2장

2장 객체 생성과 파괴

객체를 만들어야 할 때와 만들지 말아야 할 때를 구분하는 법, 올바른 객체 생성 방법과 불필요한 생성을 피하는 방법, 제때 파괴됨을 보장하고 파괴 전에 수행해야 할 정리 작업을 관리하는 요령

아이템1

public 생성자 대신 정적 팩터리 메서드 제공의 장점

- 이름을 가질 수 있다. → 생성자에 넘기는 매개변수, 생성자로는 객체의 특성을 제대로 설명하지 못함.
- 호출될 때마다 인스턴스를 새로 생성하지는 않아도 된다. → 불필요한 객체생성 피함. 생성비용이 큰 객체가 자주 요청되는 상황이라면 성능 UP!
- 반환 타입의 하위 타입 객체를 반환할 수 있는 능력이 있다. → 반환할 객체의 클래스를 자유롭게 선택하는 엄청난 유연성
- 입력 매개변수에 따라 매번 다른 클래스의 객체를 반환할 수 있다.
- 정적 팩터리 메서드를 작성하는 시점에는 반환할 객체의 클래스가 존재하지 않아도 된다.

단점

- 상속을 하려면 public이나 protected 생성자가 필요하니 정적 팩터리 메서드만 제공하면 하위 클래스를 만들 수 없다.
- 정적 팩터리 메서드는 프로그래머가 찾기 어렵다.

핵심! 정적 팩터리 메서드와 public 생성자는 각자의 쓰임새가 있으니 장단점을 이해하고 사용하자! 정적 팩터리를 사용하는게 유리한 경우가 더 많으므로 무작정 public 생성자를 사용하는 습관 고치자!

아이템2

생성자나 정적 팩터리가 처리해야 할 매개변수가 많다면 빌더 패턴을 선택하는게 더 낫다! 빌더는 점층적 생성자보다 클라이언트 코드를 읽고 쓰기가 훨씬 간결하고, 자바빈즈보다 안전하다.

아이템3

싱글턴이란? → 인스턴스를 오직 하나만 생성할 수 있는 클래스. ex) 함수와 같은 무상태 객체, 시스템 컴포넌트 클래스를 싱글턴으로 만들면 이를 사용하는 클라이언트를 테스트하기가 어려워질 수 있다.

싱글턴을 만드는 방식

- 생성자는 private로 감춰두고, 유일한 인스턴스에 접근할 수 있는 수단으로 public static 멤버를 마련.
- 정적 팩터리 메서드를 public static 멤버로 제공
- 원소가 하나인 열거 타입을 선언.

대부분 상황에서는 원소가 하나뿐인 열거 타입이 싱글턴을 만드는 가장 좋은 방법.

아이템4

인스턴스화를 막는 방법은 private 생성자를 추가하면 클래스의 인스턴스화를 막을 수 있다 → 상속을 불가능케 하는 효과도 있으며, 모든 생성자는 상위 클래스의 생성자를 호출하게 되는데, 이를 private으로 선언하여 하위 클래스가 상위 클래스의 생성자에 접근할 길이 막히도록 함.

아이템5

클래스가 내부적으로 하나 이상의 자원에 의존하고, 그 자원이 클래스 동작에 영향을 준다면 싱글턴과 정적 유틸리티 클래스는 사용하지 않는 것이 좋다. 이 자원들을 클래스가 직접 만들게 해서도 안됨. 대신 필요한 자원을 생성자에 넘겨주자. 의존 객체 주입이라 하는 이 기법은 클래스의 유연성, 재사용성, 테스트 용이성을 기막히게 개선해줌.

아이템6

```
String s = new String("bikini"); //절대 하지 말 것! String인스턴스
String s = "bikini"; //개선된 버전, 하나의 String 인스턴스 사용
```

생성자 대신 정적 팩터리 메서드를 사용해 불필요한 객체 생성을 피할 수 있다. 생성자는 호출할 때마다 새로운 객체를 만들지만, 팩터리 메서드는 전혀 그렇지 않다.

"객체 생성이 비싸니 피해야 한다"가 아님! 프로그램의 명확성, 간결성, 기능을 위한 것이 아니라면 "기존 객체를 재사용해야 한다면 새로운 객체를 만들지 말자"!

아이템7

C,C++처럼 메모리를 직접 관리해야 하는 언어와 다르게 자바처럼 가비지 컬렉터를 갖춘 언어는 객체를 알아서 회수해 간다. 메모리 관리에 신경을 안써도 된다고 생각할 수 있는데, 오해이다!

'메모리 누수'가 있는 코드는 오래 실행하다 보면 점차 가비지 컬렉션 활동과 메모리 사용량이 늘어나 성능 저하가 유발된다. 심한경우 디스크 페이징과 OutOfMemoryError를 일으킨다.

```
public Object pop() {
    if (size == 0)
            throw new EmptyStackException();
    Object result = elements[--size];
    elements[size] = null; //참조해제
    return result;
}
```

다 쓴 참조를 null처리하여 참조 해제를 구현했다. 하지만, 일일이 항상 Null처리는 프로그램을 지저분하게 만들 뿐, null 처리를 하는 일은 예외적인 경우여야 한다. 다 쓴 참조를 해제하는 가장 좋은 방법은 참조를 담은 변수를 유효 범위 밖으로 밀어내는 것이다.

✍️ 항상 메모리 누수에 주의하자!

아이템8

자바는 두 가지 객체 소멸자 finalizer,cleaner가 있다. finalizer는 예측할 수 없고 위험할 수 있어 일반적으로 불필요하다. cleaner는 덜 위험하지만, 여전히 예측할 수 없고, 느리고 일반적으로 불필요하다.

때문에 소멸자 사용을 피하고 AutoCloseable을 구현하면 된다.

✍️ cleaner는 안전망 역할이나 중요하지 않은 네이티브 자원 회수용으로만 사용하자.

아이템9(try-finally보다는 try-with-resources를 사용하라)

자원이 제대로 닫힘을 보장하는 수단으로 try-finally를 사용해왔지만 자원이 둘 이상일 경우 지저분하고 디버깅이 어려워진다. 자바7에서의 try-with-resources를 사용하면 된다. 이 구조를 사용하려면 AutoCloseable 인터페이스를 구현해야 한다.

꼭 회수해야 하는 자원을 다룰 때는 Try-finally 말고, try-with-resources를 사용하자. 예외없다!