


이펙티브 자바 CP.9

| | |
|------------|--|
| 🕒 작성 일시 | @2023년 3월 16일 오후 11:42 |
| 🕒 최종 편집 일시 | @2023년 3월 20일 오후 10:06 |
| 📄 유형 | 이펙티브 자바 |
| 👤 작성자 |  종현 박 |
| 👥 참석자 | |
| 🗣️ 언어 | |

9 예외

- 69. 예외는 진짜 예외 상황에만 사용하라.
- 70. 복구할 수 있는 상황에는 검사 예외를, 프로그래밍 오류에는 런타임 예외를 사용하라
- 71. 필요 없는 검사 예외 사용은 피해라
- 72. 표준 예외를 사용하라
- 73. 추상화 수준에 맞는 예외를 던져라
- 74. 메서드가 던지는 모든 예외를 문서화하라
- 75. 예외의 상세 메시지에 실패 관련 정보를 담으라
- 76. 가능한 한 실패 원자적으로 만들라
- 77. 예외를 무시하지 말라

9 예외

▼ 69. 예외는 진짜 예외 상황에만 사용하라.

- 예외
 - 제대로 사용한다면 프로그램의 가독성, 신뢰성, 유지보수성이 높아진다
 - 하지만 반대로, 잘못 사용하면 그 반대의 효과를 보게 된다.
- ex) 예외를 잘 못 사용한 예

```
try {
    int i = 0;
    while(true)
        range[i++].climb();
}
```

```

} catch (ArrayIndexOutOfBoundsException e) {
}

```

- 전혀 직관적이지 않다는 사실 하나 만으로도 코드를 이렇게 작성하면 안된다. (아이템 67)
- 무한 루프를 돌다가 배열의 끝에 도달해 예외가 발생하면 끝을 내는 것이다...
- 표준적인 관용구 표현으로는


```
for (Mountain m : range) m.climb();
```

 이다.
- 예외를 써서 루프를 종료한 이유
 - JVM 은 배열에 접근할 때마다, 경계를 넘지 않는지 검사하는데, 일반적인 반복 문도 배열 경계에 도달하면 종료한다. 따라서 이 검사를 반복문에도 명시하면 같은 일이 중복됨으로 하나를 생략한 것이다.
 - 하지만 이는 3 가지 면에서 잘못된 추론이다.
 1. 예외는 예외 상황에 쓸 용도로 설계되었으므로 JVM 구현자 입장에서는 명확한 검사만큼 빠르게 만들어야 할 동기가 약하다. (최적화에 별로 신경 쓰지 않았을 가능성이 크다)
 2. 코드를 `try-catch` 블록 안에 넣으면 JVM이 적용할 수 있는 최적화가 제한된다.
 3. 배열을 순회하는 표준 관용구는 앞서 걱정한 중복 검사를 수행하지 않는다. JVM이 알아서 최적화해 없애준다.
 - 즉, 예외를 사용한 쪽이 오히려 더 느리게 동작한다.
- 예외를 다른 곳(진짜 예외가 아닌 경우)에 사용한 경우
 - 코드를 헛갈리게 하고 성능을 떨어뜨린다.
 - 심지어, 제대로 동작하지 않을 수도 있다.
 - 반복문 내 버그가 숨어 있다면 흐름 제어에 쓰인 예외가 이 버그를 숨겨 디버깅을 어렵게 할 것이다.
- 예외는 오직 예외 상황에서만 써야 한다.
절대로 일상적인 제어 흐름용으로 쓰여서는 안 된다.
- 잘 설계된 API 라면 클라이언트가 정상적인 제어 흐름에서 예외를 사용할 일이 없게 해야함.
 - 특정 상태에서만 호출 할 수 있는 '상태 의존적 메서드'를 제공하는 클래스는, '상태 검사 메서드'도 함께 제공해야 한다.

- `Iterator` 인터페이스의 `next` (상태 의존적 메서드), `hasNext` (상태 검사 메서드)

```
for (Iterator<Foo> i = collection.iterator(); i.hasNext(); ) {
    Foo foo = i.next();
    ...
}
```

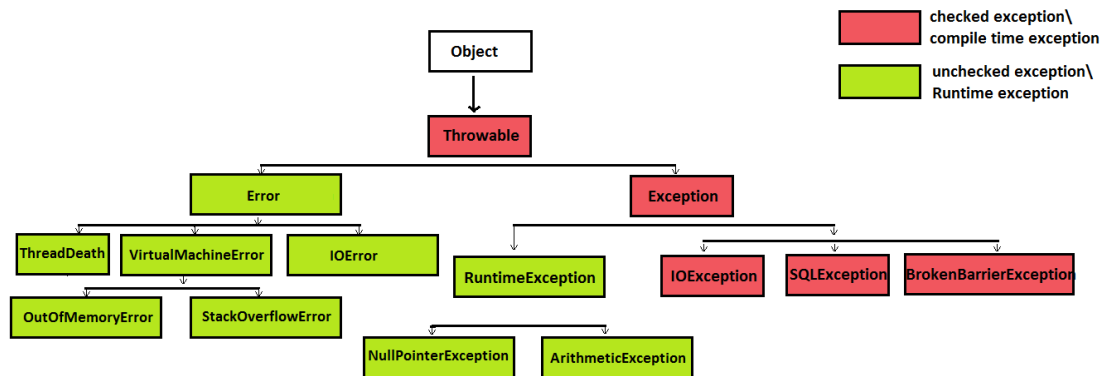
- 만약, `hasNext` 가 없다면

```
try {
    Iterator<Foo> i = collection.iterator();
    while(true) {
        Foo foo = i.next(0);
        ...
    }
} catch (ArrayIndexOutOfBoundsException e) {
}
```

- 굉장히 맨 처음 ex 과 비슷해보인다.
 - 반복문에 예외를 사용하면 상황하고 헷갈리며 속도도 느리고, 엉뚱한 곳에서 발생한 버그를 숨기기도 한다.
- 상태 검사 메서드 대신 사용할 수 있는 선택지도 있다.
올바르지 않은 상태 일 때 빈 옵셔널 (아이템 55), 혹은 `null` 같은 특수 값 반환
 1. 외부 동기화 없이 **여러 스레드가 동시에 접근** 할 수 있거나, **외부 요인으로 상태가 변할** 수 있다면 옵셔널이나 특정 값을 사용한다.
상태 검사 메서드와 상태 의존적 메서드 호출 사이에 객체의 상태가 변할 수도 있기 때문이다.
 2. 성능이 중요한 상황에서 **상태 검사 메서드가 상태 의존적 메서드의 작업 일부를 중복** 수행한다면 옵셔널이나 특정 값을 선택한다.
 3. 다른 모든 경우엔 상태 검사 메서드 방식이 조금 더 낫다.
가독성이 더 좋고, 잘못 사용시 발견하기 쉽다.
상태 검사 메서드 호출을 하지 않았다면 상태 의존적 메서드가 예외를 던져 버그를 들어 낼 것이다. 반면 특정 값을 검사하지 않고 지나쳐도 발견하기 어렵다.
(옵셔널은 해당하지 않는 문제)

▼ 70. 복구할 수 있는 상황에는 검사 예외를, 프로그래밍 오류에는 런타임 예외를 사용하라

- 자바가 문제 상황을 알리는 타입
 - java 예외



- 검사 예외 (checked exception / compile time exception)
- 비 검사 예외 (unchecked exception / Runtime exception)
 - 런타임 예외
 - 에러
- 검사 예외
 - 호출하는 쪽에서 복구하리라 여겨지는 상황이라면 사용한다.
이는 검사와 비검사 예외를 구분하는 기본 규칙이다.
 - 호출자가 그 예외를 `catch` 로 잡아 처리하거나 더 바깥으로 전파하도록 강제하게 된다.
따라서 메서드 선언에 포함된 검사 예외 각각은 그 메서드를 호출했을 때 발생할 수 있는 유력한 결과임을 API 사용자에게 알려주는 것이다.
 - API 설계자는 API 사용자에게 검사 예외를 던져주어, 그 상황에서 회복해내라고 요구한 것이다.
 - 물론 사용자는 예외를 잡기만 하고 별다른 조치를 취하지 않을 수도 있지만, 이는 좋지 못하다. (아이템 77)
 - 일반적으로 복구할 수 있는 조건일 때 발생하므로, 호출자가 예외 상황에서 벗어나는 데 필요한 정보를 알려주는 메서드를 함께 제공하는 것이 중요하다. (아이템 75)
- 비검사 예외

- 런타임 예외, 에러
- 둘 다 동작 측면에서는 다르지 않다.
프로그램에서 잡을 필요가 없거나 혹은 통상적으로는 잡지 말아야 한다.
- 프로그램에서 런타임 예외나 에러를 던졌다는 것은 복구가 불가능하거나 더 실행해봤자 손해가 더 많다는 뜻이다. 이런 `throwable` 을 잡지 않은 스레드는 적절한 오류 메시지를 뱉으며 중단된다.

• 런타임 예외

- 프로그래밍 오류를 나타낼 때는 런타임 예외를 사용하자.
- 런타임 예외의 대부분은 전제 조건을 만족하지 못한 경우 발생된다.
- 전제 조건 위반란 단순히 클라이언트가 해당 API 의 명세에 기록된 제약을 지키지 못했다는 뜻이다.
 - 배열의 index 는 `0 ~ array.length() - 1` 사이 이다
 - `ArrayIndexOutOfBoundsException` 이 발생했다는 뜻은 이 전제 조건을 어긴 것이다.

• 예외

- 보통은 JVM의 자원 부족, 불변식 깨짐 등 더 이상 수행을 계속할 수 없는 상황을 나타낼 때 사용한다.
- 자바 언어 명세가 요구하는 것은 아니지만, 널리 퍼진 규약으로 `Error` 클래스를 상속해 하위 클래스를 만드는 일은 자제하기 바란다.
- 즉, 구현하는 비검사 `throwable` 은 모두 `RuntimeException` 의 하위 클래스여야 한다.
- `Error` 는 상속하지 말아야 할 뿐 아니라, `throw` 문으로 직접 던지는 일도 없어야 한다.
- 이상 조건에서 문제가 있다면, 복구할 수 있는 상황인지, 오류인지는 명확히 구분되지 않는다.
 - 예를 들어 자원 고갈은 말도 안 되는 크기의 배열을 할당해 생긴 프로그래밍 오류 일 수도 있고, 진짜로 자원이 부족해서 발생한 문제일 수도 있다.
 - 만약, 자원이 일시적으로만 부족하거나 수요가 순간적으로만 몰린 것이라면 충분히 복구할 수 있는 상황일 것이다.
 - 따라서 해당 자원 고갈 상황이 복구될 수 있는 것인지는 API 설계자의 판단에 달렸다.

- 복구가 가능하다고 믿는다면 검사 예외를, 아니라면 런타임 예외를 사용하자.
 - 확신하기 어렵다면, 비검사 예외를 선택하는 편이 좋다. (아이템 71)
- **throwable 절대 사용하지 말자.**
 - **throwable** 은 정상적인 검사 예외보다 나을게 하나도 없으면서 API 사용자를 헛갈리게 한다.

▼ 71. 필요 없는 검사 예외 사용은 피해라

- 검사 예외
 - 제대로 활용하면 API 와 프로그램의 질을 높일 수 있다.
 - 결과를 코드로 반환하거나 비검사 예외를 던지는 것과 달리, 검사 예외는 발생한 문제를 프로그래머가 처리하여 안전성을 높이게끔 해준다.
 - 과하게 사용하면, 오히려 쓰기 불편한 API 가 된다.
 - 메서드가 검사 예외를 던질 수 있다고 선언했다면, 이를 호출하는 코드에서는 catch 블록을 두어 그 예외를 붙잡아 처리하거나 더 바깥으로 던져 문제를 전파해야 한다.
 - 어느 쪽이든 API 사용자에게 부담을 준다.
 - 검사 예외를 던지는 메서드는 스트림안에서 직접 사용할 수 없다. (아이템 45 ~ 48)
- 사용하면 좋을 때
 - API 를 제대로 사용해도 발생할 수 있는 예외
 - 프로그래머가 의미 있는 조치를 취할 수 있는 경우
 - **하지만, 둘 중 어디에도 해당하지 않는다면 비검사 예외를 사용하자**
- 하나의 검사 예외만 던질 경우
 - API 사용자는 try 블록을 추가해야 하고 스트림에서 직접 사용 불가능해진다.
 - 회피하는 방법
 - 적절한 결과 타입을 담은 옵셔널을 반환하는 것이다. (아이템 55)
검사 예외를 던지는 대신 단순히 빈 옵셔널을 반환하면 됨
하지만, 예외 발생 이유를 알려줄 정보를 담을 수 없음.
 - 검사 예외를 던지는 메서드를 2개로 쪼개 비검사 예외로 변경할 수 있다.

ex) 검사 예외를 던지는 메서드, 상태 검사와 비검사 예외를 던지는 메서드

```
// 리팩터링 before
try {
    obj.action(args);
} catch (TheCheckedException e) {
    ... // 예외 상황에 대처
}

// 리팩터링 after
if (obj.actionPermitted(args)) {
    obj.action(args);
} else {
    ... // 예외 상황에 대처
}
```

- 리팩터링 후 로직의 단점
 - 모든 상황에 적용 불가
 - 외부 동기화 없이 여러 스레드가 동시에 접근할 수 있거나, 외부 요인에 의해 상태가 변할 수 있다면 Thread-Safe 하지 않다.
 - `actionPermitted`, `action` 메서드가 작업 일부를 중복 수행한다면, 성능에서도 손해
- 정리
 - 필요한 곳에서 검사 예외는 프로그램의 안정성을 높인다. 남용하면 쓰기 어려운 API를 만든다.
 - API 호출자가 예외 복구를 할 수 없다면 비검사 예외를 던지자.
 - API 호출자가 예외 복구를 할 수 있고, 해주길 바란다면 옵셔널을 반환할 수 있을지 고민하자. 옵셔널로 상황을 처리하기에 충분한 정보를 줄 수 없을 때만 검사 예외를 던지자.

▼ 72. 표준 예외를 사용하라

- 좋은 코드
 - 코드를 재사용한다. 예외도 마찬가지로 재사용하는 것이 좋다.
 - Java Library 는 API 에서 쓰기 충분한 표준 예외를 제공한다.
- 표준 예외
 - 이미 익숙해진 규약을 그대로 따르기에, API가 다른 사람이 익히고 사용하기 쉬워진다.

- 예외 클래스 수가 적을수록 메모리 사용량도 줄고 클래스를 적재하는 시간도 적게 걸림.
- 표준 예외 종류
 - `IllegalArgumentException` (아이템 49)
 - 호출자가 인수로 부적절한 값을 넘길 때
 - 반복 횟수를 지정하는 매개변수에 음수를 건넬 때
 - **인수 값이 무엇이었던 어차피 실패하지 않았을 경우**
 - `IllegalStateException`
 - 대상 객체의 상태가 호출된 메서드를 수행하기에 적합하지 않을 때
 - 제대로 초기화되지 않은 객체를 사용 할 때
 - **인수 값이 무엇이었던 어차피 실패한 경우**
 - `NullPointerException`
 - `null` 값을 허용하지 않는 메서드에 `null` 값을 건넬 때
 - `IndexOutOfBoundsException`
 - 인덱스의 허용범 범위를 넘는 값을 건넬 때
 - `ConcurrentModificationException`
 - 단일 스레드에서 사용하려고 설계한 객체를 여러 스레드가 동시에 수정하려고 할 때
 - 사실, 동시 수정을 확실히 검출할 수 있는 안전된 방법이 없으니, 이 예외는 문제가 생길 가능성이 있다고 알려주는 info 역할로 쓰인다.
 - `UnsupportedOperationException`
 - 클라이언트가 요청한 동작을 대상 객체가 지원하지 않을 때
 - 대부분 객체는 자신이 정의한 메서드를 모두 지원하니 흔히 쓰이는 예외는 아니다.
 - 보통 구현하려는 인터페이스의 메서드 일부를 구현할 수 없을 때 쓰인다.
 - ex) `AbstractList` (추상 클래스) 의 `remove` 메서드

```
public E remove(int index) {
    throw new UnsupportedOperationException();
}
```


- `Exception`, `RuntimeException`, `Throwable`, `Error` 는 직접 재사용하지 말자.
 - 이 클래스들은 추상 클래스라고 생각하자.
 - 다른 예외의 상위 클래스임으로, 여러 성격의 예외를 포괄하는 클래스임으로, 안전하게 테스트 할 수 없다.
- 추가로 알면 좋은 예외
 - `ArithmeticException`
 - 산술 예외가 발생할 때, (나누기가 분모가 0 일 경우)
 - `NumberFormatException`
 - 문자열을 숫자 유형 중 하나로 변환하려고 시도하지만, 문자열에 적절한 형식이 없을 때 발생 (`"010"` 변환)
- 추가 내용
 - 상황에 부합한다면 항상 표준 예외를 재사용하자.
 - 더 많은 정보를 제공하기 원한다면, 표준 예외를 확장해도 좋다. 단, 예외는 직렬화할 수 있다는 사실을 기억하자. 이 사실만으로도 나만의 예외를 새로 만들지 않아야 할 근거로 충분하다.

▼ 73. 추상화 수준에 맞는 예외를 던져라

- 개요



수행하려는 일과 관련 없어 보이는 예외가 튀어나오면 당황스러울 것이다.

메서드가 저수준 예외를 처리하지 않고 바깥으로 전파해버릴 때 종종 일어나는 일이다. 사실 이는 단순히 프로그래머를 당황시키는데 그치지 않고, 내부 구현 방식을 드러내어 윗 레벨 API 를 오염시킨다. 다음 릴리스에서 구현 방식을 바꾸면 다른 예외가 나타나 기존 클라이언트 프로그램을 깨질 수도 있게 한다.

- 이 문제를 피하기 위해선 **상위 계층에서는 저수준 예외를 잡아 자신의 추상화 수준에 맞는 예외로 바꿔 던져야 한다.** 이를 **예외 번역**이라 한다.
- ex) 예외 번역

```
try {
    ... // 저수준 추상화를 이용한다.
```

```

    } catch (LowerLevelException e) {
        // 추상화 수준에 맞게 번역한다.
        throw new HigherLevelException(...);
    }

```

- ex) `AbstractSequentialList` 에서 수행하는 예외 번역

```

/**
 * 이 리스트 안의 지정한 위치의 원소를 반환한다.
 * @throws IndexOutOfBoundsException index 가 범위 밖이라면,
 *      즉 ({@code index < 0 || index >= size()}} 이면 발생한다.
 */
public E get(int index) {
    ListIterator<E> i = listIterator(index);
    try {
        return i.next();
    } catch (NoSuchElementException e) {
        throw new IndexOutOfBoundsException("인덱스: " + index);
    }
}

```

- `AbstractSequentialList` 는 `List` 인터페이스의 골격 구현(아이템 20) 이다.

• 예외 연쇄

- 예외를 번역할 때, 저수준 예외가 디버깅에 도움이 된다면, 예외 연쇄를 사용하는게 좋다.
- 문제의 근본 원인인 저수준 예외를 고수준 예외에 실어서 보내는 방식이다.
- 그러면, 별도의 접근자 메서드 (Throwable의 `getCause` 메서드)를 통해 필요하면 언제든지 저수준 예외를 꺼내 볼 수 있다.
- ex) 예외 연쇄

```

try {
    ... // 저수준 추상화를 이용한다.
} catch (LowerLevelException cause) {
    // 추상화 수준에 맞게 번역한다.
    throw new HigherLevelException(cause);
}

```

- 고수준 예외의 생성자는 (예외 연쇄용으로 설계된) 상위 클래스의 생성자에 '원인'을 건네주어, 최종적으로 Throwable 생성자까지 건네지게 한다.
- ex) 예외 연쇄용 생성자

```
class HigherLevelException extends Exception {
    HigherLevelException(Throwable cause) {
        super(cause);
    }
}
```

- 대부분 표준 예외는 예외 연쇄용 생성자를 갖추고 있다. 그렇지 않은 예외라도 `Throwable` 의 `initCause` 메서드를 이용해 '원인'을 직접 못 박을 수 있다.
 - 예외 연쇄는 문제의 원인을 (`getCause` 메서드로) 프로그램에서 접근할 수 있게 해주며, 원인과 고수준 예외의 스택 추적 정보를 잘 통합해준다.
- 무턱대고 예외를 전파하는 것 보다 예외 번역이 우수한 방법이지만, 그렇다고 남용해서는 곤란하다.
 - 가능하다면 저수준 메서드가 반드시 성공하도록 하여 아래 계층에서는 예외가 발생하지 않도록 하는 것이 최선이다. 때론 상위 계층 메서드의 매개변수 값을 아래 계층 메서드로 건네기 전에 미리 검사하는 방법으로 이 목적을 달성할 수 있다.
 - 차선책
 - 아래 계층에서의 예외를 피할 수 없다면, 상위 계층에서 그 예외를 조용히 처리하여 문제는 API 호출자에게까지 전파하지 않는 방법이 있다.
 - 이 경우 발생한 예외는 `java.util.logging` 같은 적절한 로깅 기능을 활용하여 기록해두면 좋다. (문제를 전파하지 않으면서도, 프로그래머는 로그 분석할 수 있음)

▼ 74. 메서드가 던지는 모든 예외를 문서화하라

▼ 75. 예외의 상세 메시지에 실패 관련 정보를 담으라

▼ 76. 가능한 한 실패 원자적으로 만들라

▼ 77. 예외를 무시하지 말라