Ch2. 객체 생성과 파괴

태그	1주차

📌 목차

Item1 생성자 대신 정적 팩터리 메서드를 고려하라

직접적으로 생성자를 통해 객체를 생성하는 것이 아닌 메서드를 통해서 객체를 생성하는 것을 **정적 팩토리 메서드**라고 한다.

정적 팩터리 메서드를 사용할 때의 장점

1. **이름을 가질 수 있다. & 여러 경우의 생성자를 만들기 용이하다.**

생성자는 클래스 이름과 동일한 이름을 가져야한다. 또한 다른 기능의 생성자를 만드려면 입력 매개변수의 순서를 다르게 한 생성자를 추가하는 식의 방법을 써야한다. 매개변수로 해당 생성자가 어떤 기능을 하는지 표현하기 어렵다는 단점이 있다. 정적 팩터리 메서드를 통해 이름을 부여하여 이를 극복할 수 있다.

아래 예시를 보자.

```
public class LottoFactory() {
  private static final int LOTTO_SIZE = 6;

  private static List<LottoNumber> allLottoNumbers = ...; //

  public static Lotto createAutoLotto() {
    Collections.shuffle(allLottoNumbers);
    return new Lotto(allLottoNumbers.stream()
            .limit(LOTTO_SIZE)
            .collect(Collectors.toList()));
  }

  public static Lotto createManualLotto(List<LottoNumber> lot
    return new Lotto(lottoNumbers);
  }
  ...
}
```

정적 팩터리 메서드를 통해 이름을 부여하여, 어떤 객체를 반환하는지 명확히 할 수 있게 되었다.

2. 호출될 때마다 인스턴스를 새로 생성하지 않아도 된다.

`불변클래스` 또는 `Enum클래스`는 인스턴스를 미리 만들어 놓거나 새로 생성한 인스턴스를 캐싱하여 재활용하는 식으로 불필요한 객체 생성을 피할 수 있다.

이와 같이 같은 객체를 반환하게 하여 인스턴스를 통제하면 `싱글톤`, `인스턴스화불가`로 만들 수 있다는 장점이 생긴다.

3. 반환 타입의 하위 타입 객체를 반환할 수 있는 능력이 있다.

`Basic`, `Intermediate`, `Advanced` 클래스가 `Level`라는 상위 타입을 상속받고 있는 구조라 생각해보자.

```
public class Level {
  ...
  public static Level of(int score) {
    if (score < 50) {
      return new Basic();
    } else if (score < 80) {
      return new Intermediate();
    } else {
      return new Advanced();
    }
  }
  ...
}
```

위 예시처럼 반환할 객체의 클래스를 자유롭게 선택할 수 있게하는 유연성을 가질 수 있다.

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

3번의 맥락과 비슷하게 매개변수에 따라 다른 클래스를 반환할 수 있게 된다.

5. 정적 팩터리 메서드를 작성하는 시점에는 반환할 객체의 클래스가 존재하지 않아도 된다.

가장 대표적인 사례가 JDBC!

- `DrivaerManager.registerDriver`가 등록 API역할
- `DrivaerManager.get`이 서비스 접근 API 역할

- `Driver`가 서비스 제공자 인터페이스 역할

이처럼 작성 시점에 해당 클래스가 존재하지 않아도 된다. 이를 서비스 제공자 프레임워크 패턴이라 한다. Spring에서 쓰이는 의존 객체 주입(DI)는 이 패턴이 변형된 것이다.

정적 팩터리 메서드를 사용할 때의 단점

1. **상속을 하려면 public, protected 생성자가 필요하니 정적 팩토리 메서드만 제공하면 하위 클래스를 만들 수 없다.**
2. **정적 팩터리 메서드는 프로그래머가 찾기 어렵다.**

이를 극복하기 위해 정적 팩터리 메서드도 네이밍 컨벤션이 존재한다.

정적 팩토리 메서드 네이밍 컨벤션

- `from` : 하나의 매개 변수를 받아서 해당 타입의 객체를 생성
  - Data d = Date.from(instant);
- `of` : 여러개의 매개 변수를 받아서 적합한 타입의 객체를 생성
  - Set<Rank> faceCards = EnumSet.of(JACK, QUEEN, KING);
- `valueOf` : from과 of의 더 자세한 버전 (범용적으로 쓰이는듯)
  - BigInteger prime = BigInteger.valueOf(Integer.MAX_VALUE);
- `getInstance` | `instance` : 인스턴스를 반환. 이전에 반환했던 것과 같을 수 있음.
  - StackWalker luke = StackWalker.getInstance(options);
- `newInstance` | `create` : 매번 새로운 인스턴스를 생성
  - Object newArray = Array.newInstance(classObject, arrayLen);
- `getType` : 다른 타입의 인스턴스를 생성. 이전에 반환했던 것과 같을 수 있음.
  - FileStore fs = Files.getFileStore(path);
- `newType` : 다른 타입의 새로운 인스턴스를 생성.
  - BufferedReader br = Files.newBufferedReader(path);

⭐핵심정리

핵심 정리

정적 팩터리 메서드와 public 생성자는 각자의 쓰임새가 있으니 상대적인 장단점을 이해하고 사용하는 것이 좋다. 그렇다고 하더라도 정적 팩터리를 사용하는 게 유리한 경우가 더 많으므로 무작정 public 생성자를 제공하던 습관이 있다면 고치자.

Item2 생성자에 매개변수가 많다면 빌더를 고려하라

매개변수가 많은 경우 빌더 패턴을 이용하자. (4개 이상은 되어야 값어치를 한다.)

- 점층적생성자: 매개변수의 개수를 다르게 한 생성자들을 전부 만들어 필요한 것을 사용 → 가독성이 떨어짐
- 자바빈즈 패턴: 일반적인 setter로 값 설정 → 안정성이 떨어짐

코드로 편의성을 확인해보자

예시 코드

```
// 코드 2-3 빌더 패턴 - 점층적 생성자 패턴과 자바빈즈 패턴의 장점만 취했다
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;
    }

        public static void main(String[] args) {
        NutritionFacts cocaCola = new NutritionFacts.Builder(
                .calories(100).sodium(35).carbohydrate(27).bu
    }
```

⭐핵심정리

핵심 정리

생성자나 정적 팩터리가 처리해야 할 매개변수가 많다면 빌더 패턴을 선택하는 게 더 낫다. 매개변수 중 다수가 필수가 아니거나 같은 타입이면 특히 더 그렇다. 빌더는 점층적 생성자보다 클라이언트 코드를 읽고 쓰기가 훨씬 간결하고, 자바빈즈보다 훨씬 안전하다.

Item3 Private 생성자나 열거 타입으로 싱글톤임을 보장하라

💡 싱글톤이란 인스턴스를 오직 하나만 생성할 수 있는 클래스를 말한다.

일반적으로 내부 생성자를 `Private`으로 선언하고, 정적 팩터리 메서드를 통해 미리 생성해 둔 인스턴스를 반환하는 식으로 설계한다.

예시 코드

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

    // 이 메서드는 보통 클래스 바깥(다른 클래스)에 작성해야 한다!
    public static void main(String[] args) {
        Elvis elvis = Elvis.getInstance();
    }
}
```

Item4 인스턴스화를 막으려거든 Private 생성자를 사용하라

단순히 정적 메서드와 정적 필드만을 담은 클래스를 만들고 싶을 때 사용한다.

`java.lang.Math`, `java.util.Arrays`, `java.util.Collections` 가 그 예시이다.

생성자를 명시하지 않으면 컴파일러가 자동으로 기본 생성자를 만들기 때문에, `Private` 생성자를 추가하여 클래스의 인스턴스화를 막자!

예시코드

```
// 코드 4-1 인스턴스를 만들 수 없는 유틸리티 클래스 (26~27쪽)
public class UtilityClass {
    // 기본 생성자가 만들어지는 것을 막는다(인스턴스화 방지용).
    private UtilityClass() {
        throw new AssertionError();
    }

    // 나머지 코드는 생략
}
```

Item5 자원을 직접 명시하지 말고 의존 객체 주입을 사용하라

어떤 클래스가 여러 자원에 의존할때(자원에 따라 동작이 달라지는 클래스인 경우) 자원을 직접 명시하는 것이 아닌, 인스턴스를 생성할 때 생성자에 필요한 자원을 넘겨주는 방식으로 설계해보자.

예시코드

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

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

`SpellChecker`를 생성하면서 `dictionary`를 주입해줄 수 있으며, final로 선언되어 있기 때문에, 주입 후 값이 바뀌지 않는다. 이처럼 특정 자원에 의존하지 않고, 생성 시점에 객체를 주입해주는 방식을 사용하자. (DI)

⭐핵심 정리

> **핵심 정리**
>
> 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 안 된다. 대신 필요한 자원을 (혹은 그 자원을 만들어주는 팩터리를) 생성자에 (혹은 정적 팩터리나 빌더에) 넘겨주자. 의존 객체 주입이라 하는 이 기법은 클래스의 유연성, 재사용성, 테스트 용이성을 기막히게 개선해준다.

Item6 불필요한 객체 생성을 피하라

같은 기능의 객체를 재사용하자

똑같은 기능의 객체를 매번 생성하는것보다 객체 하나를 재사용하는 편이 나을 때가 많다.

```
String s = new String("bikini");
String s = "bikini";
```

자바에서는 하나의 `String` 객체를 만들어두면 이를 다음에 호출하면 같은 String 객체를 호출하게 된다. 불필요하게 `String` 객체를 여러개 만들지 말자.

생성 비용이 비싼 객체가 있다면 캐싱한 후 재사용하자.

생성 비용이 비싼 객체가 반복해서 사용하게 된다면 캐싱해서 재사용하는 걸 권장한다.

```
// 이렇게 하면 더 느리다.
static boolean isRomanNumeralSlow(String s) {
    return s.matches("^(?=.)M*(C[MD]|D?C{0,3})"
            + "(X[CL]|L?X{0,3})(I[XV]|V?I{0,3})$");
}

// 미리 컴파일 해놓은 객체를 불러 사용하는 방식으로 개선했다
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 isRomanNumeralFast(String s) {
    return ROMAN.matcher(s).matches();
}
```

첫번째 경우 Pattern 인스턴스는 한 번 쓰고 버려져서 곧바로 가비지 컬렉션 대상이 된다. Pattern은 생성비용이 비싼 객체이기 때문에, 두번째 경우와 같이 미리 캐싱해두고 사용해야한다.

오토박싱을 통한 객체 생성을 유의하자

> 💡 **오토박싱**: 기본 타입(primitive type)을 해당 래퍼 클래스(wrapper class) 객체로 변환하는 과정.
>
> **언박싱**: 래퍼 클래스 객체를 기본 타입으로 변환하는 과정.

```
public class AutoBoxingExample {
    public static void main(String[] args) {
        long start = System.currentTimeMillis();
           Long sum = 0l; //문제가 발생하는 부분
        for (long i = 0 ; i <= Integer.MAX_VALUE ; i++) {
            sum += i;
        }
        System.out.println(sum);
        System.out.println(System.currentTimeMillis() - start
    }
}
```

`sum`이 Long 래퍼 클래스의 객체이기 때문에 `sum += i` 를 실행할 때, Java는 `long` 타입의 `i`를 자동으로 `Long` 객체로 변환(오토박싱)하고, 이 결과를 다시 `long` 타입으로 변환(언박싱)한 후, 더하기 연산을 수행하고, 그 결과를 다시 `Long` 객체로 변환(오토박싱)한다.

즉 불필요한 Long 인스턴스가 2^31개 만들어 지는 것이다. 실행시간도 6.3초와 0.59초로 상당히 차이가 난다.

의도치 않은 오토박싱이 숨어들지 않도록 주의하자.

객체 생성에 대한 오해

객체생성은 비싸니 피해야 한다로 오해하면 안된다. 특히 JVM에서는 별다른 일을 하지 않는 작은 객체에 대해서는 큰 부담이 되지 않는다고 한다

프로그램의 명확성, 간결성, 기능을 위해서는 객체를 추가로 만들 수도 있어야 한다

또 객체 생성을 효율적으로 해보겠다고 사소한 것들도 다 캐싱하거나 자체 풀(pool)을 만들어서 유지보수하기 어려운 복잡한 프로그램을 만드는 것도 피해야 하는 부분이다 (JVM에게 위임할 부분은 위임해라 가비지 컬렉터를 신뢰하자)

[Item50]방어적 복사(defensive copy) 와 대비되는 내용이기 때문에 객체 생성을 해야하는 경우와 하지 않고 기존 것을 재사용해야 하는 부분은 개발자의 역량에 달려있다 (혹은 성능 테스트를 직접 해서 비교해보아라)

새로 만든 생성자가 필요한 경우와 재사용 가능한 불변 객체를 사용하는 구분을 할 수 있어야 한다

Item7 다 쓴 객체 참조를 해제하라

메모리 누수가 발생하는 예시상황

```
public class Stack {
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_CAPACITY = 16;
    public Stack() {
        elements = new Object[DEFAULT_CAPACITY];
    }
    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }
    public Object pop() {
        if (size == 0) throw new EmptyStackException();
        return elements[--size];
    }
    private void ensureCapacity() { // 원소들이 들어갈 공간 확보
        if (elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
```

```
    }
}
```

위 코드에서 문제점은 스택에서 pop을 할때 인덱스만 이동하고 참조를 해지하지 않아서 발생한다. 이렇게 스택에 남아있는 값들은 가비지컬렉션이 처리하지 못하므로 메모리 누수가 발생할 수 있다.

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

null 값으로 변경하여 다쓴 객체를 해지해주자!

메모리 누수가 발생하는 주요 상황들

1. 자신의 메모리를 직접 관리하는 클래스(위 예시의 Stack)
    a. 원소를 다 사용한 즉시 그 원소가 참조한 객체들을 다 `null` 처리해줘야 한다.
2. 캐시: 객체 참조를 캐시에 넣고 객체를 다 쓴 뒤로도 한참을 놔두는 경우
    a. 캐시 외부에서 `key`를 참조하는 동안만 엔트리가 살아있는 캐시가 필요한 상황이면 `WaekHachMap`을 사용한다.
    b. 주기적으로 또는 어떤 이벤트마다 엔트리를 청소해주는 방법 → **백그라운드 쓰레드를 사용하거나 캐시에 새 엔트리가 등록될 때 제거하는 방법**이 있다. `LinkedHashMap`은 `removeEldeestEntry` 메소드를 사용하여 후자의 방식으로 처리하게 된다.
3. 리스너(listener) or 콜백(callback) : 클라이언트가 콜백을 등록만하고 해지하지 않는다면 콜백은 계속 쌓인다.
    a. 콜백을 약한 참조(weak reference)로 저장하면 컬렉터가 즉시 수거해갑니다.

⭐핵심정리

> **핵심 정리**
>
> 메모리 누수는 겉으로 잘 드러나지 않아 시스템에 수년간 잠복하는 사례도 있다. 이런 누수는 철저한 코드 리뷰나 힙 프로파일러 같은 디버깅 도구를 동원해야만 발견되기도 한다. 그래서 이런 종류의 문제는 예방법을 익혀두는 것이 매우 중요하다.

Item8 finalizer와 cleaner 사용을 피하라

자바는 두 가지 객체 소멸자인 finalizer와 cleaner를 제공한다. 이는 C와 C++에서 사용하는 `destructor`와는 다른 개념이다. 자바에서는 `destructor` 의 역할은 gc가 담당한다.

finalizer와 cleaner를 사용하지 말아야하는 이유

1. finalizer와 cleaner는 실행시점을 정할 수 없다. 개발자가 원하는 방식으로 프로그램을 설계할 수 없다. 가장 확실한 방법은 그냥 사용하지 않는 것이다.
2. finalizer와 cleaner는 심각한 성능 문제도 동반한다. finalizer가 gc의 효율을 떨어트리기 때문이다.
3. finalizer를 사용한 클래스는 finalizer 공격에 노출되어 심각한 보안 문제를 일으킬 수도 있다.

위와 같은 문제점들을 해결하기 위해서는 `AutoCloseable`을 구현해주고 `try-with-resources`를 사용하자!

⭐핵심 정리

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

Item9 try-finally 보다는 try-with-resources를 사용하라

try-with-resources를 사용해야하는 이유

자바 라이브러리에는 close 메서드를 호출해 직접 닫아줘야 하는 자원이 많다.

ex) `InputStream`, `OutputStream`, `java.sql.Connection`

`try-finally`를 사용해 `finally` 구문에서 직접 닫아줄 수도 있지만, 닫아줘야하는 자원이 2개 이상인 경우 코드의 가독성이 상당히 떨어진다.

```
// 코드 9-2 자원이 둘 이상이면 try-finally 방식은 너무 지저분하다! (4
    static void copy(String src, String dst) throws IOExcepti
        InputStream in = new FileInputStream(src);
        try {
            OutputStream out = new FileOutputStream(dst);
            try {
                byte[] buf = new byte[BUFFER_SIZE];
                int n;
                while ((n = in.read(buf)) >= 0)
                    out.write(buf, 0, n);
            } finally {
                out.close();
            }
        } finally {
            in.close();
        }
    }
```

프로그래머가 실수를 할 확률도 당연히 높아진다!

`AutoCloseable` 인터페이스를 구현하여 `try-with-resources`를 이용하자!

```
// 코드 9-4 복수의 자원을 처리하는 try-with-resources - 짧고 매혹적이
    static void copy(String src, String dst) throws IOExcepti
        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);
```

```
        }
    }
```

직접 닫아주지 않아도 `Autocloseble` 에 의해 닫히므로 훨씬 실용적이다.

⭐핵심 정리

> **핵심 정리**
>
> 꼭 회수해야 하는 자원을 다룰 때는 try-finally 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.