체크 예외(checked exception)

자바의 예외의 종류 세 가지

* Error

Java.lang.Error클래스의 서브클래스들이다. 에러는 시스템에 뭔가 비정상적인 상황이 발생했을 경우에 사용된다. 주로 자바 VM에서 발생시키는 것이고 애플리케이션 코드에서 잡아봤자 아무런 대응 방법이 없기 때문이다. 따라서 시스템 레벨에서 특별한 작업을 하는 게 아니라면 이 예외에 대한 신경은 쓰지 않아도 된다.

* Exception과 체크 예외

Java.lang.Exception과 그 서브클래스로 정의되는 예외들은 에러와 달리 개발자들이 만든 애플리케이션 코드 작업 중에 예외 상황이 발생했을 경우에 사용된다.

Exception 클래스는 다시 체크예외와 언체크 예외로 구분된다. 전자는 Exception 클래스의 서브 클래스이면서 RuntimeException 클래스를 상속하지 않은 것들이고, 후자는 상속한 클래스를 말한다. 사용할 메소드가 체크 예외를 던진다면 이를 .catch문으로 잡든 지 throws를 정의해서 메소드 밖으로 던져야 한다. 그렇지 않으면 컴파일 에러가 발생한다.

* RuntimeException과 언체크/런타임 예외

Java.lang.RuntimeException 클래스를 상속한 예외들은 명시적인 강제를 하지 않기 때문에 언체크 예외라고 불린다. 또는 런타임 예외라고도 한다. 에러와 마찬가지로 이 런타임 예외는 catch문으로 잡거나 throws로 선언하지 않아도 된다. 물론 명시적으로 잡거나 throws로 선언해줘도 상관없다. 런타임 예외는 주로 프로그램의 오류가 있을 때 발생하도록 의도된 것들이다. 대표적으로 오브젝트를 할당하지 않은 레퍼런스 변수를 사용하려고 시도했을 때 발생하는 nullpointerexception이나 , 허용되지 않는 값을 사ㅛㅇ해서 메소드를 호출할 때 발생하는 UllegalArgumentException 등이 있다. 이런 예외는 코드에서 미리 조건을 체크하도록 주의 깊게 만든다면 피할 수잇다. 피할수 있지만 개발자가 부주의해서 발생할 수 있는 경우에 발생하도록 만든 것이 런타임 예외이다.

예외 처리 방법

* 예외 복구

예외 상황을 파악하고 문제를 해결해서 정상 상태로 돌려놓는 것이다.

예외로 인해 기본 작업흐름이 불가능하면 다른 작업 흐름으로 안내해서 예외 상황을 해결할 수 있다.

* 예외처리 회피

예외처리를 자신이 담당하지 않고 자신을 호출한 쪽으로 던져버리느 것이다. Throws 문으로 선언해서 예외를 발생하면 알아서 던져지게 하거나 catch문으로 일단 예외를 잡은 후에 로그를 남기고 다시 예외를 던지는 것이다.

예외를 자신이 처리하지 않고 회피하는 방법이다. 빈 catch 블록을 잡아서 예외가 발생하지 않은 것처럼 만드는 경우는 극히 드물지만 특별한 의도를 가지고 예외를 복구했거나 아무 개념인 없어서 그런 것이지 회피한 것은 아니라 예외처리를 회피하려면 반드시 다른 오브젝트나 메소드가 예외를 대신 처리할 수 있도록 던져주어야 한다.

예외를 회피하는 것은 예외를 복구하는 것처럼 의도가 분명해야 한다. 콜백/템플릿처럼 긴밀한 관계에 있는 다른 오브젝트에게 예외처리 책임을 분명히 지게 하거나, 자신을 사용하는 쪽에서 예외를 다루는 것이 최선의 방법이라는 분명한 확신이 있어야 한다.

* 예외 전환(Exception Translation)

예외 회피와 비슷하게 예외를 복구해서 정상적인 상태로는 만들 수 없기 때문에 예외를 메소드 밖으로 던지는 것이다. 하지만 예외 회피와는 달리, 발생한 예외를 그대로 넘기는 것이 아니라 적절한 예외로 전환해서 던진다는 특징이 있다.

P288

서비스 계층 오브젝트에서 sqlexception의 원인을 해석해서 대응하는 것도 불가능하지는

않지만 비즈니스 로직이 잇는 서비스 계층에서, 특정 기술의 정보를 해석하는 코드를

두는 것은 매우 어색하다.

* 보통 전환하는 예외에 원래 발생한 예외를 담아서 중첩 예외(nested exception)으로 만드는 것이 좋다

Ex) throw DuplicateUserIdException(e) --🡪

throw DuplicateUserIdException().initCause(e);

* 두 번째 전환방법은 예외를 처리하기 쉽고 단순하게 만들기 위해 포장(wrap)하는 것이다. 중첩 예외를 통해 새로운 예외를 만들고 원인이 되는 예외를 내부에 담아서 던지는 방식은 같다. 하지만 의미를 명확하게 하려고 다른 예외로 전환하는 것이 아니다. 주로 예외처리를 강제하는 체크 예외를 언체크 예외인 런타임 예외로 바꾸는 경우에 사용한다.

Ex) catch(NamingException ne){

Throw new EJBException(ne);

}

* 복구가 불가능한 예외라면 가능한 빨리 런타임예외로 포장하여 던지게 하여 다른 계층의 메소드를 작성할 때 불필요한 throws의 선언이 들어가지 않도록 해줘야 한다.

예외처리 전략

* 런타임 예외의 보편화

자바 엔터프라이즈 서버 환경과 독립 애플리케이션과의 상황은 많이 다르다고 볼 수 있다. 독립된 환경의 애플리케이션은 해당작업을 멈추고 즉각적으로 사용자와 커뮤니케이션을 취하여 예외를 해결하면 그만이지만, 서버 환경에서는 수많은 사용자가 동시에 들어오는 데에 다가 예외가 발생하는 지점이 서버 계층에 들어가있기 때문에 예외처리를 하기 위해서 작업을 종료 시, 다른 모든 사용자들의 서비스도 종료될 것이기 때문이다. 이를 해결하기 위해서는 애플리케이션 차원에서 예외상황을 미리 파악하고, 예외가 발생하지 않도록 차단하는 것이 좋다. 또는 프로그램의 오류나 외부 환경으로 인해 예외가 발생하는 경우라면 빨리 해당 요청의 작업을 취소하고 서버 관리자나 개발자에게 통보해주는 편이 낫다.

* SQLException은 대부분 복구가 불가능한 예외이다.(언체크 런타임 처리)

JDBC의 한계

: 내구 구현은 db마다 다르겠지만 Connection, Statement, ResultSet 등의 표준 인터페이스를 통해 그 기능을 제공해주기 때문에 매우 편하다고 말할 수 있는 JDBC는 이러한 장점 외에도 단점도 존재한다.

단점으로는 DB종류에 상관없이 사용할 수 있는 데이터 액세스 코드를 작성하는 일은 쉽지 않다는 것이다, 이 밖에도 여러가지 단점이 있다..

* 비표준 SQL : 표준 문법이 있기는 하지만, 각 DB마다 대부분이 각자의 비표준 SQL을 사용하기에 DB종류 공통 문법으로 구현하기에는 어려움이 있다.
* 호환성 없는 SQLException의 DB 에러정보

: DB마다 SQL만 다른 것이 아닌, 에러의 발생원인 및 에러코드도 다 다르다.

해결책

* DB에러 코드 매핑을 통한 전환

: 가변적인 SQLException을 사용할 것이 아닌, 각 DB에서 제공해주는 DB 전용 에러 코드를 활용하는 것

Ex) 스프링이 DB별 에러 코드를 분류해서 스프링이 정의한 예외 클래스와 매핑해놓은 에러 코드 매핑정보 테이블을 만들어두고 이를 이용하는 방식을 사용한다.

<bean id="Oracle" class="org.springframework.jdbc.support.SQLErrorCodes">

<property name = "badSqlGrammerCodes">

<value>900, 903, 904, 917, 936, 942, 17006</value>

</property>

<property name = "invalidResultSetAccessCodes">

<value>17003</value>

</property>

<property name = "duplicateKeycodes">

<value>1</value>

</property>

<property name = "dataIntegerityViolationCodes">

<value>1400, 1722, 2291, 2292</value>

</property>

<property name = "dataAccessResourceFailureCodes">

<value>17002, 17447</value>

</property>

</bean>

Ex) 예외 전환 방법(중복 키 예외의 전환)

try {

jdbcTemplate.update("insert into users(name, password) values (?, ?)", user.getName(), user.getPassword());

} catch(DuplicateKeyException e) {

throw new DuplicateUserIdException(e);

// 예외를 전환할 때에는 원인이 되는 예외를 중첩하는 것이 좋다.

}