7장 오류 처리

오류 코드보다 예외를 사용하라

논리와 오류 처리 코드가 뒤섞이지 않는다.

호출자를 고려해 예외 클래스를 정의해라

오류를 분류하는 것보다 잡는 것이 더 중요하다

Try-Catch-Finally 문부터 작성하라

프로그램에 트랜잭션 범위를 정의하는 것이다.
즉, 호출자가 기대하는 상태를 작성한다.

예외에 의미를 제공하라

오류 발생 원인과 위치 찾기가 쉬워진다

예외 전후 상황을 충분히 기술해라

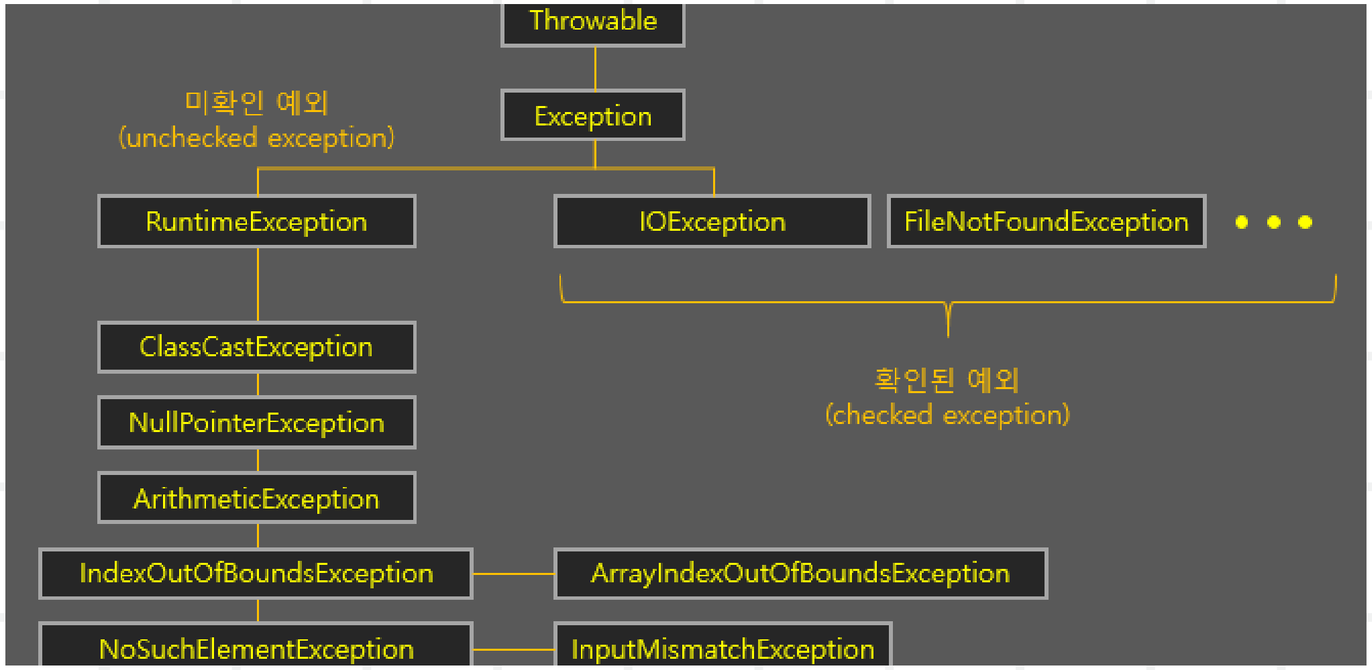
정상 흐름을 정의하라

예외가 논리의 흐름을 해치면 안된다

미확인 예외를 사용하라

확인된 예외란?
컴파일러 단계에서 확인 되는 예외

예외를 던지는 메소드가 수정되면 상위 메소드들도 전부 수정돼야한다.
결과적으로 캡슐화가 깨지는 현상이 발생한다.
+의존성이 증가



NULL을 반환/전달하지 마라

null을 확인하고 놓친 것을 찾는 시간과 비용이 너무 크다
-> 특수 사례 객체나 감싸기 메서드를 통한 예외 던지는 방식을 쓰자

의도한 경우가 아니라면 null을 넘긴 것에 대한 근본적인 해결책이 없다
-> **null**을 넘기지 못하게 금지하는 것이 제일 효율적이다

```
if (item == null) {
    throw new NullArgumentException("item");
}

//중간 생략
List<PortalMenu> children = getMenuItems(item.getPortal().getId(), item.getId()); // 603번째 줄
```

```
Caused by: java.lang.NullPointerException
    at com.mycompany.service.impl.PortalManagerImpl.deleteMenuItem(PortalManagerImpl.java:603)
```

8장 경계

외부 코드 사용하기

- 인터페이스 제공자와 사용자는 상충되는 목적으로 인터페이스를 제공/사용한다
- 외부 코드에서 제공하는 경계형 인터페이스를 여기저기 넘기면 안 된다
  - 사용자에게 목적에 반하는 메소드도 제공할 수 있다
  - 인터페이스가 바뀜에 따라 수정할 코드가 늘어난다

-> 감싸기 메서드/클래스로 외부에 노출시키지 않는 방법이 있다

경계 살피고 익히기

외부 코드와 내부 코드의 경계를 구분하고 접근하자

테스트 케이스를 작성해서 외부 코드를 먼저 익히자(학습 테스트)

학습 테스트는 공짜 이상이다

학습 테스트는 이해도를 높여주는 정확한 실험이다
통합 이후에도 재사용 할 수 있다

아직 존재하지 않는 코드를 사용하기

아는 코드와 모르는 코드를 분리하자

```
private fun handleSignInResult(result: Intent?) {
    GoogleSignIn.getSignedInAccountFromIntent(result)
        .addOnSuccessListener { googleAccount: GoogleSignInAccount ->
            // Use the authenticated account to sign in to the Drive service.
            val credential : GoogleAccountCredential! = GoogleAccountCredential.usingOAuth2(
                context: this, setOf(DriveScopes.DRIVE_FILE)
            )
            credential.selectedAccount = googleAccount.account
            googleDriveService =
                Drive.Builder(
                    GoogleNetHttpTransport.newTrustedTransport(),
                    GsonFactory(),
                    credential
                ) Drive.Builder
                    .setApplicationName("Drive API Migration") Drive.Builder!
                    .build()
        }
}
```

googleDriveService에서
nullpointer 오류 발생

```
fun createFile(fileName : String, f : java.io.File, orderId : Long, type: String) {
    val metadata : File! = File()
        .setParents(listOf("1yYYGZcoT1OoH6f3u3hvwvH5Y-44jvHT1"))
        .setName(fileName)
    val fileContent = FileContent( type: "image/$type",f)
    CoroutineScope(Dispatchers.IO) .launch { this: CoroutineScope
        val googleFile : File! = googleDriveService.files().create(metadata,fileContent).execute()
        db.DAO().insertAllImageReview(
            ImageReview(orderId, priority: 1, src: "https://drive.google.com/uc?id=${googleFile.id}")
        )
        if(googleFile == null)
            Log.i( tag: "error", msg: "Null result when requesting file creation.")
        else
            Log.i( tag: "file id ", googleFile.id)
    }
}
```

9장 단위 테스트

TDD 법칙

첫째 법칙

실패하는 단위 테스트를 작성할 때까지 실제 코드를 작성하지 않는다

둘째 법칙

컴파일은 실패하지 않으면서 실행이 실패하는 정도만 단위 테스트를 작성한다

셋째 법칙

현재 실패하는 테스트를 통과할 정도만 실제 코드를 작성한다

실제 코드와 맞먹을 정도로 방대한 테스트 코드는 관리 문제를 발생시킨다

깨끗한 테스트 코드 만들기

가독성이 제일 중요하다

- 명료성, 단순성, 풍부한 표현력이 필요하다 - 최소의 표현으로 많은 것을 나타내야 한다
- 잡다한(테스트에 필요 없는) 코드는 없애자
- BUILD-OPERATE-CHECK 패턴을 사용하자

BUILD
OPERATE
CHECK

```
public void testGetPageHierarchyAsXml() throws Exception{
    makePages( pageOne: "PageOne", s: "PageOne.ChildOne", pageTwo: "PageTwo");
    submitRequest( root: "root", s: "type:pages");
    assertResponseIsXml();
    assertResponseContains( s: "<name>PageOne</name>", s1: "<name>PageTwo</name>", s2: "<name>PageOne.ChildOne</name>");
}
```

깨끗한 테스트 코드 만들기

테스트 당 assert 하나

방향성

- assert문이 하나면 결론이 하나라서 코드를 이해하기 쉽고 빠르다
- 여러 개라면 쪼개서 하나씩으로 바꾸면 된다
- 테스트 코드 속에 감춰진 일반적인 규칙이 보인다
- 테스트 함수마다 한 개념만 테스트하라

이것은 테스트의 방향성일 뿐 절대적인 규칙은 아니다

assert문을 최소로 쓰라는 것이다

깨끗한 테스트 코드 만들기

F.I.R.S.T.

Fast: 테스트가 느리면 자주 돌리지 않게 됨 -> 초반에 문제를 고치지 못함

Independent: 의존하게 되면 연쇄 효과가 발생하게 됨 -> 원인을 진단하기 어려움

Repeatable: 어떠한 환경에서도 반복 가능해야 함 - 같은 결과를 도출

Self-Validating: Bool 값으로 결과를 도출해야함. 그렇지 않으면 판단은 주관적이 되며, 수작업으로 결과를 평가해야함

Timely: TDD 법칙을 따라야 한다 - 실제 코드를 구현하기 전에 테스트 코드를 작성해야한다