5. 자바 예외 이해

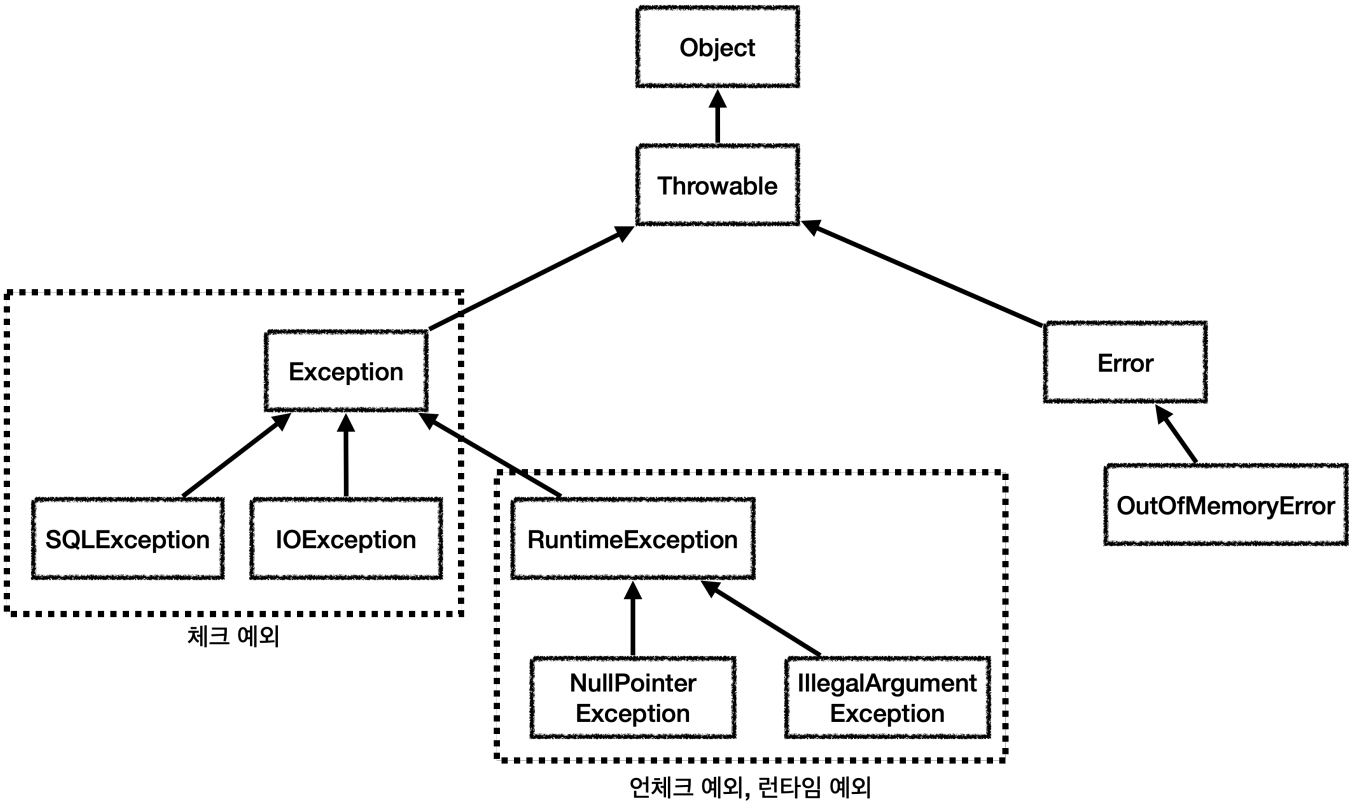
#1.인강/7.스프링 DB 1/강의#

예외 계층

스프링이 제공하는 예외 추상화를 이해하기 위해서는 먼저 자바 기본 예외에 대한 이해가 필요하다.
예외는 자바 언어의 기본 문법에 들어가기 때문에 대부분 아는 내용일 것이다. 예외의 기본 내용을 간단히 복습하고, 실무에 필요한 체크 예외와 언체크 예외의 차이와 활용 방안에 대해서도 알아보자.

예외 계층

예외 계층 그림



- `Object` : 예외도 객체이다. 모든 객체의 최상위 부모는 `Object` 이므로 예외의 최상위 부모도 `Object` 이다.
- `Throwable` : 최상위 예외이다. 하위에 `Exception` 과 `Error` 가 있다.
- `Error` : 메모리 부족이나 심각한 시스템 오류와 같이 애플리케이션에서 복구 불가능한 시스템 예외이다. 애플리케이션 개발자는 이 예외를 잡으려고 해서는 안된다.
  - 상위 예외를 `catch` 로 잡으면 그 하위 예외까지 함께 잡는다. 따라서 애플리케이션 로직에서는 `Throwable` 예외도 잡으면 안되는데, 앞서 이야기한 `Error` 예외도 함께 잡을 수 있기 때문이다. 애플리케이션 로직은 이런 이유로 `Exception` 부터 필요한 예외로 생각하고 잡으면 된다.
  - 참고로 `Error` 도 언체크 예외이다.
- `Exception` : 체크 예외
  - 애플리케이션 로직에서 사용할 수 있는 실질적인 최상위 예외이다.
  - `Exception` 과 그 하위 예외는 모두 컴파일러가 체크하는 체크 예외이다. 단 `RuntimeException` 은 예외로 한다.
- `RuntimeException` : 언체크 예외, 런타임 예외
  - 컴파일러가 체크 하지 않는 언체크 예외이다.
  - `RuntimeException` 과 그 자식 예외는 모두 언체크 예외이다.
  - `RuntimeException` 의 이름을 따라서 `RuntimeException` 과 그 하위 언체크 예외를 **런타임 예외**라고 많이 부른다. 여기서도 앞으로는 런타임 예외로 종종 부르겠다.

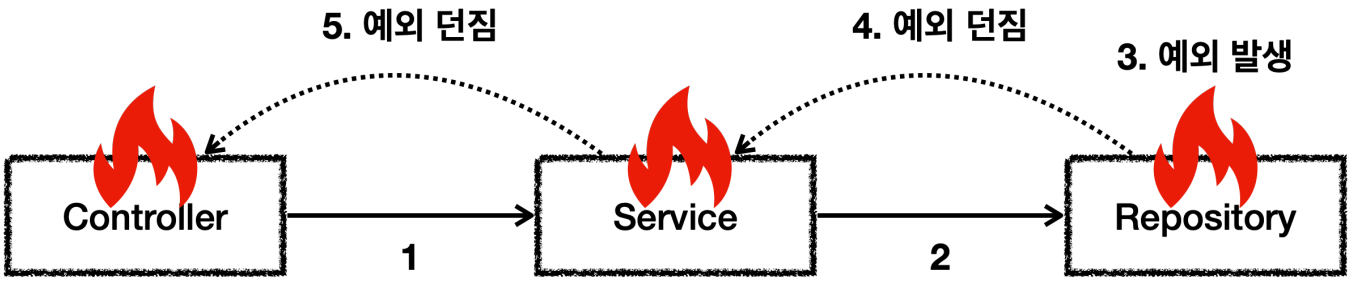
예외 기본 규칙

예외는 폭탄 돌리기와 같다. 잡아서 처리하거나, 처리할 수 없으면 밖으로 던져야한다.

예외 처리

- 5번에서 예외를 처리하면 이후에는 애플리케이션 로직이 정상 흐름으로 동작한다.

예외 던짐



예외를 처리하지 못하면 호출한 곳으로 예외를 계속 던지게 된다.

예외에 대해서는 2가지 기본 규칙을 기억하자.

- 1. 예외는 잡아서 처리하거나 던져야 한다.
- 2. 예외를 잡거나 던질 때 지정한 예외뿐만 아니라 그 예외의 자식들도 함께 처리된다.
  - 예를 들어서 `Exception`을 `catch`로 잡으면 그 하위 예외들도 모두 잡을 수 있다.
  - 예를 들어서 `Exception`을 `throws`로 던지면 그 하위 예외들도 모두 던질 수 있다.

참고: 예외를 처리하지 못하고 계속 던지면 어떻게 될까?

- 자바 `main()` 쓰레드의 경우 예외 로그를 출력하면서 시스템이 종료된다.
- 웹 애플리케이션의 경우 여러 사용자의 요청을 처리하기 때문에 하나의 예외 때문에 시스템이 종료되면 안된다. WAS가 해당 예외를 받아서 처리하는데, 주로 사용자에게 개발자가 지정한, 오류 페이지를 보여준다.

다음으로 체크 예외와 언체크 예외의 차이에 대해서 코드로 알아보자.

체크 예외 기본 이해

- `Exception`과 그 하위 예외는 모두 컴파일러가 체크하는 체크 예외이다. 단 `RuntimeException`은 예외로 한다.
- 체크 예외는 잡아서 처리하거나, 또는 밖으로 던지도록 선언해야한다. 그렇지 않으면 컴파일 오류가 발생한다.

체크 예외 전체 코드

```
package hello.jdbc.exception.basic;

import lombok.extern.slf4j.Slf4j;
import org.junit.jupiter.api.Test;

import static org.assertj.core.api.Assertions.*;

@Slf4j
public class CheckedTest {

    @Test
    void checked_catch() {
        Service service = new Service();
        service.callCatch();
    }

    @Test
    void checked_throw() {
        Service service = new Service();
        assertThatThrownBy(() -> service.callThrow())
                .isInstanceOf(MyCheckedException.class);
    }

    /**
     * Exception을 상속받은 예외는 체크 예외가 된다.
     */
    static class MyCheckedException extends Exception {
        public MyCheckedException(String message) {
            super(message);
        }
    }

    /**
```

```
     * Checked 예외는
     * 예외를 잡아서 처리하거나, 던지거나 둘중 하나를 필수로 선택해야 한다.
     */
    static class Service {
        Repository repository = new Repository();

        /**
         * 예외를 잡아서 처리하는 코드
         */
        public void callCatch() {
            try {
                repository.call();
            } catch (MyCheckedException e) {
                //예외 처리 로직
                log.info("예외 처리, message={}", e.getMessage(), e);
            }
        }

        /**
         * 체크 예외를 밖으로 던지는 코드
         * 체크 예외는 예외를 잡지 않고 밖으로 던지려면 throws 예외를 메서드에 필수로 선언해야한다.
         */
        public void callThrow() throws MyCheckedException {
            repository.call();
        }
    }

    static class Repository {
        public void call() throws MyCheckedException {
            throw new MyCheckedException("ex");
        }
    }
}
```

Exception을 상속받은 예외는 체크 예외가 된다.

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

- `MyCheckedException`는 `Exception`을 상속받았다. `Exception`을 상속받으면 체크 예외가 된다.
- 참고로 `RuntimeException`을 상속받으면 언체크 예외가 된다. 이런 규칙은 자바 언어에서 문법으로 정한 것

이다.

- 예외가 제공하는 여러가지 기본 기능이 있는데, 그 중에 오류 메시지를 보관하는 기능도 있다. 예제에서 보는 것처럼 생성자를 통해서 해당 기능을 그대로 사용하면 편리하다.

먼저 예외를 잡아서 처리하는 코드를 실행해보자.

```
@Test
void checked_catch() {
    Service service = new Service();
    service.callCatch();
}
```

- `service.callCatch()`에서 예외를 처리했기 때문에 테스트 메서드까지 예외가 올라오지 않는다.

실행 순서를 분석해보자.

1. `test` ➡ `service.callCatch()` ➡ `repository.call()` **[예외 발생, 던짐]**
2. `test` ⬅ `service.callCatch()` **[예외 처리]** ⬅ `repository.call()`
3. `test` **[정상 흐름]** ⬅ `service.callCatch()` ⬅ `repository.call()`

`Repository.call()`에서 `MyUncheckedException`예외가 발생하고, 그 예외를 `Service.callCatch()`에서 잡는 것을 확인할 수 있다.

```
log.info("예외 처리, message={}", e.getMessage(), e);
```

실행 결과

```
[Test worker] INFO hello.jdbc.exception.basic.CheckedTest - 예외 처리, message=ex
hello.jdbc.exception.basic.CheckedTest$MyCheckedException: ex
    at
hello.jdbc.exception.basic.CheckedTest$Repository.call(CheckedTest.java:64)
    at
hello.jdbc.exception.basic.CheckedTest$Service.callCatch(CheckedTest.java:45)
    at hello.jdbc.exception.basic.CheckedTest.checked_catch(CheckedTest.java:14)
```

- 실행 결과 로그를 보면 첫줄은 우리가 남긴 로그가 그대로 남는 것을 확인할 수 있다.
- 그런데 두 번째 줄 부터 예외에 대한 스택 트레이스가 추가로 출력된다.
- 이 부분은 로그를 남길 때 로그의 마지막 인수에 예외 객체를 전달해주면 로그가 해당 예외의 스택 트레이스를 추가로 출력해주는 것이다.
    - `log.info("예외 처리, message={}", e.getMessage(), e);` ⬅ 여기서 마지막에 있는 `e` 부분이다.

체크 예외를 잡아서 처리하는 코드

```
try {
    repository.call();
} catch (MyCheckedException e) {
    //예외 처리 로직
}
```

- 체크 예외를 잡아서 처리하려면 `catch(..)`를 사용해서 예외를 잡으면 된다.
- 여기서는 `MyCheckedException` 예외를 잡아서 처리한다.

catch는 해당 타입과 그 하위 타입을 모두 잡을 수 있다

```
public void callCatch() {
    try {
        repository.call();
    } catch (Exception e) {
        //예외 처리 로직
    }
}
```

- `catch`에 `MyCheckedException`의 상위 타입인 `Exception`을 적어주어도 `MyCheckedException`을 잡을 수 있다.
- `catch`에 예외를 지정하면 해당 예외와 그 하위 타입 예외를 모두 잡아준다.
- 물론 정확하게 `MyCheckedException`만 잡고 싶다면 `catch`에 `MyCheckedException`을 적어주어야 한다.

이번에는 예외를 처리하지 않고, 밖으로 던지는 코드를 살펴보자.

```
@Test
void checked_throw() {
    Service service = new Service();
    assertThatThrownBy(() -> service.callThrow())
            .isInstanceOf(MyCheckedException.class);
}
```

- `service.callThrow()`에서 예외를 처리하지 않고, 밖으로 던졌기 때문에 예외가 테스트 메서드까지 올라온다.
- 테스트에서는 기대한 것 처럼 `MyCheckedException` 예외가 던져지면 성공으로 처리한다.

실행 순서를 분석해보자.

1. `test` ➜ `service.callThrow()` ➜ `repository.call()` **[예외 발생, 던짐]**
2. `test` ← `service.callThrow()` **[예외 던짐]** ← `repository.call()`
3. `test` **[예외 도착]** ← `service.callThrow()` ← `repository.call()`

체크 예외를 밖으로 던지는 코드

```
public void callThrow() throws MyCheckedException {
    repository.call();
}
```

- 체크 예외를 처리할 수 없을 때는 `method() throws 예외`을 사용해서 밖으로 던질 예외를 필수로 지정해주어야 한다. 여기서는 `MyCheckedException`을 밖으로 던지도록 지정해주었다.

체크 예외를 밖으로 던지지 않으면 컴파일 오류 발생

```
public void callThrow() {
    repository.call();
}
```

- `throws`를 지정하지 않으면 컴파일 오류가 발생한다.
  - `Unhandled exception: hello.jdbc.exception.basic.CheckedTest.MyCheckedException`
- 체크 예외의 경우 예외를 잡아서 처리하거나 또는 `throws`를 지정해서 예외를 밖으로 던진다는 선언을 필수로 해주어야 한다.

참고로 체크 예외를 밖으로 던지는 경우에도 해당 타입과 그 하위 타입을 모두 던질 수 있다

```
public void callThrow() throws Exception {
    repository.call();
}
```

- `throws`에 `MyCheckedException`의 상위 타입인 `Exception`을 적어주어도 `MyCheckedException`을 던질 수 있다.
- `throws`에 지정한 타입과 그 하위 타입 예외를 밖으로 던진다.
- 물론 정확하게 `MyCheckedException`만 밖으로 던지고 싶다면 `throws`에 `MyCheckedException`을 적어주어야 한다.

체크 예외의 장단점

체크 예외는 예외를 잡아서 처리할 수 없을 때, 예외를 밖으로 던지는 `throws 예외`를 필수로 선언해야 한다. 그렇지 않으면 컴파일 오류가 발생한다. 이것 때문에 장점과 단점이 동시에 존재한다.

- 장점: 개발자가 실수로 예외를 누락하지 않도록 컴파일러를 통해 문제를 잡아주는 훌륭한 안전 장치이다.
- 단점: 하지만 실제로는 개발자가 모든 체크 예외를 반드시 잡거나 던지도록 처리해야 하기 때문에, 너무 번거로운 일이 된다. 크게 신경쓰고 싶지 않은 예외까지 모두 챙겨야 한다. 추가로 의존관계에 따른 단점도 있는데 이 부분

은 뒤에서 설명하겠다.

언체크 예외 기본 이해

- `RuntimeException`과 그 하위 예외는 언체크 예외로 분류된다.
- 언체크 예외는 말 그대로 컴파일러가 예외를 체크하지 않는다는 뜻이다.
- 언체크 예외는 체크 예외와 기본적으로 동일하다. 차이가 있다면 예외를 던지는 `throws`를 선언하지 않고, 생략할 수 있다. 이 경우 자동으로 예외를 던진다.

체크 예외 VS 언체크 예외

- 체크 예외: 예외를 잡아서 처리하지 않으면 항상 `throws`에 던지는 예외를 선언해야 한다.
- 언체크 예외: 예외를 잡아서 처리하지 않아도 `throws`를 생략할 수 있다.

언체크 예외 전체 코드

```
package hello.jdbc.exception.basic;

import lombok.extern.slf4j.Slf4j;
import org.junit.jupiter.api.Test;

import static org.assertj.core.api.Assertions.assertThatThrownBy;

@Slf4j
public class UncheckedTest {

    @Test
    void unchecked_catch() {
        Service service = new Service();
        service.callCatch();
    }

    @Test
    void unchecked_throw() {
        Service service = new Service();
        assertThatThrownBy(() -> service.callThrow())
                .isInstanceOf(MyUncheckedException.class);
    }
```

```java
/**
 * RuntimeException을 상속받은 예외는 언체크 예외가 된다.
 */
static class MyUncheckedException extends RuntimeException {
    public MyUncheckedException(String message) {
        super(message);
    }
}

/**
 * UnChecked 예외는
 * 예외를 잡거나, 던지지 않아도 된다.
 * 예외를 잡지 않으면 자동으로 밖으로 던진다.
 */
static class Service {

    Repository repository = new Repository();

    /**
     * 필요한 경우 예외를 잡아서 처리하면 된다.
     */
    public void callCatch() {
        try {
            repository.call();
        } catch (MyUncheckedException e) {
            //예외 처리 로직
            log.info("예외 처리, message={}", e.getMessage(), e);
        }
    }

    /**
     * 예외를 잡지 않아도 된다. 자연스럽게 상위로 넘어간다.
     * 체크 예외와 다르게 throws 예외 선언을 하지 않아도 된다.
     */
    public void callThrow() {
        repository.call();
    }
}

static class Repository {
    public void call() {
        throw new MyUncheckedException("ex");
    }
```

```
        }
    }
```

언체크 예외를 잡아서 처리하는 코드

```
try {
    repository.call();
} catch (MyUncheckedException e) {
    //예외 처리 로직
    log.info("error", e);
}
```

- 언체크 예외도 필요한 경우 이렇게 잡아서 처리할 수 있다.

언체크 예외를 밖으로 던지는 코드 - 생략

```
public void callThrow() {
    repository.call();
}
```

- 언체크 예외는 체크 예외와 다르게 `throws 예외`를 선언하지 않아도 된다.
- 말 그대로 컴파일러가 이런 부분을 체크하지 않기 때문에 언체크 예외이다.

언체크 예외를 밖으로 던지는 코드 - 선언

```
public void callThrow() throws MyUncheckedException {
    repository.call();
}
```

- 참고로 언체크 예외도 `throws 예외`를 선언해도 된다. 물론 생략할 수 있다.
- 언체크 예외는 주로 생략하지만, 중요한 예외의 경우 이렇게 선언해두면 해당 코드를 호출하는 개발자가 이런 예외가 발생한다는 점을 IDE를 통해 좀 더 편리하게 인지할 수 있다.(컴파일 시점에 막을 수 있는 것은 아니고, IDE를 통해서 인지할 수 있는 정도이다.)

언체크 예외의 장단점

언체크 예외는 예외를 잡아서 처리할 수 없을 때, 예외를 밖으로 던지는 `throws 예외`를 생략할 수 있다. 이것 때문에 장점과 단점이 동시에 존재한다.

- 장점: 신경쓰고 싶지 않은 언체크 예외를 무시할 수 있다. 체크 예외의 경우 처리할 수 없는 예외를 밖으로 던지려면 항상 `throws 예외`를 선언해야 하지만, 언체크 예외는 이 부분을 생략할 수 있다. 이후에 설명하겠지만, 신경쓰고 싶지 않은 예외의 의존관계를 참조하지 않아도 되는 장점이 있다.
- 단점: 언체크 예외는 개발자가 실수로 예외를 누락할 수 있다. 반면에 체크 예외는 컴파일러를 통해 예외 누락을 잡아준다.

정리

체크 예외와 언체크 예외의 차이는 사실 예외를 처리할 수 없을 때 예외를 밖으로 던지는 부분에 있다. 이 부분을 필수로 선언해야 하는가 생략할 수 있는가의 차이다.

체크 예외 활용

그렇다면 언제 체크 예외를 사용하고 언제 언체크(런타임) 예외를 사용하면 좋을까?

기본 원칙은 다음 2가지를 기억하자.

- 기본적으로 언체크(런타임) 예외를 사용하자.
- 체크 예외는 비즈니스 로직상 의도적으로 던지는 예외에만 사용하자.
    - 이 경우 해당 예외를 잡아서 반드시 처리해야 하는 문제일 때만 체크 예외를 사용해야 한다. 예를 들어서 다음과 같은 경우가 있다.
    - 체크 예외 예)
        - 계좌 이체 실패 예외
        - 결제시 포인트 부족 예외
        - 로그인 ID, PW 불일치 예외
    - 물론 이 경우에도 100% 체크 예외로 만들어야 하는 것은 아니다. 다만 계좌 이체 실패처럼 매우 심각한 문제는 개발자가 실수로 예외를 놓치면 안된다고 판단할 수 있다. 이 경우 체크 예외로 만들어 두면 컴파일러를 통해 놓친 예외를 인지할 수 있다.

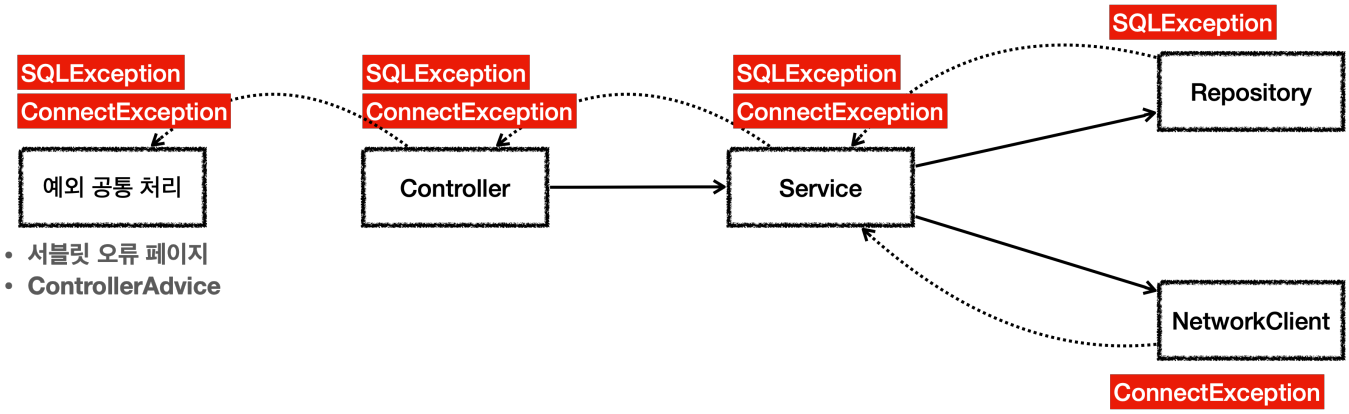
체크 예외의 문제점

체크 예외는 컴파일러가 예외 누락을 체크해주기 때문에 개발자가 실수로 예외를 놓치는 것을 막아준다. 그래서 항상 명시적으로 예외를 잡아서 처리하거나, 처리할 수 없을 때는 예외를 던지도록 `method() throws 예외`로 선언해야 한다.

지금까지 이야기를 들어보면 체크 예외가 런타임 예외보다 더 안전하고 좋아보이는데, 왜 체크 예외를 기본으로 사용하는 것이 문제가 될까?

그림과 코드로 문제점을 이해해보자.

체크 예외 문제점 - 그림



- 리포지토리는 DB에 접근해서 데이터를 저장하고 관리한다. 여기서는 `SQLException` 체크 예외를 던진다.
- `NetworkClient` 는 외부 네트워크에 접속해서 어떤 기능을 처리하는 객체이다. 여기서는 `ConnectException` 체크 예외를 던진다.
- 서비스는 리포지토리와 `NetworkClient` 를 둘다 호출한다.
  - 따라서 두 곳에서 올라오는 체크 예외인 `SQLException` 과 `ConnectException` 을 처리해야 한다.
  - 그런데 서비스는 이 둘을 처리할 방법을 모른다. `ConnectException` 처럼 연결이 실패하거나, `SQLException` 처럼 데이터베이스에서 발생하는 문제처럼 심각한 문제들은 대부분 애플리케이션 로직에서 처리할 방법이 없다.
- 서비스는 `SQLException` 과 `ConnectException` 를 처리할 수 없으므로 둘다 밖으로 던진다.
  - 체크 예외이기 때문에 던질 경우 다음과 같이 선언해야 한다.
  - `method() throws SQLException, ConnectException`
- 컨트롤러도 두 예외를 처리할 방법이 없다.
  - 다음을 선언해서 예외를 밖으로 던진다.
  - `method() throws SQLException, ConnectException`
- 웹 애플리케이션이라면 서블릿의 오류 페이지나, 또는 스프링 MVC가 제공하는 `ControllerAdvice` 에서 이런 예외를 공통으로 처리한다.
  - 이런 문제들은 보통 사용자에게 어떤 문제가 발생했는지 자세히 설명하기가 어렵다. 그래서 사용자에게는 "서비스에 문제가 있습니다." 라는 일반적인 메시지를 보여준다. ("데이터베이스에 어떤 오류가 발생했어요" 라고 알려주어도 일반 사용자가 이해할 수 없다. 그리고 보안에도 문제가 될 수 있다.)
  - API라면 보통 HTTP 상태코드 500(내부 서버 오류)을 사용해서 응답을 내려준다.
  - 이렇게 해결이 불가능한 공통 예외는 별도의 오류 로그를 남기고, 개발자가 오류를 빨리 인지할 수 있도록 메일, 알림(문자, 슬랙)등을 통해서 전달 받아야 한다. 예를 들어서 `SQLException` 이 잘못된 SQL을 작성해서 발생했다면, 개발자가 해당 SQL을 수정해서 배포하기 전까지 사용자는 같은 문제를 겪게 된다.

체크 예외 문제점 - 코드 - CheckedAppTest

```
package hello.jdbc.exception.basic;

import lombok.extern.slf4j.Slf4j;
```

```java
import org.junit.jupiter.api.Test;

import java.net.ConnectException;
import java.sql.SQLException;

import static org.assertj.core.api.Assertions.assertThatThrownBy;

@Slf4j
public class CheckedAppTest {

    @Test
    void checked() {
        Controller controller = new Controller();
        assertThatThrownBy(() -> controller.request())
                .isInstanceOf(Exception.class);
    }

    static class Controller {
        Service service = new Service();

        public void request() throws SQLException, ConnectException {
            service.logic();
        }
    }


    static class Service {
        Repository repository = new Repository();
        NetworkClient networkClient = new NetworkClient();

        public void logic() throws SQLException, ConnectException {
            repository.call();
            networkClient.call();
        }
    }

    static class NetworkClient {
        public void call() throws ConnectException {
            throw new ConnectException("연결 실패");
        }
    }

    static class Repository {
```

```
        public void call() throws SQLException {
            throw new SQLException("ex");
        }
    }

}
```

- 서비스
  - 체크 예외를 처리하지 못해서 밖으로 던지기 위해 `logic() throws SQLException, ConnectException`를 선언했다.
- 컨트롤러
  - 체크 예외를 처리하지 못해서 밖으로 던지기 위해 `request() throws SQLException, ConnectException`를 선언했다.

2가지 문제

지금까지 설명한 예시와 코드를 보면 크게 2가지 문제를 알 수 있다.

1. 복구 불가능한 예외
2. 의존 관계에 대한 문제

1. 복구 불가능한 예외

대부분의 예외는 복구가 불가능하다. 일부 복구가 가능한 예외도 있지만 아주 적다.

`SQLException`을 예를 들면 데이터베이스에 무언가 문제가 있어서 발생하는 예외이다. SQL 문법에 문제가 있을 수도 있고, 데이터베이스 자체에 뭔가 문제가 발생했을 수도 있다. 데이터베이스 서버가 중간에 다운 되었을 수도 있다. 이런 문제들은 대부분 복구가 불가능하다. 특히나 대부분의 서비스나 컨트롤러는 이런 문제를 해결할 수는 없다. 따라서 이런 문제들은 일관성 있게 공통으로 처리해야 한다. 오류 로그를 남기고 개발자가 해당 오류를 빠르게 인지하는 것이 필요하다. 서블릿 필터, 스프링 인터셉터, 스프링의 `ControllerAdvice`를 사용하면 이런 부분을 깔끔하게 공통으로 해결할 수 있다.

2. 의존 관계에 대한 문제

체크 예외의 또 다른 심각한 문제는 예외에 대한 의존 관계 문제이다.

앞서 대부분의 예외는 복구 불가능한 예외라고 했다. 그런데 체크 예외이기 때문에 컨트롤러나 서비스 입장에서는 본인이 처리할 수 없어도 어쩔 수 없이 `throws`를 통해 던지는 예외를 선언해야 한다.

체크 예외 throws 선언

```
class Controller {
    public void request() throws SQLException, ConnectException {
        service.logic();
    }
```

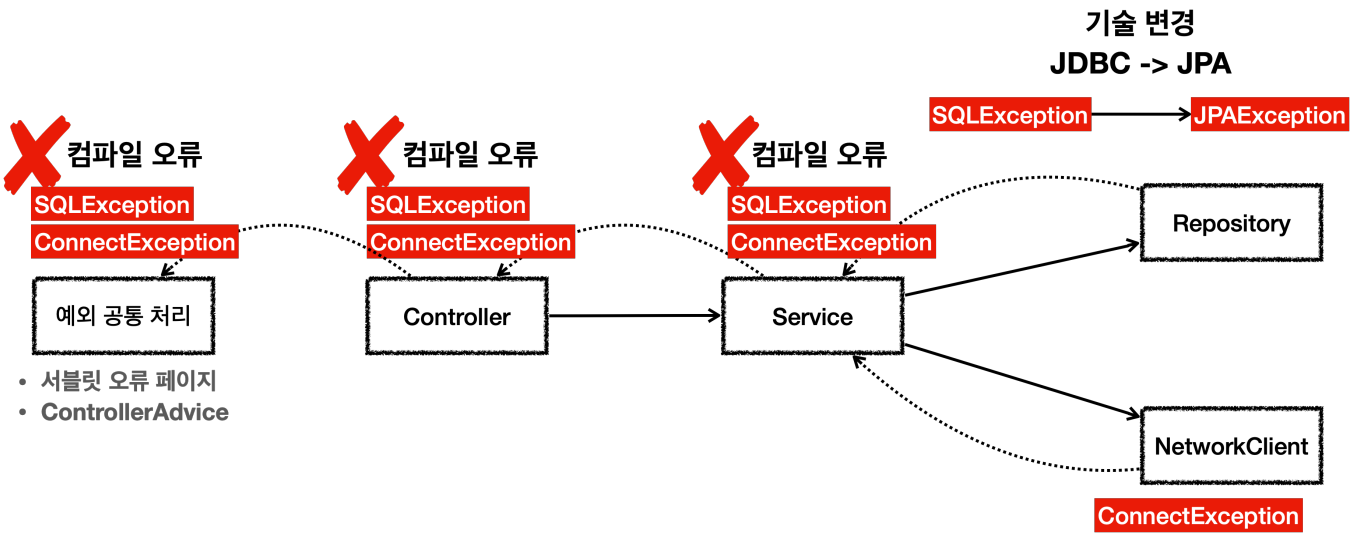
```
    }

    class Service {
        public void logic() throws SQLException, ConnectException {
            repository.call();
            networkClient.call();
        }
    }
```

`throws SQLException, ConnectException`처럼 예외를 던지는 부분을 코드에 선언하는 것이 왜 문제가 될까?
바로 서비스, 컨트롤러에서 `java.sql.SQLException`을 의존하기 때문에 문제가 된다.

향후 리포지토리를 JDBC 기술이 아닌 다른 기술로 변경한다면, 그래서 `SQLException`이 아니라 예를 들어서 `JPAException`으로 예외가 변경된다면 어떻게 될까?
`SQLException`에 의존하던 모든 서비스, 컨트롤러의 코드를 `JPAException`에 의존하도록 고쳐야 한다.

서비스나 컨트롤러 입장에서는 어차피 본인이 처리할 수 도 없는 예외를 의존해야 하는 큰 단점이 발생하게 된다.
결과적으로 OCP, DI를 통해 클라이언트 코드의 변경 없이 대상 구현체를 변경할 수 있다는 장점이 체크 예외 때문에 발목을 잡게 된다.

체크 예외 구현 기술 변경시 파급 효과



- JDBC ➜ JPA 같은 기술로 변경하면 예외도 함께 변경해야한다. 그리고 해당 예외를 던지는 모든 다음 부분도 함께 변경해야 한다.
  - `logic() throws SQLException` ➜ `logic() throws JPAException`
  - (참고로 JPA 예외는 실제 이렇지는 않고, 이해하기 쉽게 예를 든 것이다.)

정리

- 처리할 수 있는 체크 예외라면 서비스나 컨트롤러에서 처리하겠지만, 지금처럼 데이터베이스나 네트워크 통신처

럼 시스템 레벨에서 올라온 예외들은 대부분 복구가 불가능하다. 그리고 실무에서 발생하는 대부분의 예외들은 이런 시스템 예외들이다.

- 문제는 이런 경우에 체크 예외를 사용하면 아래에서 올라온 복구 불가능한 예외를 서비스, 컨트롤러 같은 각각의 클래스가 모두 알고 있어야 한다. 그래서 불필요한 의존관계 문제가 발생하게 된다.

throws Exception

`SQLException`, `ConnectException` 같은 시스템 예외는 컨트롤러나 서비스에서는 대부분 복구가 불가능하고 처리할 수 없는 체크 예외이다. 따라서 다음과 같이 처리해주어야 한다.

```
void method() throws SQLException, ConnectException {..}
```

그런데 다음과 같이 최상위 예외인 `Exception`을 던져도 문제를 해결할 수 있다.

```
void method() throws Exception {..}
```

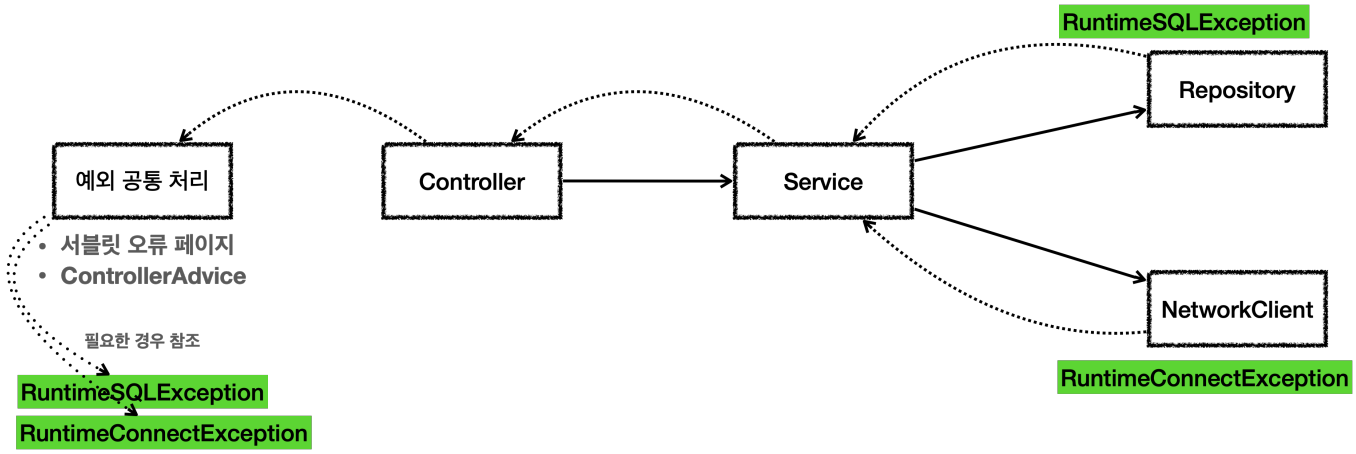
이렇게 하면 `Exception`은 물론이고 그 하위 타입인 `SQLException`, `ConnectException`도 함께 던지게 된다. 코드가 깔끔해지는 것 같지만, `Exception`은 최상위 타입이므로 모든 체크 예외를 다 밖으로 던지는 문제가 발생한다.

결과적으로 체크 예외의 최상위 타입인 `Exception`을 던지게 되면 다른 체크 예외를 체크할 수 있는 기능이 무효화되고, 중요한 체크 예외를 다 놓치게 된다. 중간에 중요한 체크 예외가 발생해도 컴파일러는 `Exception`을 던지기 때문에 문법에 맞다고 판단해서 컴파일 오류가 발생하지 않는다.
이렇게 하면 모든 예외를 다 던지기 때문에 체크 예외를 의도한 대로 사용하는 것이 아니다. 따라서 꼭 필요한 경우가 아니면 이렇게 `Exception` 자체를 밖으로 던지는 것은 좋지 않은 방법이다.

언체크 예외 활용

이번에는 런타임 예외를 사용해보자.

런타임 예외 사용 - 그림



- `SQLException`을 런타임 예외인 `RuntimeSQLException`으로 변환했다.
- `ConnectException` 대신에 `RuntimeConnectException`을 사용하도록 바꾸었다.
- 런타임 예외이기 때문에 서비스, 컨트롤러는 해당 예외들을 처리할 수 없다면 별도의 선언 없이 그냥 두면 된다.

런타임 예외 사용 변환 - 코드 - UncheckedAppTest

```
package hello.jdbc.exception.basic;

import lombok.extern.slf4j.Slf4j;
import org.junit.jupiter.api.Test;

import java.sql.SQLException;

import static org.assertj.core.api.Assertions.assertThatThrownBy;

@Slf4j
public class UncheckedAppTest {

    @Test
    void unchecked() {
        Controller controller = new Controller();
        assertThatThrownBy(() -> controller.request())
                .isInstanceOf(Exception.class);
    }

    @Test
    void printEx() {
        Controller controller = new Controller();
        try {
            controller.request();
        } catch (Exception e) {
            //e.printStackTrace();
            log.info("ex", e);
```

```
    }
}

static class Controller {
    Service service = new Service();

    public void request() {
        service.logic();
    }
}


static class Service {
    Repository repository = new Repository();
    NetworkClient networkClient = new NetworkClient();

    public void logic() {
        repository.call();
        networkClient.call();
    }
}

static class NetworkClient {
    public void call() {
        throw new RuntimeConnectException("연결 실패");
    }
}

static class Repository {
    public void call() {
        try {
            runSQL();
        } catch (SQLException e) {
            throw new RuntimeSQLException(e);
        }
    }

    private void runSQL() throws SQLException {
        throw new SQLException("ex");
    }
}

static class RuntimeConnectException extends RuntimeException {
```

```
        public RuntimeConnectException(String message) {
            super(message);
        }
    }

    static class RuntimeSQLException extends RuntimeException {
        public RuntimeSQLException() {
        }

        public RuntimeSQLException(Throwable cause) {
            super(cause);
        }
    }

}
```

예외 전환

- 리포지토리에서 체크 예외인 `SQLException`이 발생하면 런타임 예외인 `RuntimeSQLException`으로 전환해서 예외를 던진다.
  - 참고로 이때 기존 예외를 포함해주어야 예외 출력시 스택 트레이스에서 기존 예외도 함께 확인할 수 있다. 예외 포함에 대한 부분은 조금 뒤에 더 자세히 설명한다.
- `NetworkClient`는 단순히 기존 체크 예외를 `RuntimeConnectException`이라는 런타임 예외가 발생하도록 코드를 바꾸었다.

런타임 예외 - 대부분 복구 불가능한 예외

시스템에서 발생한 예외는 대부분 복구 불가능 예외이다. 런타임 예외를 사용하면 서비스나 컨트롤러가 이런 복구 불가능한 예외를 신경쓰지 않아도 된다. 물론 이렇게 복구 불가능한 예외는 일관성 있게 공통으로 처리해야 한다.

런타임 예외 - 의존 관계에 대한 문제

런타임 예외는 해당 객체가 처리할 수 없는 예외는 무시하면 된다. 따라서 체크 예외 처럼 예외를 강제로 의존하지 않아도 된다.

런타임 예외 throws 생략

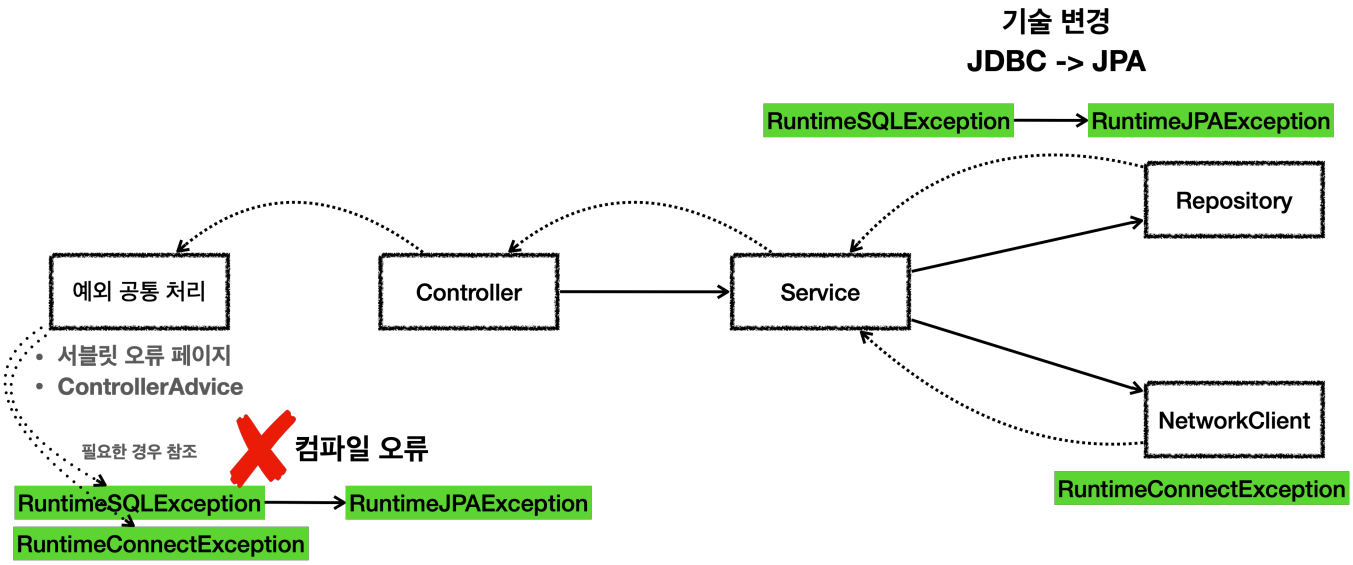
```
class Controller {
    public void request() {
        service.logic();
    }
}

class Service {
    public void logic() {
```

```
            repository.call();
            networkClient.call();
        }
    }
```

런타임 예외이기 때문에 컨트롤러나 서비스가 예외를 처리할 수 없다면 다음 부분을 생략할 수 있다.

- `method() throws RuntimeSQLException, RuntimeConnectException`
- 따라서 컨트롤러와 서비스에서 해당 예외에 대한 의존 관계가 발생하지 않는다.

런타임 예외 구현 기술 변경시 파급 효과



- 런타임 예외를 사용하면 중간에 기술이 변경되어도 해당 예외를 사용하지 않는 컨트롤러, 서비스에서는 코드를 변경하지 않아도 된다.
- 구현 기술이 변경되는 경우, 예외를 공통으로 처리하는 곳에서는 예외에 따른 다른 처리가 필요할 수 있다. 하지만 공통 처리하는 한곳만 변경하면 되기 때문에 변경의 영향 범위는 최소화 된다.

정리

처음 자바를 설계할 당시에는 체크 예외가 더 나은 선택이라 생각했다. 그래서 자바가 기본으로 제공하는 기능들에는 체크 예외가 많다. 그런데 시간이 흐르면서 복구 할 수 없는 예외가 너무 많아졌다. 특히 라이브러리를 점점 더 많이 사용하면서 처리해야 하는 예외도 더 늘어났다. 체크 예외는 해당 라이브러리들이 제공하는 모든 예외를 처리할 수 없을 때마다 `throws`에 예외를 덕지덕지 붙어야 했다. 그래서 개발자들은 `throws Exception`이라는 극단적?인 방법도 자주 사용하게 되었다. 물론 이 방법은 사용하면 안된다. 모든 예외를 던진다고 선언하는 것인데, 결과적으로 어떤 예외를 잡고 어떤 예외를 던지는지 알 수 없기 때문이다. 체크 예외를 사용한다면 잡을 건 잡고 던질 예외는 명확하게 던지도록 선언해야 한다.

체크 예외의 이런 문제점 때문에 최근 라이브러리들은 대부분 런타임 예외를 기본으로 제공한다. 사실 위에서 예시로 설명한 JPA 기술도 런타임 예외를 사용한다. 스프링도 대부분 런타임 예외를 제공한다.

런타임 예외도 필요하면 잡을 수 있기 때문에 필요한 경우에는 잡아서 처리하고, 그렇지 않으면 자연스럽게 던지도록 둔다. 그리고 예외를 공통으로 처리하는 부분을 앞에 만들어서 처리하면 된다.

추가로 런타임 예외는 놓칠 수 있기 때문에 문서화가 중요하다.

런타임 예외는 문서화

- 런타임 예외는 문서화를 잘해야 한다.
- 또는 코드에 `throws 런타임예외`을 남겨서 중요한 예외를 인지할 수 있게 해준다.

JPA EntityManager

```
/**
 * Make an instance managed and persistent.
 * @param entity  entity instance
 * @throws EntityExistsException if the entity already exists.
 * @throws IllegalArgumentException if the instance is not an
 *         entity
 * @throws TransactionRequiredException if there is no transaction when
 *         invoked on a container-managed entity manager of that is of type
 *         <code>PersistenceContextType.TRANSACTION</code>
 */
public void persist(Object entity);
```

- 예) 문서에 예외 명시

스프링 JdbcTemplate

```
/**
 * Issue a single SQL execute, typically a DDL statement.
 * @param sql static SQL to execute
 * @throws DataAccessException if there is any problem
 */
void execute(String sql) throws DataAccessException;
```

- 예) `method() throws DataAccessException`와 같이 문서화 + 코드에도 명시
  - 런타임 예외도 `throws`에 선언할 수 있다. 물론 생략해도 된다.
  - 던지는 예외가 명확하고 중요하다면, 코드에 어떤 예외를 던지는지 명시되어 있기 때문에 개발자가 IDE를 통해서 예외를 확인하기가 편리하다.
  - 물론 컨트롤러나 서비스에서 `DataAccessException`을 사용하지 않는다면 런타임 예외이기 때문에 무시해도 된다.

예외 포함과 스택 트레이스

예외를 전환할 때는 꼭! 기존 예외를 포함해야 한다. 그렇지 않으면 스택 트레이스를 확인할 때 심각한 문제가 발생한다.

```
@Test
void printEx() {
    Controller controller = new Controller();
    try {
        controller.request();
    } catch (Exception e) {
        //e.printStackTrace();
        log.info("ex", e);
    }
}
```

- 로그를 출력할 때 마지막 파라미터에 예외를 넣어주면 로그에 스택 트레이스를 출력할 수 있다.
  - 예) `log.info("message={}", "message", ex)`, 여기에서 마지막에 `ex`를 전달하는 것을 확인할 수 있다. 이렇게 하면 스택 트레이스에 로그를 출력할 수 있다.
  - 예) `log.info("ex", ex)` 지금 예에서는 파라미터가 없기 때문에, 예외만 파라미터에 전달하면 스택 트레이스를 로그에 출력할 수 있다.
- `System.out`에 스택 트레이스를 출력하려면 `e.printStackTrace()`를 사용하면 된다.
  - 실무에서는 항상 로그를 사용해야 한다는 점을 기억하자.

기존 예외를 포함하는 경우

```
public void call() {
    try {
        runSQL();
    } catch (SQLException e) {
        throw new RuntimeSQLException(e); //기존 예외(e) 포함
    }
}
```

```
13:10:45.626 [Test worker] INFO hello.jdbc.exception.basic.UncheckedAppTest - ex
hello.jdbc.exception.basic.UncheckedAppTest$RuntimeSQLException:
java.sql.SQLException: ex
    at
hello.jdbc.exception.basic.UncheckedAppTest$Repository.call(UncheckedAppTest.jav
a:61)
    at
hello.jdbc.exception.basic.UncheckedAppTest$Service.logic(UncheckedAppTest.java:
45)
    at
hello.jdbc.exception.basic.UncheckedAppTest$Controller.request(UncheckedAppTest.
```

```
java:35)
    at
hello.jdbc.exception.basic.UncheckedAppTest.printEx(UncheckedAppTest.java:24)
Caused by: java.sql.SQLException: ex
    at
hello.jdbc.exception.basic.UncheckedAppTest$Repository.runSQL(UncheckedAppTest.j
ava:66)
    at
hello.jdbc.exception.basic.UncheckedAppTest$Repository.call(UncheckedAppTest.jav
a:59)
```

예외를 포함해서 기존에 발생한 `java.sql.SQLException`과 스택 트레이스를 확인할 수 있다.

기존 예외를 포함하지 않는 경우

```
public void call() {
    try {
        runSQL();
    } catch (SQLException e) {
        throw new RuntimeSQLException(); //기존 예외(e) 제외
    }
}
```

```
[Test worker] INFO hello.jdbc.exception.basic.UncheckedAppTest - ex
hello.jdbc.exception.basic.UncheckedAppTest$RuntimeSQLException: null
    at
hello.jdbc.exception.basic.UncheckedAppTest$Repository.call(UncheckedAppTest.jav
a:61)
    at
hello.jdbc.exception.basic.UncheckedAppTest$Service.logic(UncheckedAppTest.java:
45)
```

예외를 포함하지 않아서 기존에 발생한 `java.sql.SQLException`과 스택 트레이스를 확인할 수 없다. 변환한 `RuntimeSQLException`부터 예외를 확인할 수 있다. 만약 실제 DB에 연동했다면 DB에서 발생한 예외를 확인할 수 없는 심각한 문제가 발생한다.

예외를 전환할 때는 꼭! 기존 예외를 포함하자

정리