2장 객체 생성과 파괴

작성자 : 전우성

2021.09.12

아이템 1. 생성자 대신 정적 팩터리 메서드를 고려하라

아이템1. 생성자 대신 정적 팩터리 메서드를 고려하라

1. 클래스 인스턴스를 얻는 수단
- public
- 정적 팩터리 메서드(static factory method)

2. 장점

- 이름을 가질 수 있다.

  -> 유지보수, 사용성 증가

- 호출될 때마다 인스턴스를 새로 생성하지는 않아도 된다.

  -> 불필요한 객체 생성을 피하고, 재활용성 증가. 자주 사용하는 불변하는 인스턴스에 해당.

- 반환 타입의 하위 타입 객체를 반환할 수 있는 능력이 있다.

  -> API 유연함. API를 사용하는 입장에서 자세히 분석하지 않아도 된다. (약속된 결과)

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

  -> 입력 매개변수의 상태에 따라 내부 로직을 거쳐 알맞은 객체를 얻을 수 있다.

- 정적 패터리 메서드를 작성하는 시점에는 반환할 객체의 클래스가 존재하지 않아도 된다.

  -> 반환할 객체의 클래스 PATH만 정해둔채로 개발을 할 수 있다.

3. 단점

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

  -> 프레임워크 유틸리티 구현 클래스들은 상속할 수 없음.

- 정적 팩터리 메서드는 프로그래머가 찾기 어렵다.

  -> 말그대로 찾기가 어렵기 때문에 API 문서, doc 활용이 및 명칭에 신경써야한다.

아이템 2. 생성자에 매개변수가 많다면 빌더를 고려하라

1. 점층적 생성자 패턴 (확장하기 어렵다)

```
public class NutritionFacts {
    private final int servingSize;  // (ml, 1회 제공량)        필수
    private final int servings;     // (회, 총 n회 제공량)     필수
    private final int calories;     // (1회 제공량당)          선택
    private final int fat;          // (g/1회 제공량)          선택
    private final int sodium;       // (mg/1회 제공량)         선택
    private final int carbohydrate; // (g/1회 제공량)          선택

    public NutritionFacts(int servingSize, int servings) {
        this(servingSize, servings, 0);
    }

    public NutritionFacts(int servingSize, int servings, int calories) {
        this(servingSize, servings, calories, 0);
    }

    public NutritionFacts(int servingSize, int servings,int calories,
                          int fat) {
        this(servingSize, servings, calories, fat, 0);
    }

    public NutritionFacts(int servingSize, int servings, int calories,
                          int fat, int sodium) {
        this(servingSize, servings, calories, fat, sodium, 0);
    }

    public NutritionFacts(int servingSize, int servings, int calories,
                          int fat, int sodium, int carbohydrate) {
        this.servingSize  = servingSize;
        this.servings     = servings;
        this.calories     = calories;
        this.fat          = fat;
        this.sodium       = sodium;
        this.carbohydrate = carbohydrate;
    }
}
```

2. 자바빈즈 패턴 (일관성이 깨지고, 불변으로 반들 수 없다)

```
public class NutritionFacts {
    // 매개변수들은 (기본값이 있다면) 기본값으로 초기화된다.
    private int servingSize  = -1; // 필수; 기본값 없음
    private int servings     = -1; // 필수; 기본값 없음
    private int calories     = 0;
    private int fat          = 0;
    private int sodium       = 0;
    private int carbohydrate = 0;

    public NutritionFacts() { }
    // 세터 메서드들
    public void setServingSize(int val)  { servingSize = val; }
    public void setServings(int val)     { servings = val; }
    public void setCalories(int val)     { calories = val; }
    public void setFat(int val)          { fat = val; }
    public void setSodium(int val)       { sodium = val; }
    public void setCarbohydrate(int val) { carbohydrate = val; }
}
```

```
NutritionFacts cocaCola = new NutritionFacts();
cocaCola.setServingSize(240);
cocaCola.setServings(8);
cocaCola.setCalories(100);
cocaCola.setSodium(35);
cocaCola.setCarbohydrate(27);
```

객체의 완전 생성까지 일관성이 없다.

3. 빌더 패턴

```
public class NutritionFacts {
    private final int servingSize;
    private final int servings;
    private final int calories;
    private final int fat;
    private final int sodium;
    private final int carbohydrate;

    public static class Builder {
        // 필수 매개변수
        private final int servingSize;
        private final int servings;

        // 선택 매개변수 - 기본값으로 초기화한다.
        private int calories      = 0;
        private int fat           = 0;
        private int sodium        = 0;
        private int carbohydrate  = 0;

        public Builder(int servingSize, int servings) {
            this.servingSize = servingSize;
            this.servings    = servings;
        }

        public Builder calories(int val)
            { calories = val;      return this; }
        public Builder fat(int val)
            { fat = val;           return this; }
        public Builder sodium(int val)
            { sodium = val;        return this; }
        public Builder carbohydrate(int val)
            { carbohydrate = val;  return this; }

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

    private NutritionFacts(Builder builder) {
        servingSize  = builder.servingSize;
        servings     = builder.servings;
        calories     = builder.calories;
        fat          = builder.fat;
        sodium       = builder.sodium;
        carbohydrate = builder.carbohydrate;
    }
}
```

```
NutritionFacts cocaCola = new NutritionFacts.Builder(240, 8)
        .calories(100).sodium(35).carbohydrate(27).build();
```

아이템 3. private 생성자나 열거 타입으로 싱글턴임을 보증하라

아이템3. private 생성자나 열거 타입으로 싱글턴임을 보증하라

코드 3-1 public static final 필드 방식의 싱글턴

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

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

싱글턴임이 API에 드러남.
간결함.
역직렬화 주의.

코드 3-2 정적 팩터리 방식의 싱글턴

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

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

API를 바꾸지 않고도 싱글턴이 아니게 변경 가능.
역직렬화 주의.

```
// 싱글턴임을 보장해주는 readResolve 메서드
private Object readResolve() {
    // '진짜' Elvis를 반환하고, 가짜 Elvis는 가비지 컬렉터에 맡긴다.
    return INSTANCE;
}
```

코드 3-3 열거 타입 방식의 싱글턴 - 바람직한 방법

```
public enum Elvis {
    INSTANCE;

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

단순함.
직렬화, 리플렉션 공격 완전 방지.
Enum 이외의 다른 상위 클래스를 상속해야 한다면 사용할 수 없다.

아이템 4. 인스턴스화를 막으려거든 private 생성자를 사용하라

아이템4. 인스턴스화를 막으려거든 private 생성자를 사용하라

- java.util.Arrays, java.util.Collections 와 같은 정적 메서드(팩터리)
- private ClassName() {//TODO};
- 유틸리티 목적성 클래스는 인스턴스화가 필요없으므로 명시적으로 private 생성자를 생성한다.

아이템 5. 자원을 직접 명시하지 말고 의존 객체 주입을 사용하라

아이템5. 자원을 직접 명시하지 말고 의존 객체 주입을 사용하라

- 인스턴스 생성 시 필요한 자원을 파라미터로 받도록 함.
- 여러 프레임워크에서 사용 중.
- 유연성, 사용성 증가.

코드 5-3 의존 객체 주입은 유연성과 테스트 용이성을 높여준다.

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

    public SpellChecker(Lexicon dictionary) {
        this.dictionary = Objects.requireNonNull(dictionary);
    }

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

아이템 6. 불필요한 객체 생성을 피하라

- 안좋은 예

```
String s = new String("bikini"); // 따라 하지 말 것!
```

코드 6-1 성능을 훨씬 더 끌어올릴 수 있다!

```
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})$");
}
```

- 개선

```
String s = "bikini";
```

코드 6-2 값비싼 객체를 재사용해 성능을 개선한다.

```
public class RomanNumerals {
    private static final Pattern ROMAN = Pattern.compile(
            "^(?=.)M*(C[MD]|D?C{0,3})"
            + "(X[CL]|L?X{0,3})(I[XV]|V?I{0,3})$");

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

아이템6. 불필요한 객체 생성을 피하라

- 안좋은 예

코드 6-3 끔찍이 느리다! 객체가 만들어지는 위치를 찾았는가?

```
private static long sum() {
    Long sum = 0L;
    for (long i = 0; i <= Integer.MAX_VALUE; i++)
        sum += i;

    return sum;
}
```

- 개선

```
private static long sum() {
    long  sum = 0L;
    for (long i = 0; i <= Integer.MAX_VALUE; i++)
        sum += i;

    return sum;
}
```

아이템 7. 다 쓴 객체 참조를 해제하라

- 다 사용하였으면, null 처리

코드 7-2 제대로 구현한 pop 메서드

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

아이템7. 다 쓴 객체 참조를 해제하라

```java
package com.ktko.init;

import java.lang.ref.SoftReference;
import java.lang.ref.WeakReference;
import java.util.LinkedList;
import java.util.List;

// PhantomReference

class BigData {
    private int[] array = new int[2500]; //10000byte, 10K
}

public class ReferenceTest {
    private List<WeakReference<BigData>> weakRefs = new LinkedList<>();
    private List<SoftReference<BigData>> softRefs = new LinkedList<>();
    private List<BigData> strongRefs = new LinkedList<>();


    public void weakReferenceTest() {
        try {
            for (int i = 0; true; i++) {
                weakRefs.add(new WeakReference<BigData>(new BigData()));
            }
        } catch (OutOfMemoryError ofm) { // weak일 경우 out of memory 발생 하지 않는다.
            System.out.println("out of memory!");
        }
    }

    public void softReferenceTest() {
        try {
            for (int i = 0; true; i++) {
                softRefs.add(new SoftReference<BigData>(new BigData()));
            }
        } catch (OutOfMemoryError ofm) { // weak일 경우 out of memory 발생 하지 않는다.
            System.out.println("out of memory!");
        }
    }


    public void strongReferenceTest() {
        try {
            for (int i = 0; true; i++) {
                strongRefs.add(new BigData());
            }
        } catch (OutOfMemoryError ofm) { // Strong일 경우 out of memory 발생
            System.out.println("out of memory!");
        }
    }


    public static void main(String[] args) {
        System.out.println("실행");

        ReferenceTest test = new ReferenceTest2();
        test.weakReferenceTest();
        //test.softReferenceTest();
        //test.strongReferenceTest();

        System.out.println("종료");
    }
}
```

아이템 8. finalizer와 cleaner 사용을 피하라

아이템8. finalizer와 cleaner 사용을 피하라

- 객체 소멸자 finalizer, cleaner
- c++ 파괴자와는 다른 개념
- finalizer와 cleaner로는 제때 실행되어야 하는 작업은 절대 할 수 없다.

- 대안 - AutoCloseable : close()

핵심 정리

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

아이템9. try-finally보다는 try-with-resources를 사용하라

- close 메서드를 호출해 직접 닫아줘야 하는 자원
- InputStream, OutputStream, java.sql.Connection
- 일반적인 try-finally -> 자바7 try-with-resources 사용
- AutoCloseable interface

코드 9-3 try-with-resources - 자원을 회수하는 최선책!

```
static String firstLineOfFile(String path) throws IOException {
    try (BufferedReader br = new BufferedReader(
            new FileReader(path))) {
        return br.readLine();
```

코드 9-4 복수의 자원을 처리하는 try-with-resources - 짧고 매혹적이다!

```
static void copy(String src, String dst) throws IOException {
    try (InputStream in = new FileInputStream(src);
         OutputStream out = new FileOutputStream(dst)) {
        byte[] buf = new byte[BUFFER_SIZE];
        int n;
        while ((n = in.read(buf)) >= 0)
            out.write(buf, 0, n);
    }
}
```

코드 9-5 try-with-resources를 catch 절과 함께 쓰는 모습

```
static String firstLineOfFile(String path, String defaultVal) {
    try (BufferedReader br = new BufferedReader(
            new FileReader(path))) {
        return br.readLine();
    } catch (IOException e) {
        return defaultVal;
    }
}
```

핵심 정리

꼭 회수해야 하는 자원을 다룰 때는 try-finall 말고, try-with-resources를 사용하자. 예외는 없다. 코드는 더 짧고 분명해지고, 만들어지는 예외 정보도 훨씬 유용하다. try-finally로 작성하면 실용적이지 못할 만큼 코드가 지저분해지는 경우라도, try-with-resources로는 정확하고 쉽게 자원을 회수할 수 있다.

summary

1. 정적 팩터리 메서드
2. 생성자에 매개변수가 많다면 빌더패턴
3. 열거 타입의 싱글턴
4. 유틸성 클래스는 명시적으로 private 생성자를 작성하여 인스턴스화 방지
5. 불필요한 객체생성을 피하기
6. 다 쓴 객체의 참조 해제
7. finalizer, cleaner 사용 피하기
8. try-finally 보다는 try-with-resources 사용