8. 예외 처리

예외 계층 구조 (`Exception`, `Error`, `RuntimeException`)

✅ 1. 예외(Exception)이란?

> 프로그램 실행 중 발생할 수 있는 **비정상적인 상황(에러)**을 객체로 표현한 것.
> 예외는 자바에서 **클래스로 정의되어 있고, 계층 구조를 가짐.**

```
Throwable
├── Error
└── Exception
     ├── RuntimeException
     └── (Checked Exception)
```

✅ 2. 최상위 클래스: `Throwable`

- 자바 예외 계층의 루트 클래스
- 예외 또는 오류가 발생했을 때 JVM이 생성하여 던지는 객체

```
public class Throwable implements Serializable {
    ...
}
```

`Throwable`은 **두 가지 하위 클래스**를 가진다:

1. `Error`
2. `Exception`

✅ 3. `Error`: 복구 불가능한 시스템 오류

> JVM 또는 시스템 레벨에서 발생하는 **심각한 문제**
> 보통 개발자가 **직접 처리하지 않음**

대표 예

클래스	설명
`OutOfMemoryError`	힙 메모리 부족
`StackOverflowError`	무한 재귀 호출
`InternalError`	JVM 내부 오류

```
throw new OutOfMemoryError("메모리 부족!");
```

> 일반적으로 try-catch로 잡지 않음. 처리 대상이 아님.

✅ 4. `Exception`: 복구 가능한 예외

> **개발자가 예측하여 처리할 수 있는 오류 상황**

두 분류로 나뉜다:

1. ✔ Checked Exception

- **컴파일러가 강제**하는 예외 처리 필요
- `try-catch` 또는 `throws`로 반드시 처리해야 함

예시 클래스	설명
`IOException`	파일 입출력 오류
`SQLException`	데이터베이스 접속 실패
`ParseException`	날짜 파싱 오류

```
try {
    FileReader fr = new FileReader("file.txt");
} catch (FileNotFoundException e) {
    e.printStackTrace();
}
```

---

2. ✗ Unchecked Exception (`RuntimeException` 하위)

- **컴파일러가 강제하지 않음**
- 개발자 실수에 의한 **논리 오류 또는 프로그래밍 오류**

예시 클래스	설명
`NullPointerException`	null 접근
`ArrayIndexOutOfBoundsException`	배열 범위 초과
`ArithmeticException`	0으로 나눔
`IllegalArgumentException`	잘못된 인자 전달

```
int[] arr = new int[3];
System.out.println(arr[10]); // ArrayIndexOutOfBoundsException
```

> 이들은 **try-catch 없이도 컴파일 가능**,
> 필요에 따라 런타임에서만 처리

✅ 5. 예외 계층 구조 시각화

```
Throwable
├── Error                        ← 시스템 레벨 오류 (복구 불가)
|    ├── OutOfMemoryError
|    └── StackOverflowError
|
└── Exception                    ← 복구 가능 예외
     ├── IOException             ← Checked Exception
     ├── SQLException
     ├── ...
     └── RuntimeException        ← Unchecked Exception
          ├── NullPointerException
          ├── IllegalArgumentException
          ├── ArrayIndexOutOfBoundsException
          └── ...
```

✅ 6. 사용자 정의 예외 만들기

```
class MyException extends Exception {
    public MyException(String message) {
        super(message);
    }
}
```

```
throw new MyException("사용자 정의 오류 발생!");
```

> 필요에 따라 Checked 또는 Unchecked로 상속 계열 결정 가능

✅ 7. Checked vs Unchecked 정리

항목	Checked Exception	Unchecked Exception (RuntimeException)
컴파일 시 처리 강제	✅ 예	❌ 아니오
예시	IOException, SQLException	NullPointerException, IllegalArgumentException
처리 대상	**예측 가능한 외부 문제** (네트워크, 파일 등)	**프로그래밍 실수** (로직 오류)
상속	`Exception`	`RuntimeException`

✅ 8. 예외 계층 설계 전략

- 외부 시스템 관련 오류는 `Checked Exception` → 강제 처리
- 내부 로직 오류는 `RuntimeException` → 필요 시 처리
- 공통 예외 추상 클래스 → 세분화된 하위 예외로 구분

✅ 9. 예외 클래스 선언 팁

```
// Checked 예외로 만들기
class FileReadException extends Exception { ... }

// Unchecked 예외로 만들기
class InvalidUserInputException extends RuntimeException { ... }
```

✅ 10. 요약

계층	설명	예시	처리 방식
`Error`	시스템 에러	OutOfMemoryError	처리 ❌
`Exception`	애플리케이션 예외	IOException	처리 ⭕
`RuntimeException`	프로그래밍 오류	NullPointerException	선택적 처리

`try-catch-finally` 구조

✅ 1. 예외 처리란?

> 프로그램 실행 중 예외 상황이 발생했을 때
> **프로그램의 비정상 종료를 방지하고**
> **안정적으로 복구하거나 종료할 수 있도록 처리하는 구조**

✅ 2. 기본 구조

```
try {
    // 예외가 발생할 수 있는 코드
} catch (예외타입 변수명) {
    // 예외가 발생했을 때 처리할 코드
} finally {
    // 예외 발생 여부와 상관없이 무조건 실행되는 코드
}
```

- `try` : 예외 발생 가능성이 있는 코드 블록
- `catch` : 예외가 발생했을 때 처리 로직
- `finally` : 예외 발생 여부와 관계없이 **항상 실행됨** (리소스 해제 용도)

✅ 3. 기본 예제

```
public class TryCatchExample {
    public static void main(String[] args) {
        try {
            int result = 10 / 0;
            System.out.println("결과: " + result);
        } catch (ArithmeticException e) {
            System.out.println("예외 발생: " + e.getMessage());
        } finally {
            System.out.println("항상 실행되는 블록");
        }
    }
}
```

출력:

```
예외 발생: / by zero
항상 실행되는 블록
```

✅ 4. `catch` 블록 여러 개 사용하기

```
try {
    String s = null;
    System.out.println(s.length());
} catch (NullPointerException e) {
    System.out.println("널 포인터 예외");
} catch (Exception e) {
    System.out.println("기타 예외");
}
```

> **특정 → 일반 순서로 작성해야 함**
> `RuntimeException` → `Exception` 순서로
> (그렇지 않으면 컴파일 에러)

✅ 5. 다중 예외 통합 처리 (`|` 연산자)

```
try {
    int[] arr = new int[3];
    System.out.println(arr[5]);
} catch (NullPointerException | ArrayIndexOutOfBoundsException e) {
    System.out.println("배열 관련 예외");
}
```

- Java 7 이상
- **다른 타입의 예외를 한 번에 처리 가능**

- 단, 예외 객체(`e`)는 `final`처럼 작동 → `e = new ...` 불가

✅ 6. `finally` 블록의 역할

> `try` 또는 `catch` 블록의 실행 결과와 상관없이
> **무조건 실행되는 블록**

사용 목적:

- 파일/DB/네트워크 **리소스 해제**
- **로그 출력**, 정리 작업 등

```
finally {
    System.out.println("파일 닫기, 연결 해제 등 정리 작업");
}
```

✅ 7. 예외 발생 없이도 finally 실행됨

```
try {
    System.out.println("정상 실행");
} finally {
    System.out.println("무조건 실행");
}
```

✅ 8. `return`과 `finally`

```
public static int test() {
    try {
        return 1;
    } finally {
        System.out.println("finally 실행");
    }
}
```

> `finally`는 `return`**보다 나중에 실행됨**
> 하지만 반환 값은 `try`의 값

✅ 9. finally 생략 가능한가?

- `try-catch` 또는 `try-finally` → 둘 중 하나만 있어도 OK
- 하지만 `try` **단독 사용은 불가**

```
try {
    // 불가능 (컴파일 에러)
}
```

✅ 10. try-with-resources (Java 7+)

> 리소스를 자동으로 닫아주는 구조

```
try (BufferedReader br = new BufferedReader(new FileReader("file.txt"))) {
    System.out.println(br.readLine());
} catch (IOException e) {
    e.printStackTrace();
}
```

> `AutoCloseable` 또는 `Closeable`을 구현한 객체는 자동으로 `.close()` 호출됨

✅ 11. 예외 흐름 요약

상황	흐름
예외 없음	`try` → `finally`
예외 발생 + catch 존재	`try` → `catch` → `finally`
예외 발생 + catch 없음	`try` → `finally` → 호출자에게 예외 전파
`return` 존재	`try` → `finally` → return

✅ 12. 예외 처리 전략

- 구체적인 예외부터 먼저 catch
- 반드시 필요한 경우만 catch
- 리소스 해제는 반드시 finally 또는 try-with-resources 사용
- 예외 발생 시 의미 있는 메시지와 함께 로깅
- 복구 불가능한 예외는 **전파** (`throws`)도 고려

✅ 13. 요약

블록	역할
`try`	예외 발생 가능 코드 실행
`catch`	예외 처리
`finally`	항상 실행 (정리, 리소스 해제)

블록	역할
`throws`	예외를 호출자에게 전파
`throw`	예외 직접 발생

`throw`, `throws` 키워드

✅ 1. 핵심 차이 요약

항목	`throw`	`throws`
의미	**예외를 발생시키는 명령어**	**예외를 던질 수 있다고 선언**
위치	**메서드 내부에서 사용**	**메서드 선언부에 사용**
목적	특정 조건에서 예외 객체를 명시적으로 발생시킴	호출자에게 예외를 **전달(전파)**
뒤에 오는 것	예외 객체 (new로 생성한 것)	예외 타입 이름
예제	`throw new IllegalArgumentException()`	`void myMethod() throws IOException`

✅ 2. `throw` 키워드

> 예외를 **직접 발생시킬 때 사용**
> 반드시 `Throwable` (또는 하위 클래스) **객체를 명시**해야 함

🔹 문법

```
throw new 예외객체;
```

🔹 예제

```
public void validateAge(int age) {
    if (age < 0) {
        throw new IllegalArgumentException("나이는 음수일 수 없음");
    }
}
```

- `IllegalArgumentException`은 `RuntimeException`의 하위 → throws 선언 없이 가능

✅ 3. `throws` 키워드

> 메서드에서 발생할 수 있는 **예외를 호출자에게 알리는 선언**
> 즉, **"나 예외 던질 수 있어!"** 라는 **예외 전달 선언**

🔹 문법

```
리턴타입 메서드명() throws 예외타입1, 예외타입2 {
    ...
}
```

🔹 예제

```
public void readFile() throws IOException {
    FileReader fr = new FileReader("a.txt");
}
```

- `IOException`은 **Checked Exception** → 처리 또는 `throws` 필요

✅ 4. `throw` + `throws` 함께 사용

```
public void doSomething() throws MyException {
    throw new MyException("직접 던진 예외");
}
```

> `throw`로 던질 수 있는 예외가 **Checked Exception일 경우**,
> 반드시 `throws`로 선언 필요

✅ 5. `throws`로 선언한 예외는 호출자가 처리해야 함

```
public void run() throws IOException {
    throw new IOException("파일 없음");
}

public void test() {
    try {
        run();
    } catch (IOException e) {
        System.out.println("예외 처리 완료");
    }
}
```

✅ 6. `throws`는 여러 개 선언 가능 (쉼표로 구분)

```
public void process() throws IOException, SQLException {
    ...
}
```

✅ 7. 사용자 정의 예외와 throw/throws

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

public void check(boolean flag) throws MyException {
    if (!flag) {
        throw new MyException("조건 불만족");
    }
}
```

- Checked Exception이므로 `throws` 선언 필수

✅ 8. 예외 전파 흐름 이해

```
main()
 ↳ methodA() throws IOException
     ↳ methodB() throws IOException
         ↳ throw new IOException()

=> main()에서 try-catch 또는 다시 throws
```

✅ 9. 주의할 점

항목	설명
`throw`는 객체를 던짐	`throw new 예외()` 필수
`throws`는 예외 타입 명시	클래스 이름만 적음 (`IOException`)
`throw` 후에는 코드 실행 안됨	무조건 메서드 종료
`throws`는 선언만 함	실제 발생은 `throw`로 해야 함

✅ 10. 예외 흐름 예제 전체

```
class CustomException extends Exception {
    public CustomException(String msg) {
        super(msg);
    }
}

public class ThrowThrowsDemo {

```

```
    public static void riskyMethod() throws CustomException {
        throw new CustomException("문제 발생");
    }

    public static void main(String[] args) {
        try {
            riskyMethod();
        } catch (CustomException e) {
            System.out.println("예외 처리: " + e.getMessage());
        }
    }
}
```

✅ 11. 요약 정리

키워드	목적	위치	뒤에 오는 것	처리 의무
`throw`	예외 발생	메서드 내부	예외 객체	즉시 발생
`throws`	예외 전파	메서드 선언부	예외 클래스명	호출자가 처리

사용자 정의 예외

✅ 1. 사용자 정의 예외란?

> Java에서 제공하는 표준 예외 외에,
> **도메인 또는 비즈니스에 맞는 의미 있는 예외 상황**을
> 직접 정의한 클래스를 통해 처리하는 방식

예:

- `InvalidUserInputException`
- `OutOfStockException`
- `PermissionDeniedException`

✅ 2. 예외 클래스 만들기 – 기본 구조

```
public class MyException extends Exception {
    public MyException(String message) {
        super(message);
    }
}
```

- 예외 메시지 전달을 위해 `super(message)` 호출
- `Exception` 또는 `RuntimeException`을 상속받음

✅ 3. Checked vs Unchecked 사용자 정의 예외

예외 유형	상속 대상	예시	처리 방식
Checked	`Exception`	`FileReadException`	반드시 `throws` or `try-catch`
Unchecked	`RuntimeException`	`InvalidInputException`	선택적 처리

✔ Checked Exception 예제

```
public class FileReadException extends Exception {
    public FileReadException(String message) {
        super(message);
    }
}
```

```
public void readFile(String path) throws FileReadException {
    if (path == null) {
        throw new FileReadException("파일 경로가 null입니다.");
    }
}
```

✔ Unchecked Exception 예제

```
public class InvalidInputException extends RuntimeException {
    public InvalidInputException(String message) {
        super(message);
    }
}
```

```
public void validateAge(int age) {
    if (age < 0) {
        throw new InvalidInputException("나이는 음수일 수 없습니다.");
    }
}
```

✅ 4. 생성자 오버로딩: 원인 추적용

```
public class BusinessException extends Exception {
    public BusinessException(String message) {
        super(message);
    }
    public BusinessException(String message, Throwable cause) {
        super(message, cause);  // 예외 체인 (stack trace 추적)
    }
}
```

> 실무에서는 `Throwable cause`를 함께 제공해
> **예외 전파 흐름 추적**에 유용하게 사용

✅ 5. 사용 예시 – 도메인 예외 설계

```
public class OutOfStockException extends RuntimeException {
    public OutOfStockException(String item) {
        super("재고 부족: " + item);
    }
}
```

```
public void orderItem(String item, int stock) {
    if (stock <= 0) {
        throw new OutOfStockException(item);
    }
}
```

✅ 6. 계층 구조 설계 전략

> 예외들도 **도메인별로 계층화**하면 유지보수에 좋음

```
AppException (추상)
├── UserException
|   ├── UserNotFoundException
|   └── InvalidPasswordException
└── OrderException
    ├── OutOfStockException
    └── PaymentFailedException
```

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

✅ 7. 예외 처리 시 메시지 커스터마이징

```
throw new UserNotFoundException("ID: " + userId + " not found");
```

- 에러 로그에서 바로 원인 파악 가능
- 프론트에 넘길 때도 명확한 피드백 제공

✅ 8. 실무 팁 및 전략

전략	설명
의미 있는 이름 부여	`InvalidInputException`, `PermissionException`
계층 구조 설계	공통 상위 예외로 그룹화
`Serializable` 구현	분산 환경에서도 안전하게 사용 가능
로깅과 함께 사용	`logger.error("...", ex)`
API 응답 매핑	예외 → HTTP 상태 코드로 변환 (Spring 등에서 사용)

✅ 9. 사용자 정의 예외 + 예외 처리 예제 (실전 스타일)

```
class InsufficientBalanceException extends RuntimeException {
    public InsufficientBalanceException(double balance) {
        super("잔액 부족: 현재 잔액은 " + balance + "원입니다.");
    }
}

public class BankAccount {
    private double balance = 1000;

    public void withdraw(double amount) {
        if (balance < amount) {
            throw new InsufficientBalanceException(balance);
        }
        balance -= amount;
    }
}
```

✅ 10. 요약 정리

항목	설명
정의	`Exception` 또는 `RuntimeException`을 상속
Checked	`Exception` 상속, 반드시 처리
Unchecked	`RuntimeException` 상속, 선택적 처리
생성자	메시지, cause 인자 포함 가능
계층화	도메인 예외로 그룹화 시 유지보수 향상