2025-09-03

오늘의 강의

예외 처리가 필요한 이유

- 프로그램의 안정성과 신뢰성 보장
- 오류의 조기 발견 및 대응
- 사용자 경험 개선
- 시스템 자원의 안전한 관리
  ex) `finally` 블록이나 try-with-resources를 통해 자원 해제

> 🖊 오류 분류
>
> **Error**: 수습할 수 없는 심각한 오류
> **Exception(예외)**: 예외 처리를 통해 수습할 수 있는 덜 심각한 오류

예외 처리의 두 가지 방법

try-catch-finally 문법

```
try {
    /**
     * 경우에 따라 예외가 발생할 수 있다.
     * 예외는 throw 문에 의해 직접적으로 발생할 수도 있고,
     * 또는 예외를 발생시키는 메서드의 호출에 의해 발생
     */
} catch (Exception e) {
    /**
     * try 블록에서 예외가 발생할 경우에만 실행
     * e를 사용하여 Error 다른 값을 참조 가능
     * 이 블록에서는 어떻게든 그 예외를 처리할  및 무시
     * throw 를 사용해서 예외를 다시 발생 가능
     */
} finally {
    /**
     * 이 블록에는 try 블록에서 일어난 일에 관계없이 무조건 실행
     * 이 코드는 try 블록이 어떻게든 종료되면 실행
     *   1) 정상적으로 블록의 끝에 도달했을 때
     *   2) break, continue 또는 return 문에 의해서
     *   3) 예외가 발생했지만 catch 절에서 처리했을 때
     *   4) 예외가 발생했고 그것이 잡히지 않은 채 퍼져나갈 때
     */
}
```

- `try`: 예외 발생을 조사하는 문장 검사
- `catch`: 예외가 발생했을 때 실행시킬 코드
- `finally`: 마지막에 반드시 실행시켜야 하는 코드

throws 문법

```
int myFun1(int a) throws Exception{
    if (a => 0){
        Throw new Exception();
    }
}

int fFun2(){
    try {
        myFun1(-1);
    } catch (Exception e) {
        System.out.printLn("Exception!!!");
    } finally {
        System.out.printLn("종료");
    }
}
```

- 메소드 선언부 겉부분에 작성하며, 여러가지 명시 가능
- 처리하지 않은 예외를 호출한 곳으로 떠넘기는 역활
- 예외 처리가 아닌 역활을 떠넘기는 것임으로 따로 처리 필요
- try-catch블록으로 예외를 최종 처리한느것이 바람직

예외 유형

1. **RuntimeException (언체크 예외)**
2. 컴파일러가 예외 처리를 강제하지 않음
3. 프로그래머의 실수로 발생하는 예외
4. 예: `NullPointerException`, `ArrayIndexOutOfBoundsException`, `ArithmeticException`

5. **Checked Exception (체크 예외)**
6. 컴파일러가 예외 처리를 강제함
7. 외부 환경에 의해 발생하는 예외
8. 예: `IOException`, `ClassNotFoundException`, `SQLException`

다중 catch 주의사항

- **구체적인 예외를 먼저 처리**
- `Exception`은 모든 예외의 부모 클래스이므로 가장 마지막에 위치
- 상위 클래스 예외를 먼저 catch하면 하위 클래스 예외는 도달 불가

사용자 정의 예외

```
public class MyException extends RuntimeException {
    public MyException(String msg) {
        super(msg);
    }

    public MyException(Exception ex) {
        super(ex);
    }
}
```

- Exception 클래스를 상속받는 클래스
- 문자열을 받아 부모인 Exception 클래스에 전달하는 생성자를 작성
- 필요시 선언한 예외 클래스를 객체 생성 후 강제로 예외를 발생

예외 처리 원칙

1. **구체적인 예외를 처리하라**
2. **예외를 무시하지 마라**
3. **리소스는 finally나 try-with-resources로 정리하라**

사용자 정의 예외 가이드라인

1. **Exception vs RuntimeException 상속 선택**
2. 복구 가능한 예외: Exception 상속
3. 프로그래밍 오류: RuntimeException 상속
4. **의미 있는 정보 포함**
5. 예외 발생 시점의 상태 정보 저장
6. 디버깅에 도움이 되는 컨텍스트 제공
7. **표준 예외 재사용 고려**
8. IllegalArgumentException, IllegalStateException 등 표준 예외 활용

try-with-resources

Java 7부터 도입된 기능으로, AutoCloseable 인터페이스를 구현한 리소스를 자동으로 해제해줍니다.

```
// Java 7 이후 권장 방법
try(FileInputStream fis = new FileInputStream("file.txt")) {
    // 파일 처리
} catch(IOException e) {
    // 예외 처리
}
// 자동으로 fis.close() 호출됨
```

예외 정보 얻기

- 문제를 해결하고 디버깅하는 데 필수적
- `getMessage()`: 예외에 대한 기본적인 설명 메시지를 반환
- `printStackTrace()`: 예외 발생 시의 호출 스택을 출력
- `getStackTrace()`: 예외가 발생한 위치의 상세한 정보를 제공

자바 IO

- 입력과 출력 관련(외부와의 소통)
- 세 가지 핵심 원칙(유연성, 확장성, 재사용성)
- **Decorator 패턴** 사용(조립되어서 사용되게 설계됨)
- Java IO를 이해하는 핵심은 **주인공**과 **장식**을 구분
- 입출력은 동시에 불가 따로 통로 필요

장식 (Decorator)

- InputStream, OutputStream, Reader, Writer
- 입출력을 하는 클래스
- 바이트단위 입/출 (InputStream, OutputStream): I/O Stream
- 문자열단위 입/출 (Reader, Writer): R/W
- **다양한 방식으로 읽고 쓰는 메소드** (`기능`)

주인공 (주 구성요소)

- **어떤 대상에게서 읽어들일지, 쓸지를 결정** (`위치`)
- **1byte 또는 byte[] 단위로 읽고 쓰는 메소드** I/O Stream
- **1char 또는 char[] 단위로 읽고 쓰는 메소드**를 가집니다 R/W

스트림(Stream)의 개념

- 데이터가 흐르는 통로
- 입출력 통로는 각각 존재해야함
- Java IO는 크게 4개의 추상 클래스를 중심으로 구성
  InputStream, **OutputStream**, **Reader**, **Writer**

스트림의 분류

데이터 단위에 따른 분류

- **바이트 스트림 (Byte Stream)**: 1바이트 단위로 처리
- **문자 스트림 (Character Stream)**: 2바이트(char) 단위로 처리

데이터 흐름에 따른 분류

- **입력 스트림 (Input Stream)**: 데이터를 읽어오는 스트림
- **출력 스트림 (Output Stream)**: 데이터를 내보내는 스트림