<2장> 객체 생성과 파괴

일자 @2024년 6월 12일

> **이번 장은 객체의 생성과 파괴를 다룬다.**
>
> **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 요령을 알아본다.**

▼ [아이템 1] 생성자 대신 정적 팩토리 메소드를 고려하라

정적 팩토리 메소드 (Static Factory Method)

클래스의 인스턴스를 반환하는 단순한 정적 메소드.
디자인 패턴 중에는 이와 일치하는 패턴은 없다.
아래 예시는 박싱 클래스인 Boolean의 정적 팩토리 메소드이다.

```
public static Boolean valueOf(boolean b){
        return b ? Boolean.TRUE : Boolean.FALSE;
}
```

정적 팩토리 메소드의 장점

1. 이름을 가질 수 있다

- 생성자로는 반환될 객체의 특성을 제대로 설명하지 못한다
- 반면 정적 팩토리 메소드는 이름만 잘 지으면 반환될 객체의 특성을 쉽게 묘사할 수 있다
  - ex) BigInteger.probablePrime(int bitLength, Random rnd)
- 한 클래스에 시그니처가 같은 생성자가 여러 개 필요할 것 같으면 생성자를 정적 팩토리 메소드로 만들고 이름을 잘 지어주자

2. 호출될 때마다 인스턴스를 새로 생성하지 않아도 된다

- 불변 클래스(immutable class; 아이템17)는 인스턴스를 미리 만들어 놓거나 새로 생성한 인스턴스를 캐싱하여 재활용하는 식으로 불필요한 객체 생성을 피할 수 있다
    - 반복되는 요청에 같은 인스턴스를 반환하게 하여 언제 어느 인스턴스를 살아있게 할 지를 통제할 수 있는 **인스턴스 통제(instance-controlled) 클래스**로 만들 수 있다
    - ex) Boolean.valueOf(boolean): Boolean.TRUE / Boolean.FALSE를 반환
- 플라이웨이트 패턴(Flyweight Pattern)도 이와 비슷한 기법이다

3. 반환 타입의 하위 타입 객체를 반환할 수 있는 능력이 있다

- 반환할 객체의 클래스를 자유롭게 선택할 수 있는 **유연성**을 제공한다

4. 입력 매개변수에 따라 매번 다른 클래스의 객체를 반환할 수 있다

- 반환 타입의 하위 타입이기만 하면 어떤 클래스의 객체를 반환하든 상관 없다 (또다른 클래스를 반환해도 된다)
    - ex) EnumSet 클래스(아이템36): public 생성자 없이 정적 메소드만 지원하며, 원소의 개수가 64개 이하면 RegularEnumSet, 초과면JumboEnumSet를 반환한다.

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

- **서비스 제공자 프레임워크**를 만드는 근간이 되는 유연함을 제공한다
    - ex) JDBC(Java Database Connectivity)

📕 **서비스 제공자 프레임워크(service provider framework)의 핵심 컴포넌트**

- **서비스 인터페이스(service interface)**
  - 구현체의 동작 정의
- **제공자 등록 API(provider registration API)**
  - 제공자가 구현체를 등록할 때 사용
- **서비스 접근 API(service access API)**
  - 클라이언트가 서비스의 인스턴스를 얻을 때 사용
  - ***유연한 정적 팩토리의 실체***

서비스 제공자 프레임워크에서 제공자(provider)는 서비스의 구현체이다.

이 구현체들을 클라이언트에 제공하는 역할을 프레임워크가 통제하여 클라이언트를 구현체로부터 분리해준다.

정적 팩토리 메소드의 단점

1. 상속을 하려면 public이나 protected 생성자가 필요하므로, 정적 팩토리 메소드만 제공해서는 하위 클래스를 만들 수 없다

- 단, 상속보다 컴포지션을 사용(아이템 18)하도록 유도하고 불변 타입(아이템 17)으로 만드려면 오히려 장점으로 받아들일 수도 있다

2. 정적 팩토리 메소드는 프로그래머가 찾기 어렵다

- 생성자처럼 API 설명에 명확히 드러나지 않으므로 사용자는 정적 팩토리 메소드 방식 클래스를 인스턴스화할 방법을 알아내야 한다

정적 팩토리 메소드에 주로 사용되는 명명 방식

메소드명	설명
from	매개변수를 하나 받아 해당 타입의 인스턴스를 반환하는 형변환 메소드 ex) `Date d = Date.from(instance);`
of	여러 매개변수를 받아 적합한 타입의 인스턴스를 반환하는 집계 메소드 ex) `Set<Rank> faceCards = EnumSet.of(JACK, QUEEN, KING);`
valueOf	from과 of의 더 자세한 버전 ex) `BigInteger prime = BigInteger.valueOf(Integer.MAX_VALUE);`
instance / getInstance	(매개변수를 받는다면) 매개변수로 명시한 인스턴스를 반환. 같은 인스턴스임을 보장하지는 않음 ex) `StackWalker luke = StackWalker.getInstance(options);`
create / newInstance	instance / getInstance와 같지만, 매번 새로운 인스턴스를 생성해 반환함을 보장 ex) `Object newArray = Array.newInstance(classObject, arrayLen);`
getType	getInstance와 같으나, 생성할 클래스가 아닌 다른 클래스에 팩토리 메소드를 정의할 때 사용. Type은 팩토리 메소드가 반환할 객체의 타입 ex) `FileStore fs = Files.getFileStore(path);`
newType	newInstance와 같으나, 생성할 클래스가 아닌 다른 클래스에 팩토리 메소드를 정의할 때 사용. Type은 팩토리 메소드가 반환할 객체의 타입 ex) `BufferedReader br = Files.newBufferedReader(path);`
type	getType과 newType의 간결한 버전 ex) `List<Complaint> litany = Collections.list(legacyLitany);`

▼ [아이템 2] 생성자에 매개변수가 많다면 빌더를 고려하라

정적 팩토리와 생성자의 생성자 패턴

1. 점층적 생성자 패턴

- 선택 매개변수 조합마다 생성자(정적 팩토리)를 일일이 새로 만드는 방식
- 선택적 매개변수가 많아지면 클라이언트 코드를 작성하거나 읽기 어렵다

2. 자바빈즈 패턴

- Setter를 사용한 방식
- 객체 하나를 생성하기 위해 여러 메소드를 호출해야 하고, 객체가 완성되기 전까지는 일관성(consistency)이 무너진 상태에 놓이게 된다
    - 따라서 클래스를 불변(아이템 17)으로 만들 수 없다
    - 스레드 안정성을 얻기 위해서는 프로그래머가 추가 작업을 해줘야 한다

3. ★빌더 패턴★

- 필수 매개변수만으로 빌더클래스를 생성한 뒤 선택 매개변수를 설정한 다음 build 메소드를 이용해 목표 객체를 얻는 방법
- 점층적 생성자 패턴과 자바빈즈 패턴의 장점만 취했다
- 메소드 호출이 흐르듯 연결되어 **플루언트 API(fluent API)** 혹은 **메소드 연쇄(method chaining)**이라 한다
- 계층적으로 설계된 클래스와 함께 쓰기 좋다
- 가변 인수(vargargs) 매개변수를 여러 개 사용할 수 있다

> ☝🏻 빌더 패턴은 (파이썬과 스칼라에 있는) 명명된 선택적 매개변수(named optional parameters)를 흉내낸 방식이다

▼ [아이템 3] private 생성자나 열거 타입으로 싱글턴임을 보증하라

☝🏻 **참고)** 클래스를 싱글톤으로 만들면 이를 사용하는 클라이언트를 테스트하기 어려워질 수 있다!

싱글톤 생성 방법

1. public static final 필드 방식

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

        public void leaveTheBuilding() { ... }
}
```

- Elvis 클래스가 초기화될 때 만들어진 인스턴스가 전체 시스템에서 하나뿐임이 보장된다
- 그러나 권한이 있는 클라이언트는 **리플렉션 API(아이템 65)**인 AccessibleObject.setAccessible을 사용해 private 생성자를 호출할 수 있다
  - 이러한 공격을 방어하려면 생성자를 수정하여 두번째 객체가 생성되려 할 때 예외를 던지면 된다
- 장점
  1. 해당 클래스가 싱글톤임이 API에 명백히 드러난다
  2. 간결하다

2. 정적 팩토리 방식

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

        public void leaveTheBuilding() { ... }
}
```

- public static final 필드 방식과 마찬가지로 인스턴스가 유일하다 (리플렉션을 통한 예외도 똑같이 적용된다)

- 장점
  1. API를 바꾸지 않고도 싱글톤이 아니도록 변경할 수 있다
     - 호출하는 스레드별로 다른 인스턴스를 넘겨주도록 할 수 있다
  2. 원한다면 정적 팩토리를 제네릭 싱글톤 패턴으로 만들 수 있다(아이템 30)
  3. 정적 팩토리의 메소드 참조를 공급자(supplier)로 사용할 수 있다
     - `Elvis::getInstance`를 `Supplier<Elvis>`로 사용하는 식

☝🏻 그러나 위 두 방식으로 싱글톤 작성 시, 직렬화-역직렬화 과정에서 Serializable 선언 외에도 추가적인 작업이 필요하다.

1. 모든 인스턴스 필드를 일시적(transient)이라고 선언한다
2. readResolve 메소드를 제공해야 한다(아이템 89)

이렇게 하지 않으면 직렬화된 인스턴스를 역직렬화할 때 마다 새로운 인스턴스가 만들어진다.

```
//싱글톤임을 보장해주는 readResolve 메소드
private Object readResolve(){
        //진짜 Elvis를 반환하고, 가짜 Elvis는 GC에 맡긴다
        return INSTANCE;
}
```

3. 열거 타입 방식

```
public enum Elvis{
        INSTANCE;

        public void leaveTheBuilding() { ... }
}
```

- public 필드 방식과 비슷하지만, 더 간결하고 추가 노력 없이 직렬화할 수 있다
- 뿐만 아니라 복잡한 직렬화 상황이나 리플렉션 공격에서도 제2의 인스턴스가 생기는 일을 완벽히 막아준다.
- **대부분 상황에서 원소가 하나뿐인 열거 타입이 싱글톤을 만드는 가장 좋은 방법이다**

▼ [아이템 4] 인스턴스화를 막으려거든 private 생성자를 사용하라

- 인스턴스로 만들 필요가 없는 정적 멤버만 담은 유틸리티 클래스라도, 생성자를 명시하지 않으면 컴파일러가 자동으로 디폴트 생성자를 만들어준다
- 단지 추상 클래스를 만드는 것 만으로는 인스턴스화를 막을 수 없다
  - 하위 클래스를 만들어 인스턴스화하면 되기 때문이다
- **이 때, private 생성자를 추가하면 클래스의 인스턴스화를 막을 수 있다**
  - ex) 인스턴스를 만들 수 없는 유틸리티 클래스

```
public class UtilityClass{
        //기본 생성자가 만들어지는 것을 막는다(인스턴스화 방지용)
        private UtilityClass(){
            throw new AssertionError();
        }

        ...
}
```

    - AssertionError를 꼭 던질 필요는 없지만, 클래스 안에서 실수로라도 생성자를 호출하는 것을 막아준다
    - 그러나 생성자가 존재하는데 호출할 수 없다는 것이 그닥 직관적이지는 않으므로, 적절한 주석을 달아주도록 하자
    - 이 방식은 상속을 불가능하게 하는 효과도 있다

▼ [아이템 5] 자원을 직접 명시하지 말고 의존 객체 주입을 사용하라

의존 객체 주입

- 인스턴스를 생성할 때 생성자에 필요한 자원을 넘겨주는 방식

```
public class SpellChecker {
        private final Lexion dictionary;

        public SpellChecker(Lexion dictionary) {
                this.dictionary = Objects.requireNonNul
        }

        public boolean isValid(String word) { ... }
        public List<String> suggestions(String typo) {
}
```

- 사용하는 자원에 따라 동작이 달라지는 클래스에는 정적 유틸리티 클래스나 싱글톤 방식이 적합하지 않다
- 의존 객체 주입은 유연성과 테스트 용이성을 높여준다

⚠ 클래스가 내부적으로 하나 이상의 자원에 의존하고, 그 자원이 클래스 동작에 영향으 ㄹ준다면 싱글톤과 정적 유틸리티 클래스는 사용하지 않는 것이 좋다.

이 자원들을 클래스가 직접 만들게 해서도 안된다. 대신 필요한 자원(혹은 그 자원을 만들어주는 팩토리)을 생성자(혹은 정적 팩토리나 빌더)에 넘겨주자

▼ [아이템 6] 불필요한 객체 생성을 피하라

- 똑같은 기능의 객체는 매번 생성하기 보다는 하나를 재사용하는 편이 나을 때가 많다
- 아래 두 예시는 문자열이 유효한 로마 숫자인지를 확인하는 메소드이다.

```
static boolean isRomanNumeral(String s){
        return s.matches("^(?=.)M*C[MD]|D?C{0,3})"
                        + "X[CL]|L?X{0,3})(I[XV]|V?I{0,3})
}
```

```
public class RomanNumarals{
        public static final Pattern ROMAN = Pattern.compil
```

```
                "^(?=.)M*C[MD]|D?C{0,3})" + "X[CL]|L?X{0,3
        );

        static boolean isRomanNumeral(String s){
                return ROMAN.matcher(s).matches();
        }
}
```

- 불필요한 객체를 만들어내는 또다른 예로, **오토박싱(auto boxing)**이 있다
  - 오토박싱은 기본 타입과 그에 대응되는 박싱된 기본 타입의 구분을 흐려주지만, 완전히 없애주는 것은 아니다
  - **박싱된 기본 타입보다는 기본 타입을 사용하고, 의도치 않은 오토박싱이 숨어들지 않도록 주의하자**

> ⚠ "객체 생성은 비싸니 피해야 한다"로 오해하면 안된다. 요즘의 JVM에서는 별다른 일을 하지 않는 작은 객체를 생성하고 회수하는 일은 크게 부담되지 않는다.
>
> 거꾸로, 아주 무거운 객체가 아닌 한 객체 풀(pool)을 만들지 말자. 일반적으로 객체 풀은 코드를 헷갈리게 만들고 메모리 사용량을 늘리고 성능을 저하시킨다. 요즘 JVM의 GC는 상당히 잘 최적화되어서 가벼운 객체용을 다룰 때에는 직접 만든 객체 풀보다 훨씬 빠르다.

▼ [아이템 7] 다 쓴 객체 참조를 해제하라

자바의 메모리 누수

- C/C++과 달리 자바는 다 쓴 객체를 알아서 회수해 가므로 메모리 관리를 하지 않아도 된다고 생각할 수 있지만, 그렇지 않다. 아래는 자바의 메모리 누수 예시이다

```
public class Stack{
        private Object[] elements;
        private int size = 0;
        private static final int DEFAULT_INITIAL_CAPACITY =

        public Stack() {
                elements = new Object[DEFAULT_INITIAL_CAPA
        }

        public void push(Object e){
```

```
                ensureCapacity();
                elements[size++] = e;
        }

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

        /**
        * 원소를 위한 공간을 적어도 하나 이상 확보한다.
        * 배열 크기를 늘려야 할 때마다 대략 두 배씩 늘린다.
        */
        private void ensureCapacity(){
                if (elements.length == size)
                        elements = Arrays.copyOf(elements,
        }
 }
```

  - 해당 코드에서는 스택이 크기가 늘었다가 줄어들 때 꺼내진 객체들을 GC가 회수하지 못해 메모리 누수가 발생한다. 이 스택이 객체들의 다 쓴 참조(obsolete reference)를 여전히 가지고 있기 때문이다.

- 이를 해결하기 위해, 해당 참조를 다 쓴 뒤 null 처리(참조 해제)하면 된다.
  아래는 올바른 pop 메소드이다

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

- 그러나 모든 객체를 참조 해제하기 위해 일일이 null 처리할 필요는 없다
  - 오히려 코드가 필요 이상으로 지저분해질 수 있다
  - 객체 참조를 null 처리하는 일은 예외적인 경우여야 한다

- 다 쓴 참조를 해제하는 가장 좋은 방법은 그 참조를 담은 변수를 유효 범위(scope) 밖으로 밀어내는 것이다

자바에서 메모리 누수가 발생할 수 있는 경우

1. 자기 메모리를 직접 관리하는 클래스

- 이 경우에는 원소를 다 사용한 즉시 null 처리하여 해당 객체를 더이상 쓰지 않을 것임을 GC에 알려야 한다

2. 캐시

- 객체 참조를 캐시에 넣은 채로 방치할 때에도 메모리 누수가 발생한다
- 캐시 외부에서 키(key)를 참조하는 동안에만 엔트리가 살아있는 캐시가 필요한 상황이라면 **WeakHashMap**을 사용하자
  - 다 쓴 엔트리는 그 즉시 자동으로 삭제된다
  - WeakHashMap은 이 상황에서만 유용하다

3. 리스너(listener) 또는 콜백(callback)

> ⚠ 메모리 누수는 겉으로 드러나지 않아 수년간 잠복하는 사례도 있다. 따라서 이런 종류의 문제는 예방법을 미리 익혀두는 것이 중요하다.

▼ [아이템 8] finalizer와 cleaner 사용을 피하라

- 자바는 두 가지 객체 소멸자를 제공한다.
- 그 중 finalizer는 예측할 수 없고, 상황에 따라 위험할 수 있어 일반적으로 불필요하다.
- cleaner는 finalizer보다 덜 위험하지만, 여전히 예측할 수 없고, 느리고, 일반적으로 불필요하다.

finalizer/cleaner의 문제점

1. 제 때 실행되어야 하는 작업은 절대 할 수 없다

- 즉시 수행된다는 보장이 없다
- 특히, 파일 닫기를 finalizer/cleaner에게 맡기면 중대한 오류를 일으킬 수 있다(동시에 열 수 있는 최대 파일 개수가 존재하기 때문이다)

2. 상태를 영구적으로 수정하는 작업에서는 절대 의존하면 안된다

- 수행 시점 뿐 아니라 수행 여부조차 보장하지 않는다
- 접근할 수 없는 일부 객체에 딸린 종료 작업을 수행하지 않은 채 프로그램이 종료될 수 있기 때문이다

3. 심각한 성능 문제도 동반한다

- 기본적으로 느리다. finalizer가 GC의 효율을 떨어뜨리기 때문이다
- 단, 추후 서술될 안전망 형태로만 사용하면 훨씬 빨라진다

4. finalizer를 사용한 클래스는 finalizer 공격에 노출되어 심각한 보안 문제를 야기할 수 있다

- 객체 직렬화 과정에서 예외가 발생하면 악의적인 하위 클래스의 finalizer를 수행하는 finalizer 공격이 일어날 수 있다
  - 이 finalizer는 정적 필드에 자신의 참조를 할당해 GC가 수집하지 못하게 막을 수 있다
  - 객체 생성을 막으려면 생성자에서 예외를 던지는 것으로 충분하지만, finalizer가 있다면 그렇지도 않다
- final이 아닌 클래스를 finalizer 공격으로부터 방어하려면 아무 일도 하지 않는 **finalize** 메소드를 만들고 final로 선언하자
- 이를 대체하기 위해, AutoClosable을 구현해주고, 해당 인스턴스를 다 쓴 뒤에 close 메소드를 호출해주면 된다
  - 예외가 발생해도 제대로 종료되도록 try-with-resourecs를 사용해야 한다 (아이템 9)

☝🏻 **finalizer/cleaner의 쓰임새는?**

1. 자원의 소유자가 close 메소드를 호출하지 않는 것에 대비한 안전망 역할
2. 네이티브 피어(native peer)와 연결된 객체에서 활용
   - 네이티브 피어
     - 일반 자바 객체가 네이티브 메소드를 통해 기능을 위임한 객체
     - 자바 객체가 아니므로 GC가 그 존재를 알지 못함
     - 단, 성능 저하를 감당할 수 없거나 네이티브 피어의 자원을 즉시 회수해야 한다면 close 메소드를 사용해야 함

> ⚠ cleaner(자바 8까지는 finalizer)는 안전망 역할이나 중요하지 않은 네이티브 자원 회수용으로만 사용하자. 물론 이런 경우라도 불확실성과 성능 저하에 주의해야 한다

▼ [아이템 9] try-finally보다는 try-with-resources를 사용하라

try-with-resources

- AutoClosable을 구현한 객체에 한해 close 메소드를 자동으로 호출해 준다.
- 짧고 가독성 좋을 뿐 아니라 문제를 진단하기에도 훨씬 좋다
- 꼭 회수해야 하는 자원을 다룰 때에는 try-finally 말고 try-with-resources를 사용하도록 하자.
    - 예외는 없다!