item 76

가능한 한 실패 원자적으로 만들라

유현우

실패 원자적이란?

호출된 메서드가 실패하더라도

해당 객체는 메서드 호출 전 상태를 유지해야 한다.

불변 객체의 메서드를 실패 원자적으로 만드는 방법

불변 객체는 생성 시점에 고정되어 절대 변하지 않기 때문에
기존 객체가 불안정한 상태에 빠지는 일은 없다.

불변 객체의 메서드를 실패 원자적으로 만드는 방법

```java
@ToString
class Name {
  private final String name;
  public Name(String name) {
    if ("admin".equals(name)) {
      throw new IllegalArgumentException("사용할 수 없는 이름입니다.");
    }
    this.name = name;
  }
}
```

개인적으로 지역 변수 final을 선호하는 이유

```java
public static void main(String[] args) {
    Name name = new Name("test");
    try {
      name = new Name("admin");
    } catch (Exception e) {
      System.out.println(name);
    }
  }
```

```
> Task :Main.main()
Name(name=test)
```

불변 객체를 이렇게 사용하는 것도 올바르지 않음(예제)

가변 객체의 메서드를 실패 원자적으로 만드는 방법

```java
@ToString
class User {
  private String name;

  public void setName(String name) {
    if ("admin".equals(name)) {
      throw new IllegalArgumentException("사용할 수 없는 이름입니다.");
    }
    this.name = name;
  }

  public User(String name) {
    this.name = name;
  }
}
```

```java
public static void main(String[] args) {
    User user = new User("test");
    try {
      user.setName("admin");
    } catch (Exception e) {
      System.out.println(user);
    }
  }
```

```
> Task :Main.main()
User(name=test)
```

작업 수행에 앞서 매개변수의 유효성을 검사하는 방법

로직을 수행하기 전에 매개변수의 유효성을 검사한다.

Stack.pop

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

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

조건이 없어도 스택이 비어있으면 여전히 예외를 던진다.

(ArrayIndexOutOfBoundsException)

로직을 수행하기 전에 매개변수의 유효성을 검사한다.

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

size가 음수가 되어 다음 호출도 결국 실패하기 때문에 추상화 수준이 상황에 어울리지 않다.
(ArrayIndexOutOfBoundsException)

실패할 가능성이 있는 모든 코드를 객체의 상태를 바꾸는 코드보다 앞에 배치한다.

TreeMap.put

```
private V put(K key, V value, boolean replaceOld) {
  Entry<K, V> t = root;
  if (t == null) {
    addEntryToEmptyMap(key, value);
    return null;
  }
  int cmp;
  Entry<K, V> parent;
  // split comparator and comparable paths
  Comparator<? super K> cpr = comparator;
  if (cpr != null) {
    do {
      parent = t;
      cmp = cpr.compare(key, t.key);
      if (cmp < 0)
        t = t.left;
      else if (cmp > 0)
```

```
public static void main(String[] args) {
    TreeMap<Object, Integer> treeMap = new TreeMap<>();
    treeMap.put("1", 1);
    treeMap.put(1, 1);
  }
```

TreeMap에 원소를 추가할 때 Key값을 비교하면서 원소가 있어야 하는 위치를 변경할 것이다.

실패할 가능성이 있는 모든 코드를 객체의 상태를 바꾸는 코드보다 앞에 배치한다.

TreeMap.put

```
private V put(K key, V value, boolean replaceOld) {
  Entry<K, V> t = root;
  if (t == null) {
    addEntryToEmptyMap(key, value);
    return null;
  }
  int cmp;
  Entry<K, V> parent;
  // split comparator and comparable paths
  Comparator<? super K> cpr = comparator;
  if (cpr != null) {
    do {
      parent = t;
      cmp = cpr.compare(key, t.key);
      if (cmp < 0)
        t = t.left;
      else if (cmp > 0)
```

```
Compares two keys using the correct comparison method for this TreeMap.
/unchecked/
final int compare(Object k1, Object k2) {
  return comparator == null ? ((Comparable<? super K>) k1).compareTo((K) k2)
      : comparator.compare((K) k1, (K) k2);
}
```

```
Exception in thread "main" java.lang.ClassCastException Create breakpoint : class java.lang.String cannot be cast to class
  java.lang.Integer (java.lang.String and java.lang.Integer are in module java.base of loader 'bootstrap')
  at java.base/java.lang.Integer.compareTo(Integer.java:71)
  at java.base/java.util.TreeMap.put(TreeMap.java:814)
  at java.base/java.util.TreeMap.put(TreeMap.java:534)
  at attraction.yong.exam.Main.main(Main.java:10)
```

TreeMap에 원소를 추가할 때 해당 원소가 들어갈 위치를 찾는 과정에서 ClassCastException 발생

객체의 임시 복사본에서 작업을 수행한 후에 성공적으로 완료되면 원래 객체와 교체한다.

- 데이터를 임시 자료 구조에 저장해 작업하는 것이 더 빠를 때 적용하기 좋은 방법이다.

List.sort

```
default void sort(Comparator<? super E> c) {
    Object[] a = this.toArray();
    Arrays.sort(a, (Comparator) c);
    ListIterator<E> i = this.listIterator();
    for (Object e : a) {
        i.next();
        i.set((E) e);
    }
}
```

정렬이 실패하더라도 복사본을 통해서 작업했기 때문에 원본 리스트는 변하지 않는다.

작업 도중에 발생하는 실패를 가로채는 복구 코드를 작성하여 작업 전 상태로 되돌린다.

주로 디스크 기반의 내구성(durability)를 보장해야 하는 자료구조에 쓰이는데 자주 사용되는 방법은 아니다.

실패 원자성을 무조건 지켜야 할까?

- 실패 원자적으로 만들 수 있어도 항상 그래야 하는 것도 아니다.

달성하기 위한 비용이 크거나 복잡도가 아주 큰 연산이 있을 수 있기 때문이다.

- 이 규칙을 지키지 못한다면 실패 시의 객체 상태를 API 설명에 명시해야 한다.

Error는 복구할 수 없으므로 AssertionError에 대해서는 실패 원자적으로는
만들려는 시도도 필요가 없다.