10장 예외

분류	이펙티브 자바
태그	서적
시작일자	@2022년 6월 6일 오전 10:13
종료일자	@2022년 6월 13일 오후 7:45
참고	참고 블로그 [자바봄 기술 블로그] 참고 github.ido
스터디 링크	GitHub Link [이펙티브 자바 스터디]
서적	
Java	

▼ 목차

아이템 69. 예외는 진짜 예외 상황에만 사용하라.

❓ 진짜 예외 상황에만 사용하라? 가짜 예외 상황도 있는건가?

```
// 1번 코드
try {
    int i = 0;
    while(true)
        range[i++].climb();
} catch (ArrayIndexOutOfBoundsException e) {
}
```

> 무한 루프를 돌다가 배열의 끝에 도달해 ArrayIndexOutOfBoundsException 예외가 발생하면 catch 문에서 아무 동작도 하지 않고 끝내는 로직이다.
> 즉 예외를 제어 흐름용으로 사용한 것이다.

❓ 제어 흐름용으로만 쓰는거면 처음부터 끝까지 뽑아오는 for-each 문을 사용하면 되는거 아냐?

```
// 2번 코드
for (Mountain m : range) m.climb();
```

확실히 이전 코드보다 가족성도 좋고 명확하다.

❓ 그럼 어쩌다가 1번 방법을 생각해낸걸까?

> 잘못된 추론으로 인한 최적화 시도이다.

- JVM은 배열에 접근할 때마다 경계를 넘지 않는지 검사를 한다.
- 일반적인 반복문도 배열 경계에 도달하면 종료를 하게 된다.
  그래서 위와 같은 코드로 배열 경계에서 검사되는 로직을 하나 제거 한 것이다.

위 내용으로 추론을 한다면 1번 코드는 딱히 문제가 될 것은 없어보인다.
하지만 이것은 잘못된 추론입니다. 아래 내용을 보면 알 수 있습니다.

1. 예외의 용도 자체가 예외 상황을 고려해서 만든 것이기 때문에 JVM 구현자는 명확한 검사만큼 빠르게 만들 이유가 별로 없기에 최적화가 안돼있을 확률이 높다.
2. 코드를 try-catch 블럭에 넣으면 JVM이 적용할 수 있는 최적화가 제한된다.
3. 배열을 순회하는 표준 관용구는 앞서 고민했던 중복 검사를 수행하지 않고, JVM이 알아서 최적화 해준다.

즉 위와 같이 예외를 사용한 최적화를 할 필요가 없으며, 오히려 더 느리게 동작한다.

위와 같이 예외를 예외상황이 아니게 사용하는 경우 문제는 또 있다.

```java
class MountainTest {
    static Mountain[] range = new Mountain[10000000];
    @BeforeEach
    void 세팅() {
        for (int i = 0; i <range.length; i++) range[i] = new Mountain();
    }

    @Test
    @DisplayName("예외를 사용한 반복")
    void 예외사용(){
        long start = System.currentTimeMillis();
        int i = 0;
        try {
            while (true) range[i++].climb();
        } catch (Exception e) {
        }
        long end = System.currentTimeMillis();
        System.out.println("예외를 이용한 반복이 걸린 시간 : " + (end-start) + "초(ms)");
    }

    @Test
    @DisplayName("for-each 사용한 반복")
    void for_each사용(){
        long start = System.currentTimeMillis() ;
        for (Mountain mountain : range) mountain.climb();
        long end = System.currentTimeMillis();
        System.out.println("for-each 이용한 반복이 걸린 시간 : " + (end-start) + "초(ms)");
    }
}
```

```
/Library/Java/JavaVirtualMachines/jdk-11.0.13.jdk/Contents/Home/bin/java ...
for-each 이용한 반복이 걸린 시간 : 11초(ms)
예외를 이용한 반복이 걸린 시간 : 14초(ms)

Process finished with exit code 0
```

또 다른 문제

> 예외를 예외상황이 아닌 제어 흐름과 같이 다른 상황에서 사용한다면, 속도 뿐 아니라 프로그램 유지 보수 측면에서도 별로이다.

- try 블럭 안에 로직을 작성하는데 여기서 관련 없는 배열에서 `ArrayIndexOutOfBoundsException` 이 발생해도, 이 버그를 정상적인 반복문 종료 상황으로 보고 넘어가게 된다.
- 실제로 예외가 던져져야 하는 상황에서도 그러지 못하기에 예기치 못한 곳에서 예외가 발생해 디버깅의 어려움을 겪을 수 있다.

상태 검사 메서드, 옵셔널, 특정 값 중 선택을 하는 팁

> 예외를 통해 제어흐름을 하지 않으면서 할 수 있는 방법
> 상태 검사 메서드를 사용해 진행 여부(반복)을 선택할 수도 있고, 비어있는 옵셔널 혹은 null과 같은 특정 값을 반환할 수도 있다.

1. **여러 스레드가 동시에 접근가능하거나 외부에서 상태를 바꿀 수 있다**
   - 옵셔널이나 특정 값을 사용한다.
   - 상태 검사 메서드와 상태 의존적 메서드의 동작 사이에 값이 변해버릴 수 있다.
2. **성능이 중요한 상황에서 상태 검사 메서드가 상태 의존적 메서드의 일과 중복되는 일을 한다.**
   - 옵셔널이나 특정 값을 사용한다.
3. **다른 모든 경우에는 상태 검사 메서드, 상태 의존적 메서드를 제공하자.**

아이템 70. 복구할 수 있는 상황에서는 검사 예외를, 프로그래밍 오류에는 런타임 예외를 사용하라.

> 프로그램 실행시 문제가 발생하는 것을 오류라고 한다.
> 이런 오류를 알리는 타입을 Throwable이라고 한다.

- 예외(checked exception), 런타임 에러(unckecked exception), 에러(error) 총 3가지로 분류 할 수 있다.

호출하는 쪽에서 복구하리라 여겨지는 상황이라면 검사 예외를 사용하라.

> checked와 unckecked exception을 구분하는 기본 규칙으로,
> 검사 예외의 경우 호출자가 그 예외를 잡아 처리하거나 바깥으로 전파하도록 강제하게 된다.

```
public static void main(String[] args) throws IOException {
    // CheckedException 이 발생하는 코드
    // 즉 예외처리
    byte[] inSrc = Arrays.asList(args);//{0, 1, 2, 3, 4};
    byte[] outSrc = null;
    byte[] temp = new byte[4];


    ByteArrayInputStream input = new ByteArrayInputStream(inSrc);
    ByteArrayOutputStream output = new ByteArrayOutputStream();

    try {
        while (input.available() > 0) {
            int len = input.read(temp);
            output.write(temp, 0, len);
        }
    } catch (IOException ie) {
    }

    outSrc = output.toByteArray();

    System.out.println("Input Source: " + Arrays.toString(inSrc));
    System.out.println("temp: " + Arrays.toString(temp));
    System.out.println("Output Source: " + Arrays.toString(outSrc));
}
```

프로그래밍 오류를 나타낼 때는 런타임 예외를 사용하자.

> 비검사 throwable은 Runtime Exception과 Error 두 가지가 있다.
> 둘 다 프로그램에서 잡을 필요도 없고 잡지 않아야 하는 상황에서 이 경우 적절한 오류 메시지와 함께 중단된다.

```
public class Account{
  private String name;

  public Account(String name) {
    if(name.length() > 6){
      throw new IllegalArgumentException("이름은 최대 6자까지 가능합니다.");
    }
    this.name = name;
  }
}
```

대부분 런타임 예외는 전제조건을 만족하지 못했을 경우를 의미한다.

Error는 JVM의 자원부족, 불변식 깨짐 등 더 이상 수행이 불가능한 상황을 나타낼 때 사용하는데, 이 Error 클래스를 상속해 하위 클래스를 만드는 일은 자제해야 한다.

Throwable은 그냥 사용하지 말자.

> Exception, RuntimeException, Error같은 상위 예외를 상속하지 않는 throwable을 만들 수 도 있다.
> 하지만 좋을게 없으니 사용하지 않도록 하는게 낫고, 더 좋지도 않으면서 개발자를 헷갈리게 할 뿐이다.

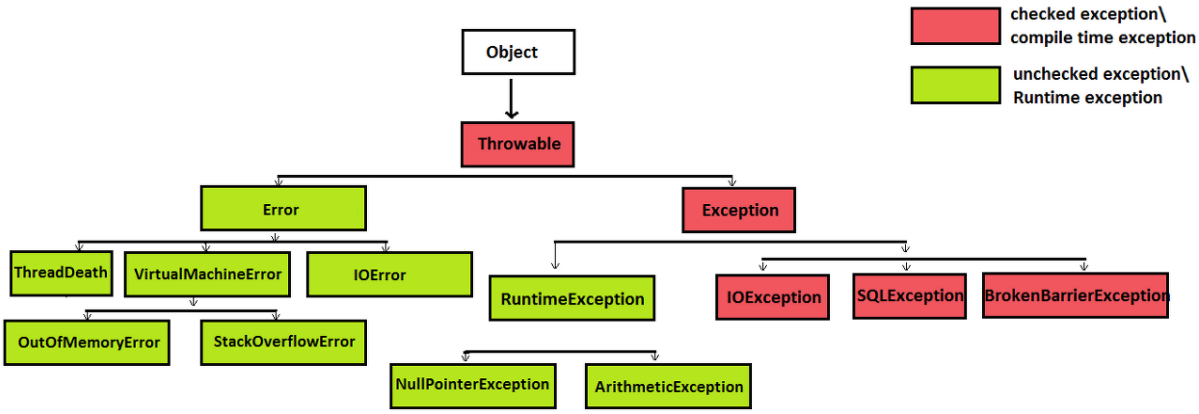
검사 예외는 복구에 필요한 정보를 제공하자.

> 검사 예외는 일반적으로 복구할 수 있는 조건에서 발생한다.

호출자가 예외 상황에서 벗어나기 위해 필요한 정보를 알려주는 메서드를 함께 제공하는것이 중요하다.

아이템 71. 필요없는 검사 예외 사용은 피하라.

❓ 검사예외? 비검사예외? 뭐가 다르지?



검사예외(checked exception) → Compile time exception

> 어플리케이션 수행 중에 일어날법한 예외를 검사하고 대비하라는 목적으로 사용
> 반드시 예외 처리를 해야 한다. 컴파일러가 발견해서 컴파일 오류를 발생시킨다.

- try-catch(예외 복구), throws(예외 회피) 방식이 존재한다.

비검사 예외(unchecked exception) → Runtime exception

> Error는 시스템적인 예외를 의미하며, 심각한 상황에서 발생하는 예외이다.
> 컴파일러가 예외 처리 여부를 확인하지 않는다.
> 즉 개발자가 예외 처리 코드를 추가할지 말지 결정한다.

- NPE, IndexOutBoundsException 등

> 🌟 **검사 예외를 제대로 사용하자.**
> 검사 예외를 제대로 활용하면 API와 프로그램의 질을 높일 수 있으며, 발생한 문제를 프로그래머가 처리해 안정성을 높이게 해준다.

검사예외를 과하게 사용하게 될 경우

> 검사 예외를 과하게 사용하면 오히려 쓰기 불편한 API가 된다.

- 이미 다른 검사 예외도 던지고 있는 상황에서 또 다른 검사 예외를 추가하는 경우라면 catch문 하나만 추가하면 된다.
- 하지만 검사 예외가 단 하나 뿐이라면 오직 그 예외 때문에 API 사용자는 try/catch를 추가해야만 하고, 스트림에서 직접 사용하지 못하게 한다.

검사예외를 안쓰고 코드를 작성하는 방법

1. **Optional**

> Optional을 반환하면 된다.

```
// Before
User user;
try {
    user = userService.findUserById(1L); // 없으면 Exception
} catch (Exception e) {
    user = new User(1L);
    e.printStackTrace("유저가 존재하지 않습니다.");
}
```

```
// After
User user = userService.findUserById(1L)
             .orElseGet(() -> new User(1L));
```

하지만 이 방법도 예외가 발생하는 이유 같은 예외 관련 정보를 전달할 수 없게 됩니다.

2. **기존 메서드를 2개로 쪼개 비검사 예외로 바꾸기**

```
// Before
try {
  obj.action(args);
} catch (TheCheckedException e) {
  ... // 예외 처리
}
```

```
// After
if(obj.actionPermitted(args)){
  obj.action(args);
} else {
  ... // 예외 처리
}
```

물론 이 방법도 완벽하다고 할수가 없다.

- 외부 동기화 없이 여러 쓰레드가 동시에 접근할 수 있거나 외부 요인에 의해 상태가 변환할 수 있다면 이 코드는 옳지 않다.
- actionPermitted메서드가 action 메서드의 작업 일부를 중복 수행한다면 성능에서 손해이다.

API 호출자가 예외상황에서 복구할 방법이 없는가?

YES → 비검사 예외

NO(복구가 가능하다.) → 예외를 호출자가 처리해주길 바란다.

YES → Optional 반환 고려(예외를 따로 던지지 않지만 예외가 발생할 수 있음

NO → 예외를 API에서 던진다. (검사 예외를 던진다.)

아이템 72. 표준 검사를 사용하라.

대표적인 표준 예외

- **IllegalArgumentException**

  → 허용하지 않은 값이 인수로 전달되었을 때

- **IllegalStateException**

  → 객체가 메서드를 사용하기에 적절하지 않은 상태일 때

- **NullPointerException**

  → null을 허용하지 않는 메서드가 null을 인자로 받을 때

- **IndexOutOfBoundsException**

  → 인덱스의 범위를 넘어설 때

- **ConcurrentModificationException**

  → 허용되지 않은 동시 수정이 발견되었을 때

- **UnsupportedOperationException**

  → 호출한 메서드가 지원하지 않을 때

🌟 **Exception, RuntimeException, Throwable, Error는 직접 재사용하지 말아야한다.**

주의 사항

> 상황에 부합하다면 항상 표준 예외를 재사용하자.

하지만 IllegalArgumentException과 IllegalStateException 이 두 개가 어디에 사용하는지 바로 아는가 부터 체크해야한다.

만약 모를경우 API문서를 다시 참고해야 하며 자세히 말해보자면 아래와 같다.

보통 어떤 값이 되든 어차피 실패했을 것이라면 State, 그렇지 않으면 Argument 에러를 던진다.

카드 덱을 예시로 현재 덱의 카드수보다 높은 값이라면 State, 낮은 값이라면 Argument을 던지라는 말이다.

아이템 73. 추상화 수준에 맞는 예외를 던져라

예외 번역

> 메서드가 저수준 예외를 처리하지 않고 바깥으로 전파하는 경우에 대한 문제의 해결책이다.

상위 계층에서 저수준 예외를 잡아 자신의 추상화 수준에 맞는 예외로 바꿔 던져야한다.

```
public abstract class AbstractSequentialList<E> extends AbstractList<E> {

    /**
     * Returns the element at the specified position in this list.
     *
     * @param index index of the element to return
     * @return the element at the specified position in this list
     * @throws IndexOutOfBoundsException if the index is out of range
     *         (index < 0 || index >= size())
     */
    public E get(int index) {
        try {
            return listIterator(index).next();
        } catch (NoSuchElementException exc) {
            throw new IndexOutOfBoundsException("Index: "+index);
        }
    }
}
```

예외 번역시 저수준 예외가 디버깅에 도움이 된다면 **예외 연쇄(exception chaining)**를 사용하자.

예외 연쇄

> 문제의 근본 원인인 저수준 예외를 고수준 예외에 실어 보내는 방식이다.

접근자 메서드(Throwable, getCause 메서드)를 통해 필요하면 언제든 저수준 예외를 꺼내 볼 수 있다.

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

class ChildClass {
    public void publicAPIMethod() {
        try {

        } catch (LowerLevelException cause) {
            throw new HigherLevelException(cause);
        }
    }
}
```

- 대부분의 표준 예외는 예외 연쇄용 생성자를 갖고 있다.
- 예외 연쇄는 문제의 원인을 getCause 메서드로 접근할 수 있게 해주며, 원인과 고수준 예외의 스택 추적 정보를 통합한다.
- **예외를 전파하는 것보다 예외 번역하는 것이 우수한 방법이지만 남용해서는 안된다.**
- 가능하다면 저수준 메서드가 반드시 성공하도록하여 아래 계층에서는 예외가 발생하지 않도록 하는 것이 최선이다.
- 상위 계층 메서드의 매개변수 값을 아래 계층 메서드로 건내기 전에 미리 검사하는 방법으로 목적을 달성할 수 있다.

> 🌟 **차선책**
> 아래 계층에서의 예외를 피할 수 없다면 상위 계층에서 그 예외를 처리하여 문제를 API호출자까지 전파하지 않게하자.
> 이러한 경우 적절한 로깅 기능을 활용해 기록하자.
> 클라이언트 코드와 사용자에게 문제를 전파하지 않으면서도 프로그래머가 로그를 분석해 추가 조치를 취하게 도와주기 때문이다.

아이템 74. 메서드가 던지는 모든 예외를 문서화하라

> 메서드가 던지는 예외는 그 메서드를 올바르게 사용하는데 아주 중요한 정보이다.
> 따라서 검사 예외는 항상 따로따로 선언하고 각 예외가 발생하는 상황을 자바독의 `@trows` 태그를 사용하여 정확히 문서화하자.

잘못된 예시

```
public void examMethod() throws Exception {
    // 로직 실행
}
```

모든 예외를 Exception으로 던지게 된다면 메서드 사용자는 각 예외에 대처할 방법이 없습니다.

→ 다른 예외까지 삼켜버릴 수 있기 때문에 API 활용성이 크게 떨어집니다.

Main 메서드는 예외가 됩니다. JVM만이 호출하기 때문입니다.

올바른 예시

```
/**
 * @throws SQLException SQL 이 잘못된 경우
 * @throws ClassNotFoundException 지정한 경로에 클래스파일이 존재하지 않는경우.
 */
public void examMethod() throws SQLException,ClassNotFoundException{
    Class.forName("/test/path/test.class");
    // 로직 실행
}
```

항상 비검사 예외도 검사 예외처럼 정성껏 문사화를 해야 한다.

잘못된 문서화 예시

```
/**
 * @throws SQLException SQL 이 잘못된 경우
 * @throws ClassNotFoundException 지정한 경로에 클래스파일이 존재하지 않는경우.
 * @throws NullPointerException 지정한 요소에 null 이 들어오는 경우
 */
public void examMethod() throws SQLException,ClassNotFoundException,NullPointerException{
    Class.forName("asd");
    // 로직 실행
}
```

비검사 예외를 throws 목록에 포함시킴

올바른 문서화

```
/**
 * @throws SQLException SQL 이 잘못된 경우
 * @throws ClassNotFoundException 지정한 경로에 클래스파일이 존재하지 않는경우.
 */
public void examMethod() throws SQLException,ClassNotFoundException{
    Class.forName("asd");
    // 로직 실행
}
```

한 클래스에 정의된 많은 메서드가 같은 이유로 예외를 던진다면?

```
public class Exam {
    /**
     * @throws SQLException SQL 이 잘못된 경우
     * @throws ClassNotFoundException 지정한 경로에 클래스파일이 존재하지 않는경우.
     * @throws NullPointerException 지정한 요소에 null 이 들어오는 경우
     */
    public void examMethod() throws SQLException,ClassNotFoundException{
        Class.forName("asd");
        // 로직 실행
    }
    /**
     * @throws NullPointerException 지정한 요소에 null 이 들어오는 경우
     */
    public void examMethod2() throws IOException{
       // 로직 실행
    }
}
```

클래스 설명에 추가한다.

```
/**
 * <p> {@code NullPointerException} 지정한 요소에 null 이 들어오는 경우 발생합니다.</p>
 */
public class Exam {

    /**
     * @throws SQLException SQL 이 잘못된 경우
     * @throws ClassNotFoundException 지정한 경로에 클래스파일이 존재하지 않는경우.
     */
    public void examMethod() throws SQLException,ClassNotFoundException{
        Class.forName("asd");
        // 로직 실행
    }
```

```
    public void examMethod2() throws IOException{
       // 로직 실행
    }

}
```

아이템 75. 예외의 상세 메세지에 실패 관련 정보를 담으라.

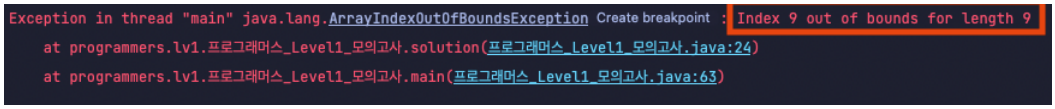
> 예외가 발생하여 프로그램이 실패했을때 프로그래머는 스택 트레이스 정보를 확인하여 버그를 해결한다.
> 하지만 이때 예외 메시지가 정보를 담고 있지 않다면? 혹시나 틀린 정보라면? 다시 버그를 재현하지 못한다면?
> 다양한 문제가 생긴다.

예외 상세 메시지 예시

> 실패 순간을 포착하기 위해서는 관여된 모든 매개변수와 필드의 값을 실제 메시지에 담아야한다.

예시로 `IndexOutOufBoundsException` 이 있다.

해당 예외는 메시지는 범위의 최솟값과 최댓값 그리고 그 범위를 벗어났다는 인덱스의 값을 담아준다.



하지만 항상 구구절절 모든 것을 작성하라는 것이 아니다.

예외 발생시 stack trace에서는 예외가 발생한 소스명과 줄 번호가지 모두 기록돼있기 때문이다.

그러니 문서나 소스코드에서 얻을 수 있는 정보를 제외하고 간단 명료하게 적어두는게 좋다.

예외는 실패 관련 정보를 접근자 메서에 적절히 제공하는게 좋다.

> 예외의 목적은 개발자에게 포착한 실패 정보를 잘 전달해 예외 상황을 보구하는데 있다.
> 그렇기에 적절한 접근자 메서드를 이용해 예외 내용을 전달할 수 있도록 해야 한다.

이런 실패 정보는 비검사 예외 보다는 검사 예외에서 더욱이 유용하게 사용된다.

바로 상황을 복구할 수 있는 정보를 받아 복구할 수 있기 때문이다.

아이템 76. 가능한 한 실패 원자적으로 만들어라.

❓ **실패 원자적?**

> 호출된 메서드가 실패하더라도 해당 객체는 메서드 호출 전 상태를 유지하도록 하는 특성을 실패 원자적이라 한다.
> 즉 메서드의 실패가 전체 로직에 영향을 주지 않도록 하는 것이다.

1. 불변 객체

> 이전에 작성했던 아이템 17을 보면 더 좋다.

불변 객체로 설계할 경우 태생적으로 실패 원자적이 된다.

메서드가 실패하면 새로운 객체가 만들어지지는 않을 수 있지만, 기존 객체가 불완전한 상태가 되는 경우는 없다.
말 그대로 불변이기 때문이다

2. 로직 수행 전 매개 변수 유효성 검사

> 아이템 49 매개변수가 유효한지 확인하라 다시 보기

객체의 내부 상태를 변경하기 전 매개변수의 유효성을 검사한다면?

잠재적 예외의 가능성 대부분을 걸러낼 수 있다.

```
// stack의 pop() 메서드
public Object pop() {
  if(size == 0) throw new EmptyStackException();

  Object result = elements[--size];
  elements[size] = null; //다 쓴 참조 해제
```

```
  return result;
}
```

> 메서드 호출후 바로 size 값을 확인해서 0인 경우 예외를 던져준다.

3. 실패할 수 있는 코드를 객체의 상태를 바꾸기전에 수행하는 방식

> 계산을 수행하보기 전에 인수의 유효성을 검사해볼 수 없을 때 앞서 방식에 덧붙여 쓸수 있는 방식이라고 생각하며 된다.

❓ **이게 무슨말이지??**

- TreeMap은 원소들을 어떤 기준에 맞춰서 정렬한다.
- 이때 그 원소는 기준에 따라 비교할 수 있는 타입이어야 한다.
  엉뚱한 타입이 들어오면 Tree를 변경하기 전에 ClassCastException을 던지게 될 것이다.

4. 객체의 임시 복사본에서 작업을 수행하고 원 객체와 교체하는 방식

> 데이터를 임시 자료구조에 저장해 작업하는게 더 빠르다면 적용하기 좋은 방식이다.

- 예를들어 어떤 정렬 메서드에서는 정렬을 수행하기 전에 입력 리스트의 원소들을 배열로 옮겨 담는다.
- 배열을 사용하면 정렬 알고리즘의 반복문에서 원소들에 빠르게 접근가능하기 때문이다.
  → 이에 대한 이점은 성능도 있지만, 정렬에 실패해도 입력 리스트는 변하지 않는 효과를 덤으로 얻게 된다.

5. 작업 도중 발생하는 실패를 가로채고 복구 코드를 작성하여 작업 전 상태로 되돌리는 방법

> 디스크 기반내 내구성을 보장해야 하는 자료구조에 쓰이는데, 자주 쓰이지는 않는다.

하지만 항상 실패 원자성을 달성하기는 어렵고 할 수도 없다.

> 멀티 쓰레드 환경에서 객체를 동시에 수정하려 하면 객체의 일관성이 깨질 수 있는데 이런 예외를 잡아냈다고 해서 쓸 수 있는 상태라고 하기도 힘들다.
> 또한 Error는 복구할 수 없기에 AssertionError에 대해서는 실패 원자성으로 만들 필요가 없다.

🌟 실패 원자적으로 만들기 위한 비용 및 복잡도가 아주 큰 경우 트레이드 오프를 계산해 결정할 필요가 있다.
다만, 이런 경우 실패할 경우 객체의 상태를 API 설명에 명시해야 한다.

아이템 77. 예외를 무시하지 말라.

🌟 **예외 처리란, 예외 상황에 적절한 처리를 하기 위해 존재한다.**

예외를 무시하는 것

```
try {
  // TODO
} catch (Exception e) {
}
```

❓ 예외를 무시해야 할 때가 있는가?

> 예외를 무시해야 하는 경우도 존재한다.
> 하지만 같은 예외가 계속 발생한다면 조사해 볼 필요가 있다. 즉 로그라도 남기라는 이야기이다.
> 또한 예외를 무시하기로 했다면 이유를 주석으로 남기고 예외 변수의 이름도 ignored로 바꾸도록 하자.

1. 입력 전용 스트림이기 때문에 파일 자체의 상태를 변경하지 않았기 때문에 복구할 필요가 없다.
2. 스트림을 닫을 때 예외가 발생한다는 것은 이미 읽은 건 다 읽었다는 뜻이기 때문에 예외를 무시할 수 있다.

예외를 무시하지 않아야 하는 이유

> 예외를 무시하고 지나가면 프로그램은 오류를 내재한 채로 작동하게 된다.
> 그 결과는 예측할 수 없게 되고 전혀 어뚱한 곳에서 오작동이 발생할 수 있다.

9장 일반적인 프로그래밍 원칙

11장 동시성