  - 컴포넌트는 개별 메서드부터 여러 패키지로 이뤄진 복잡한 프레임워크까지 재사용 가능한 모든 SW 요소를 말한다.
- 코드는 복사가 아니라 재사용되어야 한다.
- 컴포넌트 사이의 의존성은 최소로 유지해야 한다.
- 오류는 만들어지자마자 가능한 한 빨리 잡아야 한다. ← 컴파일타임에 잡는게 가장 좋다.

객체 생성과 파괴

Item 1. 생성자 대신 정적 팩터리 메서드를 고려하라.

클래스에서 public 생성자 대신에 정적 팩터리 메서드를 제공하는 방법은 다음과 같은 장점을 가진다.

장점1) 이름을 가질 수 있다.

정적 팩터리 메서드를 통해 반환되는 객체의 특성을 설명할 수 있다. 메서드나 객체의 이름을 잘 지으면 이를 사용하는 개발자는 이름만 보고도 어떤 동작을 할지, 어떤 특성을 가지는지 알 수 있다. 따라서 한 클래스에 시그니처가 같은 생성자가 여러개 필요하다면 정적 팩터리 메서드를 통해 각각의 이름을 잘 지어주는 것이 좋다.

장점2) 호출될 때마다 인스턴스를 새로 생성하지 않아도 된다.

`static`으로 인스턴스를 미리 만들어 놓거나 생성한 인스턴스를 캐싱하여 재활용하는 식으로 불필요한 객체 생성을 피할 수 있다. 생성 비용이 큰 객체가 자주 요청되는 상황에서 성능을 상당히 끌어올릴 수 있다. 또한 불변 값 클래스에서 동치인 인스턴스가 단 하나뿐임을 보장할 수 있다(`a == b` ➡ `a.equals(b)` 성립).

```
public class StaticClass {
  // static을 통해 인스턴스를 미리 만들어둔다.
  private static final StaticClass STATIC_CLASS = new StaticClass();

  private StaticClass() {}

  // 미리 만들어둔 인스턴스를 반환한다.
```

```
    public static StaticClass getInstance() {
        return STATIC_CLASS;
    }
}


public Static void main(String args[]){
    StaticClass sc1 = StaticClass.getInstance();
    StaticClass sc2 = StaticClass.getInstance();

    // sc1 == sc2 : true
    System.out.println("sc1 == sc2 : " + (sc1 == sc2))
}
```

장점3) 반환 타입의 하위 타입 객체를 반활할 수 있다.
반환할 객체의 클래스를 자신의 하위 타입 중에서 선택할 수 있어 유연성이 크다. 이를 통해 구현 클래스를 공개하지 않고도 그 객체를 반환할 수 있어 API를 작게 유지할 수 있다.

장점4) 입력 매개변수에 따라 매번 다른 클래스의 객체를 반활할 수 있다.
같은 이름의 정적 팩터리 메서드를 여러개 만들고 입력 매개변수를 다르게 하여 메서드마다 다른 하위타입 클래스 객체를 반환할 수 있다. 장점 3과 비슷한 의미이다.

장점5) 정적 팩터리 메서드를 작성하는 시점에는 반환할 객체의 클래스가 존재하지 않아도 된다.
메서드에서 객체 반환시 당장 클래스가 존재하지 않아도 특정 텍스트 파일에서 인터페이스 구현체의 위치를 알려주는 곳의 정보를 가지고 해당 객체를 읽어 생성할 수 있다. 이러한 유연함은 서비스 제공자 프레임워크를 만드는 근간이 되었고 대표적인 예로 JDBC가 있다.

반면 정적 팩터리 메서드는 다음과 같은 단점도 가진다.
단점1) 상속을 위해 public과 protected 생성자가 필요하다. 정적 팩터림 메서드만 제공하면 하위 클래스를 만들 수 없다.

단점2) 정적 팩터리 메서드는 프로그래머가 찾기 어렵다.
생성자처럼 API 설명에 명확히 드러나지 않기 때문에 사용자는 정적 팩터리 메서드 방식 클래스를 인스턴스화할 방법을 알아내야 한다. 따라서 API 문서를 잘 써놓고 메서드 이름도 널리 알려진 규약을 따라 짓는 식으로 문제를 완화할 필요가 있다.

정리

정적 팩터리 메서드와 생성자는 각자 쓰임새가 있다. 무작정 하나의 방식만을 사용하는 것은 좋지 않다. 장단점을 비교하여 적절한 방식을 사용하는 것이 좋다.

Item 2. 생성자에 매개변수가 많으면 빌더를 고려하라.

필수 매개변수와 선택 매개변수를 여러개 받아야 하는 경우 생성자를 이용하면 생성자의 개수가 너무 많아지고, 사용자가 설정하길 원치 않는 매개변수까지 포함하기 쉽다. ← 점층적 생성자 패턴, 매개변수 개수가 많아지면 클라이언트 코드를 작성하거나 읽기 어렵다.

선택 매개변수가 많을 때, 자바빈즈 패턴을 활용할 수 있다. 매개변수가 없는 생성자로 객체를 생성하고 setter 메서드들을 통해 원하는 매개변수의 값을 설정하는 방식이다. 이는 객체 하나를 만들기 위해 메서드를 여러개 호출해야 하고 객체가 완전히 생성되기 전까지는 일관성이 무너진 상태에 놓인다.

위 문제들을 해결하기 위해 점층적 생성자 패턴의 안전성과 자바빈즈 패턴의 가독성을 겸비한 **빌더 패턴**이 있다. 클라이언트는 필요한 객체를 직접 만드는 대신에 필수 매개변수만으로 생성자를 호출하여 빌더 객체를 얻는다. 이후 빌더 객체가 제공하는 일종의 setter 메서드들로 원하는 선택 매개변수들을 설정한다. 마지막으로 매개변수가 없는 build 메서드를 호출하여 필요한 객체를 얻는다.

빌더 패턴은 다음과 같은 **장점**을 가진다.

- 빌더를 이용하면 가변인수 매개변수를 여러개 사용 가능하다.
- 빌더 하나로 여러 객체를 순회하면서 만들 수 있다.
- 빌더에 넘기는 매개변수에 따라 다른 객체를 만들 수 있다.
- 특정 필드는 필드가 알아서 채우도록 할 수도 있다.

반면 다음과 같은 **단점**도 있다.

- 객체를 만드려면 빌더부터 만들어야 한다. 빌더 생성 비용이 크지는 않으나 성능에 민감한 상황에서는 문제가 될 수 있다.
- 점층적 생성자 패턴보다는 코드가 장황하다. 매개변수가 적다면 그 값어치를 하지 못할 수 있다.

결론

빌더 패턴은 점층적 생성자 패턴보다 클라이언트 코드를 읽고 쓰기가 훨씬 간결하고, 자바빈즈 패턴보다 훨씬 안전하다.

Item 3. private 생성자나 열거 타입으로 싱글턴임을 보증하라.

#Effective Java#