4장 예외

4.1 사라진 SQLException

4.1.1 초난감 예외처리

- 예외 블랙홀

```
try{...}cathch(SQLException e){}
// 예외를 잡고는 아무것도 하지 않는다. 예외 발생을 무시해버리고 정상적인 상황인 것처럼 다음 라인으로 넘어가겠다는
// 분명한 의도가 있는게 아니라면 연습중에도 절대 만들어서는 안되는 코드다.
```

예외가 발생하면 그것은 catch블록을 써서 잡아내는 것까지는 좋은데 아무것도 하지 않고 별 문제 없는 것처럼 넘어가 버리는것은 정말 위험한 일이다. 예외가 발생했는데 그것을 무시하고 계속 진행해버리기 때문이다.

이는 시스템 오류나 이상한 결과의 원인이 무엇인지 찾아내기 매우 어려운 더 큰 문제를 야기한다.

```
}catch(SQLExcpetion e) {
  System.out.println(e); // e.printStackTrace();
}
```

출력해줬다고 예외를 처리한 게 아니다.

모든 예외는 적절하게 복구되든지 아니면 작업을 중단시키고 운영자 또는 개발자에게 분명하게 통보돼야 한다.

```
}catch(SQLExcpetion e) {
  e.printStackTrace();
  System.exit(1); //그나마 나은 예외처리
}
```

예외를 무시하거나 잡아먹어 버리는 코드는 만들지 말자.

- 무의미하고 무책임한 throws
  - throw Exception은 의미 있는 정보를 얻을 수 없다.

- 적절한 처리를 통해 복구될 수 있는 예외상황도 제대로 다룰 수 있는 기회를 박탈당한다.

4.1.2 예외의 종류와 특징

- Error
  - java.lang.Error
  - 시스템에 비정상적인 상황이 발생했을 경우에 사용
  - 주로 자바 VM에서 발생시키는 것이기 때문에 애플리케이션 코드에서 잡으려고 하면 안된다.
  - 에러 처리를 신경 쓰지 않아도 된다.
- Expcetion 과 체크 예외
  - java.lang.Exception

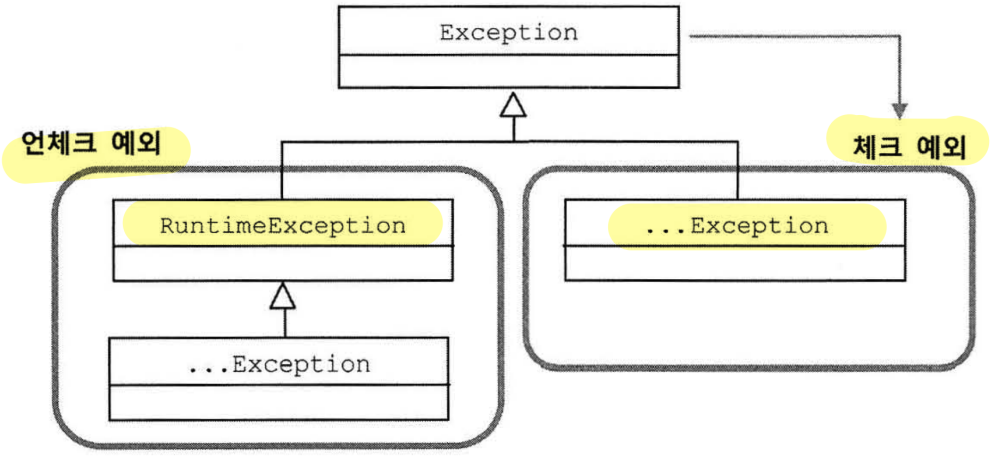

그림 4-1 Exception의 두 가지 종류

  - 개발자들이 만든 애플리케이션 코드의 작업 중에 예외상황이 발생했을 경우에 사용
- RuntimeException과 언체크/런타임 예외
  - java.lang.**RuntimeException**
  - 명시적인 예외처리를 강제하지 않기 때문에 **언체크 예외**라고 불린다.
  - 주로 프로그램의 오류가 있을 때 발생하도록 의도된 것들이다.

        - NullPointerException, IllegalArgumentExcpetion...
    - 코드에서 미리 조건을 체크하도록 주의 깊게 만든다면 피할수 있다. 개발자가 부주의 해서 발생할 수 있는 경우에 발생하도록 만든 것이 런타임 예외다. 때문에 굳이 catch나 throws를 사용하지 않아도 되도록 만든 것이다.

4.1.3 예외처리방법

- 예외 복구
    - 예외상황을 파악하고 문제를 해결해서 정상 상태로 돌려놓는 것

    ▼ 예시

    1. 사용자가 요청한 파일을 읽으려 시도했을 때 해당 파일이 없거나 다른 문제가 있어 읽히지 않아 IOException이 발생했다
    2. 사용자에게 상황을 알려주고 다른 파일을 이용하도록 안내해 예외상황 해결

       다른 작업 흐름으로 자연스럽게 유도해주는 것

        - IOException 에러 메세지가 사용자에게 그냥 던져지는 것은 예외 복구라 볼 수 없다.
    - 예외 처리 코드를 강제하는 체크 예외들은 예외를 어떤식으로든 복수할 가능성이 있는 경우에 사용한다.

```
int maxRetry = MAX_RETRY;
while(maxRetry -- > 0) {
  try{ ... // 예외가 발생할 가능성이 있는 시도
    return ; // 작업 성공
  }
  catch(SomeException e) {
    //로그 출력, 정해진 시간만큼 대기
  }finally {
    // 리소스 반납, 정리 작업
  }throw new RetryFailedExcpetion();
//최대 재시도 횟수를 넘기면 직접 예외 발생
```

- 예외처리 회피
    - 예외처리를 자신이 담당하지않고 자신을 호출한 쪽으로 던져버리는 것

```
public void add() throws SQLException {
    // JDBC API
}
```

    - 콜백 오브젝트의 메소드는 SQLException에 대한 예외를 회피하고 템플릿 레벨에서 처리하도록 던져준다.
    - 콜백과 템플릿처럼 긴밀하게 역할을 분담하고 있는 관계가 아니라면 자신의 코드에서 발생하는 예외를 그냥 던져버리는 건 무책임한 책임회피일 수 있다.
    - 예외를 회피하는 것은 예외를 복구하는 것처럼 의도가 분명해야 한다. 콜백/템플릿 처럼 긴밀한 관계에 있는 다른 오브젝트에게 예외처리 책임을 지게 하거나, 자신을 사용하는 쪽에서 에외를 다루는게 최선의 방법이라는 분명한 확신이 있어야 한다.
- 예외 전환
    - 적절한 예외로 전환해서 예외를 메소드 밖으로 던지는 것
    - 내부에서 발생한 예외를 그대로 던지는 것이 그 예외 상황에 대한 적절한 의미를 부여해주지 못하는 경우에 의미를 분명하게 해줄 수 있는 예외로 바꿔주기 위해
        - 보통 전환하는 예외에 원래 발생한 예외를 담아서 중첩 예외로 만드는 것이 좋다
    - 예외를 처리하기 쉽고 단순하게 만들기 위해 포장하는 것
        - 중첩 예외를 이용행 새로운 예외를 만들고 원인이 되는 예외를 내부에 담아 던지는 방식은 같다.
        - 예외처리를 강제하는 체크 예외를 언체크 예외인 런타임 예외로 바꾸는 경우에 사용
        - 대표적으로 EJBException
        - 런타임 예외로 만들어서 전달하면 EJB는 시스템 익셉션으로 인식하고 트랜잭션을 자동으로 롤백해준다. 런타임 예외기 때문에 EJB 컴포넌트를 사용하는 다른 EJB나 클라이언트에 예외를 잡아도 복구할 만한 방법이 없기 때문에 예외를 던질 필요가 없다.
        - 로직상에서 예외조건이 발견되거나 예외상황이 발생할 수도 있다. 이때는 애플리케이션 코드에서 의도적으로 던지는 예외기 때문에 체크 예외를 사용하는

것이 적절하다. 비지니스적인 의미가 있는 예외는 이에 대한 적절한 대응이나 복구작업이 필요하다.

- 일반적으로 체크 예외를 계속 throws를 사용해 넘기는건 무의미하다. 복구가 불가능한 예외라면 가능한 빨리 런타임 예외로 포장해 던지게 해서 다른 계층의 메소드를 작성할 때 불필요한 throws 선언이 들어가지 않도록 해줘야 한다.
- 예외 처리 서비스 등을 통해 자세한 로그를 남기고, 관리자에게는 메일 등으로 통보해주고, 사용자에게는 친절한 안내 메시지를 보여주는 식으로 처리하는게 바람직하다.

4.1.4 예외처리 전략

- 런타임 예외의 보편화
  - 자바 엔터프라이즈 서버환경에서, 예외상황을 미리 파악하고 예외가 발생하지 않도록 차단하는 게 좋다.
  - 요즘엔 항상 복구할 수 있는 예외가 아니라면 일단 언체크 예외로 만드는 경향이 있다.
- add()메소드의 예외처리

```
public class DuplicateUserIdExcpetion extends RuntimeExcpetion {
  public DuplicateUserIdExcpetion(Throwable cause) {
    super(cause);
  }
}
```

- 애플리케이션 예외
  - 애플리케이션 자체의 로직에 의해 의도적으로 발생시키고 반드시 catch해서 무엇인가 조치를 취하도록 요구하는 예외
  - 사용자가 요청한 금액을 은행 계좌에서 출금하는 기능을 가진 메소드가 있다. 출금을 허용하고 현재 잔고가 얼마인지 상관없이 요청한 금액만큼 계좌 잔액을 차감하도록 만드는 개발자는 없을 것이다. 현재 잔고를 확인하고, 허용하는 범위를 넘어 출금을 요청하면 출금 작업을 중단시키고 적절한 경고를 사용자에게 보내야 한다.
    1. 정상적인 출금처리를 햇을 경우와 잔고 부족이 발생했을 경우에 각각 다른 종류의 리턴 값을 돌려준다. ⇒ 예외상황에 대한 리턴 값을 명확하게 코드

화 하고 관리해야지 혼란이 안온다.

2. 정상적인 흐름을 따르는 코드는 그대로 두고, 잔고 부족과 같은 예외 상황에서는 비지니스 적인 의미를 띤 예외를 던지도록 한다.
   InsufficientBalanceException과 같이 의도적으로 체크 예외로 만든다. 그래서 개발자가 잊지 않고 잔고 부족과 같은 예외상황에 대한 로직을 구현하도록 강제해주는게 좋다.

리스트 4-15 애플리케이션 예외를 사용한 코드

```
try {
    BigDecimal balance = account.withdraw(amount);
    ...
    // 정상적인 처리 결과를 출력하도록 진행
}
catch(InsufficientBalanceException e) {        // 체크 예외
    // InsufficientBalanceException에 담긴 인출 가능한 잔고금액 정보를 가져옴
    BigDecimal availFunds = e.getAvailFunds();
    ...
    // 잔고 부족 안내 메시지를 준비하고 이를 출력하도록 진행
}
```

4.1.5 SQLException은 어떻게 됐나?

- SQLExcpetion은 복구 가능한 예외인가? → 복구 불가.
  - 입력단계에서 검증을 강화한다는 사실을 개발자가 빨리 인식할 수 있도록 발생한 예외를 빨리 전달하는 것 외에는 할 수 있는 게 없다.

  JdbcTemplate는 모든 SQLException을 런타임 예외인 `DataAccessException`으로 포장해서 던져준다.

  UserDao에서는 꼭 필요한 경우에만 런타임 예외인 DataAccessExcpetion을 잡아 처리하면되고 나머지는 무시해도 되기 때문에 DAO메소드에서 SQLExcpetion이 사라졌다.

스프링 API 메소드에 정의되어 있는 대부분의 예외는 런타임 예외다. 따라서 발생 가능한 예외가 있다고 해도 이를 처리하도록 강제하지 않는다.

4.2 예외 전환

4.2.1 JDBC의 한계

JDBC는 자바를 이용해 DB에 접근하는 방법을 추상화된 API 형태로 정의해놓고, 각 DB 업체가 JDBC 표준을 따라 만들어진 드라이버를 제공하게 해준다.

하지만 DB 종류에 상관없이 사용할 수 잇는 데이터 액세스 코드를 작성하는 일은 쉽지 않다.

- 비표준 SQL

  첫번째 문제는 JDBC 코드에서 사용하는 SQL이다. DB마다 비표준 문법의 기능도 제공하기 때문에 이런 비표준 SQL 은 결국 DAO 코드에 들어가고, 해당 DAO는 특정 DB에 대해 종속적인 코드가 되고만다. 다른 DB로 변경하려면 DAO에 담긴 SQL을 적지 않게 수정해야 한다.

  DAO를 DB별로 만들어 사용하거나 SQL을 외부에서 독립시켜 바꿔 쓸 수 있게 하는 방법이 제일 현실적이다.

- 호환성 없는 SQLException의 DB 정보

  두번째 문제는 SQLException이다. DB를 사용하다가 발생할 수 있는 예외의 원인은 다양하다. DB마다 SQL만 다른게 아니라 에러의 종류도 다르다.

  SQLExcpetion은 예외가 발생했을 때 DB 상태를 담은 SQL 상태 정보를 부가적으로 제공한다. `getSQLState()` 메소드로 예외상황에 대한 상태정보를 가져올 수 있다. Open Group의 XOPEN SQL스펙에 정의된 SQL상태 코드를 따르도록 되어 있다.

  DB에 독립적인 에러정보를 얻기위해 이런 상태 코드를 제공하는데 JDBC 드라이버에서 이를 정확히 만들어주지 않는다. 결과적으로 SQL 상태 코드를 믿고 결과를 파악하도록 코드를 작성하는 것은 위험하다.

4.2.2 DB 에러 코드 매핑을 통한 전환

- DB별 에러 코드를 참고해서 발생한 예외의 원인이 무엇인지 해석해주는 기능을 만드는 것이다.
- 스프링은 DataAccessExpcetion의 서브 클래스로 세분화된 예외 클래스들을 정의한다.
  - BadSqlGrammerExcpetion
  - DataAccessResourceFailureException
  - DataIntegrittViolationException
  - DuplicatedKeyException 등등..

- 문제는 DB마다 에러 코드가 제각각이다.
- 스프링은 DB별 에러 코드를 분류해서 스프링이 정의한 예외 클래스와 매핑해놓은 에러코드 매핑 정보 테이블을 만들어두고 이를 이용한다.
- JdbcTemplate을 이용한다면 JDBC에서 발생하는 DB 관련 예외는 거의 신경 쓰지 않아도 된다.
- 직접 정의한 예외를 발생시키고 싶을 때는 스프링의 ~Exception예외를 전환해주는 코드를 DAO안에 넣으면 된다.

리스트 4-18 중복 키 예외의 전환

```
public void add() throws DuplicateUserIdException {
    try {                                          → 애플리케이션 레벨의 체크 예외
        // jdbcTemplate을 이용해 User를 add 하는 코드
    }
    catch(DuplicateKeyException e) {
        // 로그를 남기는 등의 필요한 작업
        throw new DuplicateUserIdException(e);  → 예외를 전환할 때는 원인이 되는 예외를 중첩하는 것이 좋다.
    }
}
```

- SQLExcpetion의 서브 클래스이므로 여전히 체크 예외라는 점, 그 예외를 세분화하는 기준이 SQL 상태정보를 이용한다는 점에서 여전히 문제점이 있다.
- 스프링의 에러 코드 매핑을 통한 DataAccessException 방식을 사용하는 것이 이상적이다.

4.2.3 DAO 인터페이스와 DataAccessExcpetion 계층 구조

스프링은 DataAccessException계층구조를 이용해 기술에 독립적인 예외를 정의하고 사용하게 한다.

- DAO 인터페이스와 구현의 분리
  - DAO사용 이유 → 데이터 엑세스 로직을 담은 코드를 성격이 다른 코드에서 분리해놓기 위해서, 분리된 DAO는 전략패턴을 적용해 구현 방법을 변경해서 사용할 수 있게 만들기 위해서
  - 인터페이스로 메소드의 구현은 추상화햇지만 구현 기술마다 던지는 예외가 다르면 메소드의 선언이 달라지는 문제가 발생한다

- 데이터 액세스 예외 추상화와 DataAccessExcpetion 계층 구조
  - 스프링은 자바의 다양한 데이터 액세스 기술을 사용할 때 발생하는 예외들을 추상화해 DataAccessExcpetion 계층 구조 안에 정리했다.
    - JDO,JPA,하이버네이트 처럼 오브젝트/엔티티 단위로 정보를 업데이트 하는 경우에는 낙관적인 락킹이 발생할 수 있는데, 이 또한 다른 종류의 낙관적 락킹 예외를 발생시킨다.
  - JdbcTemplate와 같이 스프링의 데이터 액세스 지원 기술을 이용해 DAO를 만들면 사용 기술에 독립적인 일관성 있는 예외를 던질 수 있다.

4.2.4 기술에 독립적인 DAO 만들기

- 인터페이스 적용
- 테스트 보완
  - 구현 기술에 상관없이 DAO의 기능이 동작하는 데만 관심이 있다면 UsrDao인터페이스로 받아서 테스트 하는 편이 낮다. 나중에 DAO 빈을 변경하더라도 테스트는 유효하다.
  - 특정 기술을 사용한 구현 내용에 관심을 가지고 테스트 하려면 DI 받을 때 UserDaoJdbc나 UserDaoHibernate같이 특정 타입을 사용한다
- DataAccessException 활용 시 주의사항

  SQLExcpetion에 담긴 DB에러 코드를 바로 해석하는 JDBC와 달리 JPA나 하이버네이트, JDO 등에서는 각 기술이 재정의한 예외를 가져와 스프링이 최종적으로 DataAccessException으로 변환하는데 DB의 에러 코드와 달리 이런 예외들은 세분화되어 있지 않기 때문에 실제로 다른 예외가 던져진다.

4.3 정리

1. 예외를 잡아서 아무런 조치를 취하지 않거나 의미 없는 throws 선언을 남발하는 것은 위험하다.
2. 예외는 복구하거나 예외처리 오브젝트로 의도적으로 전달하거나 적절한 예외로 전환해야한다.
3. 좀 더 의미 있는 예외로 변경하거나 불필요한 catch/throws를 피하기 위해 런타임 예외로 포장하는 두 가지 방법의 예외 전환이 있다.

4. 복구할 수 없는 예외는 가능한 빨리 런타임 예외로 전환하는 것이 바람직하다.

5. 애플리케이션의 로직을 담기 위한 예외는 체크 예외로 만든다.

6. JDBC의 SQLException의 에러코드는 DB에 종속되기 때문에 DB에 독립적인 예외로 전환될 필요가 있다.

7. 스프링은 DataAccessExcpetion을 통해 DB에 독립적으로 적용 가능한 추상화된 런타임 예외 계층을 제공한다.

8. DAO를 데이터 액세스 기술에서 독립시키려면 인터페이스 도입과 런타임 예외 전환, 기술에 독립적인 추상화된 예외로 전환이 필요하다.