6. 스프링과 문제 해결 - 예외 처리, 반복

#2.인강/7.스프링 DB 1/강의#

체크 예외와 인터페이스

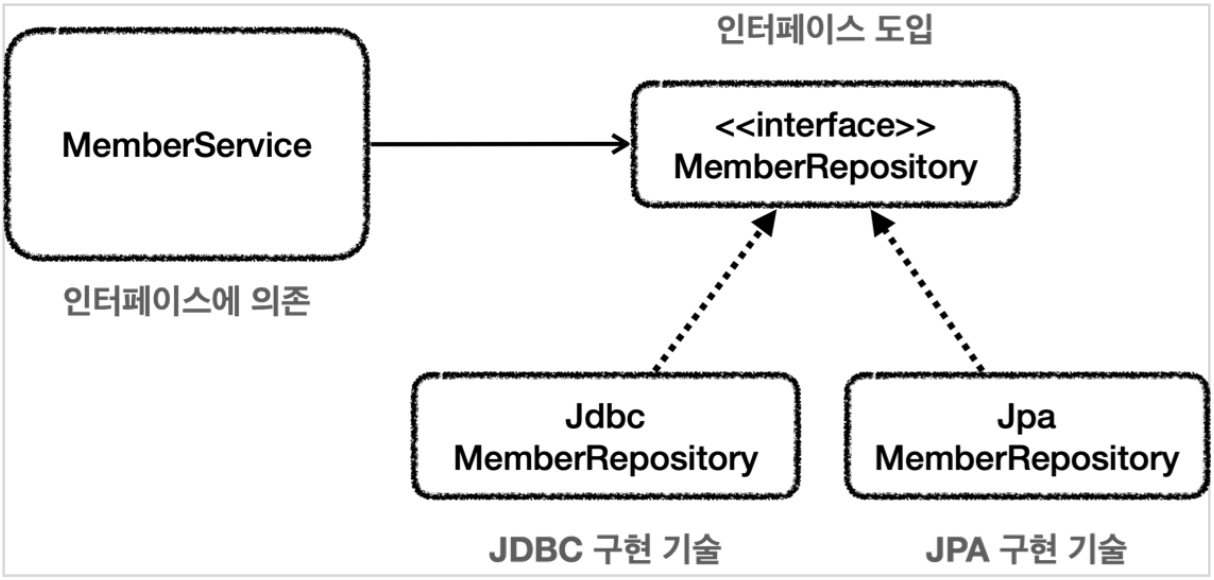
서비스 계층은 가급적 특정 구현 기술에 의존하지 않고, 순수하게 유지하는 것이 좋다. 이렇게 하려면 예외에 대한 의존도 함께 해결해야한다.
예를 들어서 서비스가 처리할 수 없는 `SQLException`에 대한 의존을 제거하려면 어떻게 해야할까?
서비스가 처리할 수 없으므로 리포지토리가 던지는 `SQLException` 체크 예외를 런타임 예외로 전환해서 서비스 계층에 던지자. 이렇게 하면 서비스 계층이 해당 예외를 무시할 수 있기 때문에, 특정 구현 기술에 의존하는 부분을 제거하고 서비스 계층을 순수하게 유지할 수 있다.

지금부터 코드로 이 방법을 적용해보자.

인터페이스 도입

먼저 `MemberRepository` 인터페이스도 도입해서 구현 기술을 쉽게 변경할 수 있게 해보자.

인터페이스 도입 그림



- 이렇게 인터페이스를 도입하면 `MemberService`는 `MemberRepository` 인터페이스에만 의존하면 된다.
- 이제 구현 기술을 변경하고 싶으면 DI를 사용해서 `MemberService` 코드의 변경 없이 구현 기술을 변경할 수 있다.

MemberRepository 인터페이스

```
package hello.jdbc.repository;
import hello.jdbc.domain.Member;

public interface MemberRepository {
    Member save(Member member);
    Member findById(String memberId);
    void update(String memberId, int money);
    void delete(String memberId);
}
```

특정 기술에 종속되지 않는 순수한 인터페이스이다. 이 인터페이스를 기반으로 특정 기술을 사용하는 구현체를 만들면 된다.

체크 예외와 인터페이스

잠깐? 기존에는 왜 이런 인터페이스를 만들지 않았을까? 사실 다음과 같은 문제가 있기 때문에 만들지 않았다.

왜냐하면 `SQLException`이 체크 예외이기 때문이다. 여기서 체크 예외가 또 발목을 잡는다.

체크 예외를 사용하려면 인터페이스에도 해당 체크 예외가 선언 되어 있어야 한다.

예를 들면 다음과 같은 코드가 된다.

체크 예외 코드에 인터페이스 도입시 문제점 - 인터페이스

```java
package hello.jdbc.repository;
import hello.jdbc.domain.Member;
import java.sql.SQLException;

public interface MemberRepositoryEx {
    Member save(Member member) throws SQLException;
    Member findById(String memberId) throws SQLException;
    void update(String memberId, int money) throws SQLException;
    void delete(String memberId) throws SQLException;
}
```

- 인터페이스의 메서드에 `throws SQLException`이 있는 것을 확인할 수 있다.

체크 예외 코드에 인터페이스 도입시 문제점 - 구현 클래스

```java
@Slf4j
public class MemberRepositoryV3 implements MemberRepositoryEx {

    public Member save(Member member) throws SQLException {
        String sql = "insert into member(member_id, money) values(?, ?)";
    }
}
```

- 인터페이스의 구현체가 체크 예외를 던지려면, 인터페이스 메서드에 먼저 체크 예외를 던지는 부분이 선언되어 있어야 한다. 그래야 구현 클래스의 메서드도 체크 예외를 던질 수 있다.
  - 쉽게 이야기 해서 `MemberRepositoryV3`가 `throws SQLException`를 하려면 `MemberRepositoryEx` 인터페이스에도 `throws SQLException`이 필요하다.
- 참고로 구현 클래스의 메서드에 선언할 수 있는 예외는 부모 타입에서 던진 예외와 같거나 하위 타입이어야 한다.
  - 예를 들어서 인터페이스 메서드에 `throws Exception`를 선언하면, 구현 클래스 메서드에 `throws SQLException`는 가능하다. `SQLException`은 `Exception`의 하위 타입이기 때문이다.

특정 기술에 종속되는 인터페이스

구현 기술을 쉽게 변경하기 위해서 인터페이스를 도입하더라도 `SQLException`과 같은 특정 구현 기술에 종속적인 체크 예외를 사용하게 되면 인터페이스에도 해당 예외를 포함해야 한다. 하지만 이것은 우리가 원하던 순수한 인터페이스가 아니다. JDBC 기술에 종속적인 인터페이스일 뿐이다. 인터페이스를 만드는 목적은 구현체를 쉽게 변경하기 위함인데, 이미 인터페이스가 특정 구현 기술에 오염이 되어 버렸다. 향후 JDBC가 아닌 다른 기술로 변경한다면 인터페이스 자체를 변경해야 한다.

런타임 예외와 인터페이스

런타임 예외는 이런 부분에서 자유롭다. 인터페이스에 런타임 예외를 따로 선언하지 않아도 된다. 따라서 인터페이스가 특정 기술에 종속적일 필요가 없다.

런타임 예외 적용

실제 코드에 런타임 예외를 사용하도록 적용해보자.

MemberRepository 인터페이스

```
package hello.jdbc.repository;
import hello.jdbc.domain.Member;

public interface MemberRepository {
    Member save(Member member);
    Member findById(String memberId);
    void update(String memberId, int money);
    void delete(String memberId);
}
```

MyDbException 런타임 예외

```
package hello.jdbc.repository.ex;

public class MyDbException extends RuntimeException {

    public MyDbException() {
    }

    public MyDbException(String message) {
        super(message);
    }

    public MyDbException(String message, Throwable cause) {
        super(message, cause);
```

```
    }

    public MyDbException(Throwable cause) {
        super(cause);
    }
}
```

- `RuntimeException`을 상속받았다. 따라서 `MyDbException`은 런타임(언체크) 예외가 된다.

MemberRepositoryV4_1

```
package hello.jdbc.repository;

import hello.jdbc.domain.Member;
import hello.jdbc.repository.ex.MyDbException;
import lombok.extern.slf4j.Slf4j;
import org.springframework.jdbc.datasource.DataSourceUtils;
import org.springframework.jdbc.support.JdbcUtils;

import javax.sql.DataSource;
import java.sql.*;
import java.util.NoSuchElementException;

/**
 * 예외 누수 문제 해결
 * 체크 예외를 런타임 예외로 변경
 * MemberRepository 인터페이스 사용
 * throws SQLException 제거
 */
@Slf4j
public class MemberRepositoryV4_1 implements MemberRepository {

    private final DataSource dataSource;

    public MemberRepositoryV4_1(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    @Override
```

```
    public Member save(Member member) {
        String sql = "insert into member(member_id, money) values(?, ?)";

        Connection con = null;
        PreparedStatement pstmt = null;

        try {
            con = getConnection();
            pstmt = con.prepareStatement(sql);
            pstmt.setString(1, member.getMemberId());
            pstmt.setInt(2, member.getMoney());
            pstmt.executeUpdate();
            return member;
        } catch (SQLException e) {
            throw new MyDbException(e);
        } finally {
            close(con, pstmt, null);
        }
    }

    @Override
    public Member findById(String memberId) {
        String sql = "select * from member where member_id = ?";

        Connection con = null;
        PreparedStatement pstmt = null;
        ResultSet rs = null;

        try {
            con = getConnection();
            pstmt = con.prepareStatement(sql);
            pstmt.setString(1, memberId);

            rs = pstmt.executeQuery();

            if (rs.next()) {
                Member member = new Member();
                member.setMemberId(rs.getString("member_id"));
                member.setMoney(rs.getInt("money"));
```

```
                return member;
            } else {
                throw new NoSuchElementException("member not found memberId=" +
memberId);
            }

        } catch (SQLException e) {
            throw new MyDbException(e);
        } finally {
            close(con, pstmt, rs);
        }

    }

    @Override
    public void update(String memberId, int money) {

        String sql = "update member set money=? where member_id=?";

        Connection con = null;
        PreparedStatement pstmt = null;

        try {
            con = getConnection();
            pstmt = con.prepareStatement(sql);
            pstmt.setInt(1, money);
            pstmt.setString(2, memberId);
            pstmt.executeUpdate();

        } catch (SQLException e) {
            throw new MyDbException(e);
        } finally {
            close(con, pstmt, null);
        }
    }

    @Override
    public void delete(String memberId) {
```

```
        String sql = "delete from member where member_id=?";

        Connection con = null;
        PreparedStatement pstmt = null;

        try {
            con = getConnection();
            pstmt = con.prepareStatement(sql);
            pstmt.setString(1, memberId);
            pstmt.executeUpdate();

        } catch (SQLException e) {
            throw new MyDbException(e);
        } finally {
            close(con, pstmt, null);
        }
    }

    private void close(Connection con, Statement stmt, ResultSet rs) {
        JdbcUtils.closeResultSet(rs);
        JdbcUtils.closeStatement(stmt);
        DataSourceUtils.releaseConnection(con, dataSource);
    }

    private Connection getConnection() {
        Connection con = DataSourceUtils.getConnection(dataSource);
        log.info("get connection={} class={}", con, con.getClass());
        return con;
    }

}
```

- `MemberRepository` 인터페이스를 구현한다.
- 이 코드에서 핵심은 `SQLException` 이라는 체크 예외를 `MyDbException` 이라는 런타임 예외로 변환해서 던지는 부분이다.

예외 변환

```
catch (SQLException e) {
```

```
        throw new MyDbException(e);
    }
```

- 잘 보면 기존 예외를 생성자를 통해서 포함하고 있는 것을 확인할 수 있다. 예외는 원인이 되는 예외를 내부에 포함할 수 있는데, 꼭 이렇게 작성해야 한다. 그래야 예외를 출력했을 때 원인이 되는 기존 예외도 함께 확인할 수 있다.
- `MyDbException`이 내부에 `SQLException`을 포함하고 있다고 이해하면 된다. 예외를 출력했을 때 스택 트레이스를 통해 둘다 확인할 수 있다.

다음과 같이 기존 예외를 무시하고 작성하면 절대 안된다!

예외 변환 - 기존 예외 무시

```
catch (SQLException e) {
    throw new MyDbException();
}
```

- 잘 보면 `new MyDbException()`으로 해당 예외만 생성하고 기존에 있는 `SQLException`은 포함하지 않고 무시한다.
- 따라서 `MyDbException`은 내부에 원인이 되는 다른 예외를 포함하지 않는다.
- 이렇게 원인이 되는 예외를 내부에 포함하지 않으면, 예외를 스택 트레이스를 통해 출력했을 때 기존에 원인이 되는 부분을 확인할 수 없다.
  - 만약 `SQLException`에서 문법 오류가 발생했다면 그 부분을 확인할 방법이 없게 된다.

> **주의!**
> 예외를 변환할 때는 기존 예외를 꼭! 포함하자. 장애가 발생하고 로그에서 진짜 원인이 남지 않는 심각한 문제가 발생할 수 있다. 중요한 내용이어서 한번 더 설명했다.

이번에는 서비스가 인터페이스를 사용하도록 하자. 기존 코드를 유지하기 위해 `MemberServiceV3_3`를 복사해서 수정하자.

MemberServiceV4

```
package hello.jdbc.service;

import hello.jdbc.domain.Member;
import hello.jdbc.repository.MemberRepository;
import lombok.RequiredArgsConstructor;
```

```java
import lombok.extern.slf4j.Slf4j;
import org.springframework.transaction.annotation.Transactional;

/**
 * 예외 누수 문제 해결
 * SQLException 제거
 *
 * MemberRepository 인터페이스 의존
 */
@Slf4j
@RequiredArgsConstructor
public class MemberServiceV4 {

    private final MemberRepository memberRepository;

    @Transactional
    public void accountTransfer(String fromId, String toId, int money) {
        bizLogic(fromId, toId, money);
    }

    private void bizLogic(String fromId, String toId, int money) {
        Member fromMember = memberRepository.findById(fromId);
        Member toMember = memberRepository.findById(toId);

        memberRepository.update(fromId, fromMember.getMoney() - money);
        validation(toMember);
        memberRepository.update(toId, toMember.getMoney() + money);
    }

    private void validation(Member toMember) {
        if (toMember.getMemberId().equals("ex")) {
            throw new IllegalStateException("이체중 예외 발생");
        }
    }

}
```

- `MemberRepository` 인터페이스에 의존하도록 코드를 변경했다.
- `MemberServiceV3_3` 와 비교해서 보면 드디어 메서드에서 `throws SQLException` 부분이 제거된 것을

확인할 수 있다.

- 드디어 순수한 서비스를 완성했다.

MemberServiceV4Test

```
package hello.jdbc.service;

import hello.jdbc.domain.Member;
import hello.jdbc.repository.MemberRepository;
import hello.jdbc.repository.MemberRepositoryV4_1;
import org.assertj.core.api.Assertions;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.springframework.aop.support.AopUtils;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.context.annotation.Bean;
import org.springframework.jdbc.datasource.DataSourceTransactionManager;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.transaction.PlatformTransactionManager;

import javax.sql.DataSource;

import static hello.jdbc.connection.ConnectionConst.*;
import static org.assertj.core.api.Assertions.assertThat;
import static org.assertj.core.api.Assertions.assertThatThrownBy;

/**
 * 예외 누수 문제 해결
 * SQLException 제거
 *
 * MemberRepository 인터페이스 의존
 */
@SpringBootTest
class MemberServiceV4Test {

    public static final String MEMBER_A = "memberA";
```

```
    public static final String MEMBER_B = "memberB";
    public static final String MEMBER_EX = "ex";

    @Autowired
    MemberRepository memberRepository;
    @Autowired
    MemberServiceV4 memberService;

    @AfterEach
    void after() throws SQLException {
        memberRepository.delete(MEMBER_A);
        memberRepository.delete(MEMBER_B);
        memberRepository.delete(MEMBER_EX);
    }

    @TestConfiguration
    static class TestConfig {

        private final DataSource dataSource;

        public TestConfig(DataSource dataSource) {
            this.dataSource = dataSource;
        }

        @Bean
        MemberRepository memberRepository() {
            return new MemberRepositoryV4_1(dataSource); //단순 예외 변환
        }

        @Bean
        MemberServiceV4 memberServiceV4() {
            return new MemberServiceV4(memberRepository());
        }
    }

    @Test
    void AopCheck() {
        log.info("memberService class={}", memberService.getClass());
        log.info("memberRepository class={}", memberRepository.getClass());
```

```
        Assertions.assertThat(AopUtils.isAopProxy(memberService)).isTrue();
        Assertions.assertThat(AopUtils.isAopProxy(memberRepository)).isFalse();
    }

    @Test
    @DisplayName("정상 이체")
    void accountTransfer() {
        //given
        Member memberA = new Member(MEMBER_A, 10000);
        Member memberB = new Member(MEMBER_B, 10000);
        memberRepository.save(memberA);
        memberRepository.save(memberB);

        //when
        memberService.accountTransfer(memberA.getMemberId(),
memberB.getMemberId(), 2000);

        //then
        Member findMemberA = memberRepository.findById(memberA.getMemberId());
        Member findMemberB = memberRepository.findById(memberB.getMemberId());
        assertThat(findMemberA.getMoney()).isEqualTo(8000);
        assertThat(findMemberB.getMoney()).isEqualTo(12000);
    }

    @Test
    @DisplayName("이체중 예외 발생")
    void accountTransferEx() {
        //given
        Member memberA = new Member(MEMBER_A, 10000);
        Member memberEx = new Member(MEMBER_EX, 10000);
        memberRepository.save(memberA);
        memberRepository.save(memberEx);

        //when
        assertThatThrownBy(() ->
memberService.accountTransfer(memberA.getMemberId(), memberEx.getMemberId(),
2000))
                .isInstanceOf(IllegalStateException.class);
```

```
        //then
        Member findMemberA = memberRepository.findById(memberA.getMemberId());
        Member findMemberEx =
memberRepository.findById(memberEx.getMemberId());

        //memberA의 돈이 롤백 되어야함
        assertThat(findMemberA.getMoney()).isEqualTo(10000);
        assertThat(findMemberEx.getMoney()).isEqualTo(10000);
    }

}
```

- `MemberRepository` 인터페이스를 사용하도록 했다.
- 테스트가 모두 정상 동작하는 것을 확인할 수 있다.

정리

- 체크 예외를 런타임 예외로 변환하면서 인터페이스와 서비스 계층의 순수성을 유지할 수 있게 되었다.
- 덕분에 향후 JDBC에서 다른 구현 기술로 변경하더라도 서비스 계층의 코드를 변경하지 않고 유지할 수 있다.

남은 문제

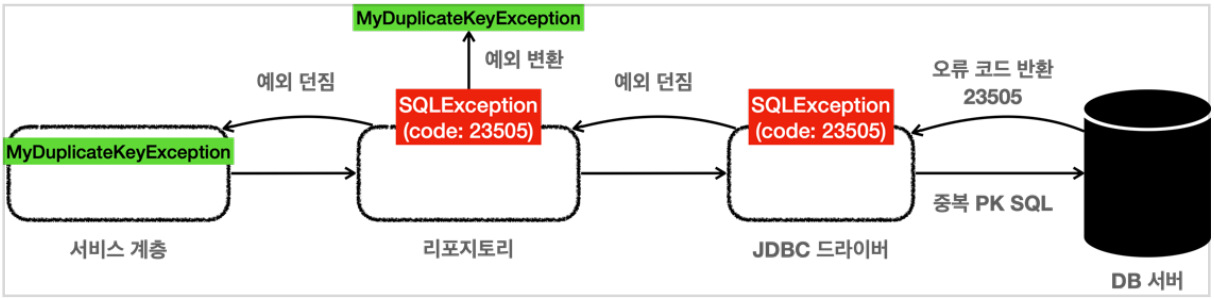
리포지토리에서 넘어오는 특정한 예외의 경우 복구를 시도할 수도 있다. 그런데 지금 방식은 항상 `MyDbException`이라는 예외만 넘어오기 때문에 예외를 구분할 수 없는 단점이 있다. 만약 특정 상황에는 예외를 잡아서 복구하고 싶으면 예외를 어떻게 구분해서 처리할 수 있을까?

데이터 접근 예외 직접 만들기

데이터베이스 오류에 따라서 특정 예외는 복구하고 싶을 수 있다.
예를 들어서 회원 가입시 DB에 이미 같은 ID가 있으면 ID 뒤에 숫자를 붙여서 새로운 ID를 만들어야 한다고 가정해보자.
ID를 `hello`라고 가입 시도 했는데, 이미 같은 아이디가 있으면 `hello12345` 와 같이 뒤에 임의의 숫자를 붙여서 가입하는 것이다.

데이터를 DB에 저장할 때 같은 ID가 이미 데이터베이스에 저장되어 있다면, 데이터베이스는 오류 코드를 반환하고, 이 오류 코드를 받은 JDBC 드라이버는 `SQLException`을 던진다. 그리고 `SQLException`에는 데이터베이스가 제공하는 `errorCode`라는 것이 들어있다.

데이터베이스 오류 코드 그림



H2 데이터베이스의 키 중복 오류 코드

```
e.getErrorCode() == 23505
```

`SQLException` 내부에 들어있는 `errorCode`를 활용하면 데이터베이스에서 어떤 문제가 발생했는지 확인할 수 있다.

H2 데이터베이스 예

- `23505` : 키 중복 오류
- `42000` : SQL 문법 오류

참고로 같은 오류여도 각각의 데이터베이스마다 정의된 오류 코드가 다르다. 따라서 오류 코드를 사용할 때는 데이터베이스 메뉴얼을 확인해야 한다.

예) 키 중복 오류 코드

- H2 DB: `23505`
- MySQL: `1062`

H2 데이터베이스 오류 코드 참고

> https://www.h2database.com/javadoc/org/h2/api/ErrorCode.html

서비스 계층에서는 예외 복구를 위해 키 중복 오류를 확인할 수 있어야 한다. 그래야 새로운 ID를 만들어서 다시 저장을 시도할 수 있기 때문이다. 이러한 과정이 바로 예외를 확인해서 복구하는 과정이다.
리포지토리는 `SQLException`을 서비스 계층에 던지고 서비스 계층은 이 예외의 오류 코드를 확인해서 키 중복 오류(`23505`)인 경우 새로운 ID를 만들어서 다시 저장하면 된다.
그런데 `SQLException`에 들어있는 오류 코드를 활용하기 위해 `SQLException`을 서비스 계층으로 던지게 되면, 서비스 계층이 `SQLException`이라는 JDBC 기술에 의존하게 되면서, 지금까지 우리가 고민했던 서비스 계층의 순수성이 무너진다.

이 문제를 해결하려면 앞서 배운 것 처럼 리포지토리에서 예외를 변환해서 던지면 된다.
`SQLException` ➡ `MyDuplicateKeyException`

먼저 필요한 예외를 만들어보자.

MyDuplicateKeyException

```
package hello.jdbc.repository.ex;

public class MyDuplicateKeyException extends MyDbException {

    public MyDuplicateKeyException() {
    }

    public MyDuplicateKeyException(String message) {
        super(message);
    }

    public MyDuplicateKeyException(String message, Throwable cause) {
        super(message, cause);
    }

    public MyDuplicateKeyException(Throwable cause) {
        super(cause);
    }
}
```

- 기존에 사용했던 `MyDbException`을 상속받아서 의미있는 계층을 형성한다. 이렇게하면 데이터베이스 관련 예외라는 계층을 만들 수 있다.
- 그리고 이름도 `MyDuplicateKeyException`이라는 이름을 지었다. 이 예외는 데이터 중복의 경우에만 던져야 한다.
- 이 예외는 우리가 직접 만든 것이기 때문에, JDBC나 JPA 같은 특정 기술에 종속적이지 않다. 따라서 이 예외를 사용하더라도 서비스 계층의 순수성을 유지할 수 있다. (향후 JDBC에서 다른 기술로 바꾸어도 이 예외는 그대로 유지할 수 있다.)

실제 예제 코드를 만들어서 확인해보자.

ExTranslatorV1Test

```
package hello.jdbc.exception.translator;

import hello.jdbc.domain.Member;
import hello.jdbc.repository.ex.MyDbException;
import hello.jdbc.repository.ex.MyDuplicateKeyException;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.springframework.jdbc.datasource.DriverManagerDataSource;

import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.Random;

import static hello.jdbc.connection.ConnectionConst.*;
import static org.springframework.jdbc.support.JdbcUtils.closeConnection;
import static org.springframework.jdbc.support.JdbcUtils.closeStatement;

public class ExTranslatorV1Test {

    Repository repository;
    Service service;

    @BeforeEach
    void init() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource(URL,
USERNAME, PASSWORD);
        repository = new Repository(dataSource);
        service = new Service(repository);
    }

    @Test
    void duplicateKeySave() {
        service.create("myId");
        service.create("myId");//같은 ID 저장 시도
```

```
    }

    @Slf4j
    @RequiredArgsConstructor
    static class Service {

        private final Repository repository;

        public void create(String memberId) {
            try {
                repository.save(new Member(memberId, 0));
                log.info("saveId={}", memberId);
            } catch (MyDuplicateKeyException e) {
                log.info("키 중복, 복구 시도");
                String retryId = generateNewId(memberId);
                log.info("retryId={}", retryId);
                repository.save(new Member(retryId, 0));
            } catch (MyDbException e) {
                log.info("데이터 접근 계층 예외", e);
                throw e;
            }
        }

        private String generateNewId(String memberId) {
            return memberId + new Random().nextInt(10000);
        }
    }

    @RequiredArgsConstructor
    static class Repository {

        private final DataSource dataSource;

        public Member save(Member member) {
            String sql = "insert into member(member_id, money) values(?, ?)";

            Connection con = null;
            PreparedStatement pstmt = null;
```

```
            try {
                con = dataSource.getConnection();
                pstmt = con.prepareStatement(sql);
                pstmt.setString(1, member.getMemberId());
                pstmt.setInt(2, member.getMoney());
                pstmt.executeUpdate();
                return member;
            } catch (SQLException e) {
                //h2 db
                if (e.getErrorCode() == 23505) {
                    throw new MyDuplicateKeyException(e);
                }
                throw new MyDbException(e);
            } finally {
                closeStatement(pstmt);
                closeConnection(con);
            }
        }

    }


}
```

실행해보면 다음과 같은 로그를 확인할 수 있다.

```
Service - saveId=myId
Service - 키 중복, 복구 시도
Service - retryId=myId492
```

같은 ID를 저장했지만, 중간에 예외를 잡아서 복구한 것을 확인할 수 있다.

리포지토리 부터 중요한 부분을 살펴보자.

```
} catch (SQLException e) {
    //h2 db
    if (e.getErrorCode() == 23505) {
```

```
            throw new MyDuplicateKeyException(e);
        }
        throw new MyDbException(e);
    }
```

- `e.getErrorCode() == 23505` : 오류 코드가 키 중복 오류(`23505`)인 경우 `MyDuplicateKeyException` 을 새로 만들어서 서비스 계층에 던진다.
- 나머지 경우 기존에 만들었던 `MyDbException` 을 던진다.

서비스의 중요한 부분을 살펴보자.

```
    try {
        repository.save(new Member(memberId, 0));
        log.info("saveId={}", memberId);
    } catch (MyDuplicateKeyException e) {
        log.info("키 중복, 복구 시도");
        String retryId = generateNewId(memberId);
        log.info("retryId={}", retryId);
        repository.save(new Member(retryId, 0));
    } catch (MyDbException e) {
        log.info("데이터 접근 계층 예외", e);
        throw e;
    }
```

- 처음에 저장을 시도한다. 만약 리포지토리에서 `MyDuplicateKeyException` 예외가 올라오면 이 예외를 잡는다.
- 예외를 잡아서 `generateNewId(memberId)` 로 새로운 ID 생성을 시도한다. 그리고 다시 저장한다. 여기가 예외를 복구하는 부분이다.
- 만약 복구할 수 없는 예외(`MyDbException`)면 로그만 남기고 다시 예외를 던진다.
  - 참고로 이 경우 여기서 예외 로그를 남기지 않아도 된다. 어차피 복구할 수 없는 예외는 예외를 공통으로 처리하는 부분까지 전달되기 때문이다. 따라서 이렇게 복구 할 수 없는 예외는 공통으로 예외를 처리하는 곳에서 예외 로그를 남기는 것이 좋다. 여기서는 다양하게 예외를 잡아서 처리할 수 있는 점을 보여주기 위해 이곳에 코드를 만들어두었다.

정리

- SQL ErrorCode로 데이터베이스에 어떤 오류가 있는지 확인할 수 있었다.
- 예외 변환을 통해 `SQLException` 을 특정 기술에 의존하지 않는 직접 만든 예외인 `MyDuplicateKeyException` 로 변환 할 수 있었다.
- 리포지토리 계층이 예외를 변환해준 덕분에 서비스 계층은 특정 기술에 의존하지 않는

`MyDuplicateKeyException`을 사용해서 문제를 복구하고, 서비스 계층의 순수성도 유지할 수 있었다.

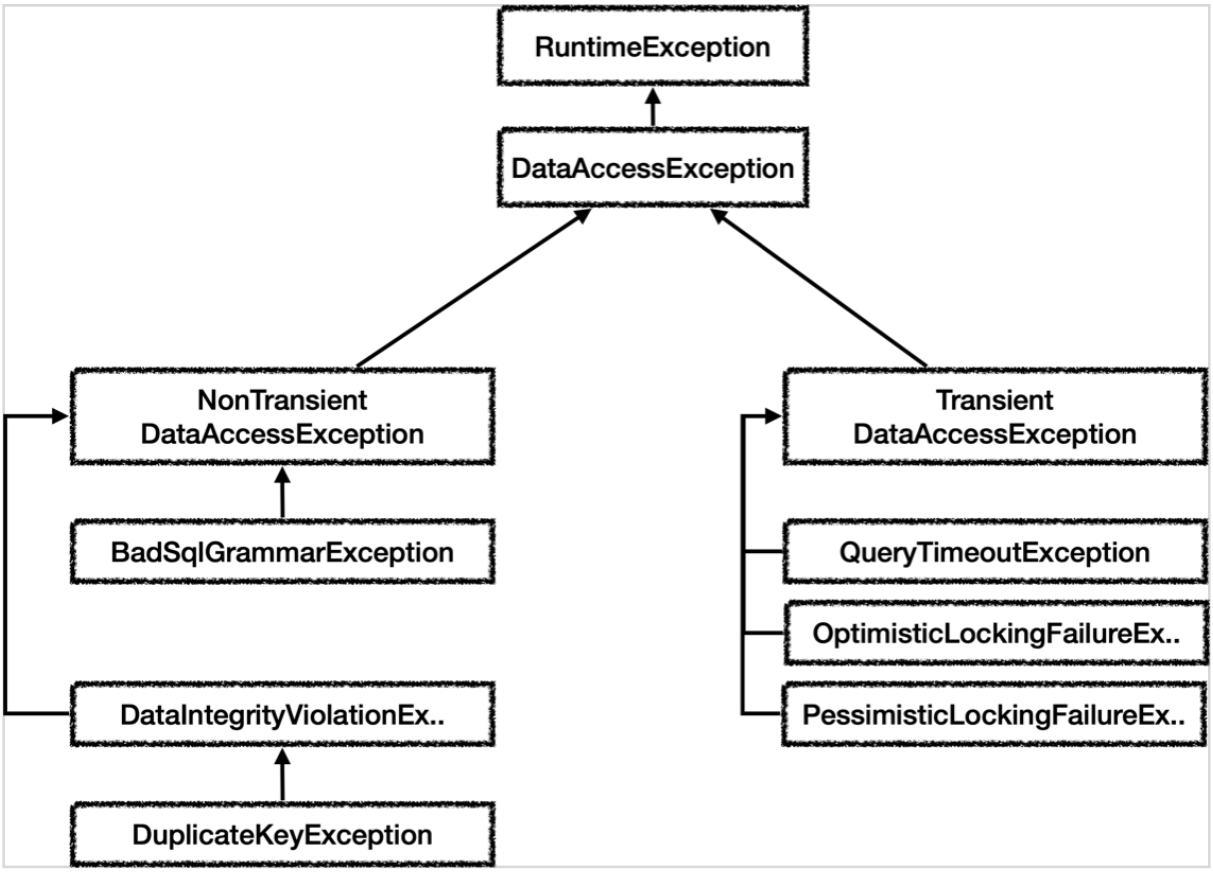
남은 문제

- SQL ErrorCode는 각각의 데이터베이스 마다 다르다. 결과적으로 데이터베이스가 변경될 때 마다 ErrorCode도 모두 변경해야 한다.
  - 예) 키 중복 오류 코드
    - H2: `23505`
    - MySQL: `1062`
- 데이터베이스가 전달하는 오류는 키 중복 뿐만 아니라 락이 걸린 경우, SQL 문법에 오류 있는 경우 등등 수십 수백가지 오류 코드가 있다. 이 모든 상황에 맞는 예외를 지금처럼 다 만들어야 할까? 추가로 앞서 이야기한 것 처럼 데이터베이스마다 이 오류 코드는 모두 다르다.

스프링 예외 추상화 이해

스프링은 앞서 설명한 문제들을 해결하기 위해 데이터 접근과 관련된 예외를 추상화해서 제공한다.

스프링 데이터 접근 예외 계층



- 스프링은 데이터 접근 계층에 대한 수십 가지 예외를 정리해서 일관된 예외 계층을 제공한다.
- 각각의 예외는 특정 기술에 종속적이지 않게 설계되어 있다. 따라서 서비스 계층에서도 스프링이 제공하는

예외를 사용하면 된다. 예를 들어서 JDBC 기술을 사용하든, JPA 기술을 사용하든 스프링이 제공하는 예외를 사용하면 된다.

- JDBC나 JPA를 사용할 때 발생하는 예외를 스프링이 제공하는 예외로 변환해주는 역할도 스프링이 제공한다.
- 참고로 그림을 단순화 하기 위해 일부 계층을 생략했다.

- 예외의 최고 상위는 `org.springframework.dao.DataAccessException` 이다. 그림에서 보는 것 처럼 런타임 예외를 상속 받았기 때문에 스프링이 제공하는 데이터 접근 계층의 모든 예외는 런타임 예외이다.
- `DataAccessException` 은 크게 2가지로 구분하는데 `NonTransient` 예외와 `Transient` 예외이다.
  - `Transient` 는 일시적이라는 뜻이다. `Transient` 하위 예외는 동일한 SQL을 다시 시도했을 때 성공할 가능성이 있다.
    - 예를 들어서 쿼리 타임아웃, 락과 관련된 오류들이다. 이런 오류들은 데이터베이스 상태가 좋아지거나, 락이 풀렸을 때 다시 시도하면 성공할 수 도 있다.
  - `NonTransient` 는 일시적이지 않다는 뜻이다. 같은 SQL을 그대로 반복해서 실행하면 실패한다.
    - SQL 문법 오류, 데이터베이스 제약조건 위배 등이 있다.

> 참고: 스프링 메뉴얼에 모든 예외가 정리되어 있지는 않기 때문에 코드를 직접 열어서 확인해보는 것이 필요하다.

스프링이 제공하는 예외 변환기

스프링은 데이터베이스에서 발생하는 오류 코드를 스프링이 정의한 예외로 자동으로 변환해주는 변환기를 제공한다.

코드를 통해 스프링이 제공하는 예외 변환기를 알아보자. 먼저 에러 코드를 확인하는 부분을 간단히 복습해보자.

SpringExceptionTranslatorTest

```
package hello.jdbc.exception.translator;

import lombok.extern.slf4j.Slf4j;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.springframework.dao.DataAccessException;
import org.springframework.jdbc.BadSqlGrammarException;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.jdbc.support.SQLErrorCodeSQLExceptionTranslator;
import org.springframework.jdbc.support.SQLExceptionTranslator;
```

```
import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;

import static hello.jdbc.connection.ConnectionConst.*;
import static org.assertj.core.api.Assertions.assertThat;

@Slf4j
public class SpringExceptionTranslatorTest {

    DataSource dataSource;

    @BeforeEach
    void init() {
        dataSource = new DriverManagerDataSource(URL, USERNAME, PASSWORD);
    }

    @Test
    void sqlExceptionErrorCode() {
        String sql = "select bad grammar";
        try {
            Connection con = dataSource.getConnection();
            PreparedStatement stmt = con.prepareStatement(sql);
            stmt.executeQuery();
        } catch (SQLException e) {
            assertThat(e.getErrorCode()).isEqualTo(42122);
            int errorCode = e.getErrorCode();
            log.info("errorCode={}", errorCode);
            //org.h2.jdbc.JdbcSQLSyntaxErrorException
            log.info("error", e);
        }
    }

}
```

- 이전에 살펴봤던 SQL ErrorCode를 직접 확인하는 방법이다. 이렇게 직접 예외를 확인하고 하나하나 스프링이 만들어준 예외로 변환하는 것은 현실성이 없다. 이렇게 하려면 해당 오류 코드를 확인하고

스프링의 예외 체계에 맞추어 예외를 직접 변환해야 할 것이다. 그리고 데이터베이스마다 오류 코드가 다르다는 점도 해결해야 한다.

- 그래서 스프링은 예외 변환기를 제공한다.

SpringExceptionTranslatorTest - 추가 exceptionTranslator

```
@Test
void exceptionTranslator() {

    String sql = "select bad grammar";

    try {
        Connection con = dataSource.getConnection();
        PreparedStatement stmt = con.prepareStatement(sql);
        stmt.executeQuery();
    } catch (SQLException e) {
        assertThat(e.getErrorCode()).isEqualTo(42122);
        //org.springframework.jdbc.support.sql-error-codes.xml

        SQLExceptionTranslator exTranslator = new
SQLErrorCodeSQLExceptionTranslator(dataSource);
        //org.springframework.jdbc.BadSqlGrammarException
        DataAccessException resultEx = exTranslator.translate("select", sql,
e);
        log.info("resultEx", resultEx);

assertThat(resultEx.getClass()).isEqualTo(BadSqlGrammarException.class);
    }
}
```

스프링이 제공하는 SQL 예외 변환기는 다음과 같이 사용하면 된다.

```
SQLExceptionTranslator exTranslator = new
SQLErrorCodeSQLExceptionTranslator(dataSource);
DataAccessException resultEx = exTranslator.translate("select", sql, e);
```

- `translate()` 메서드의 첫번째 파라미터는 읽을 수 있는 설명이고, 두번째는 실행한 sql, 마지막은 발생된 `SQLException`을 전달하면 된다. 이렇게 하면 적절한 스프링 데이터 접근 계층의 예외로 변환해서

반환해준다.

- 예제에서는 SQL 문법이 잘못되었으므로 `BadSqlGrammarException`을 반환하는 것을 확인할 수 있다.
  - 눈에 보이는 반환 타입은 최상위 타입인 `DataAccessException`이지만 실제로는 `BadSqlGrammarException` 예외가 반환된다. 마지막에 `assertThat()` 부분을 확인하자.
  - 참고로 `BadSqlGrammarException`은 최상위 타입인 `DataAccessException`를 상속 받아서 만들어진다.

각각의 DB마다 SQL ErrorCode는 다르다. 그런데 스프링은 어떻게 각각의 DB가 제공하는 SQL ErrorCode까지 고려해서 예외를 변환할 수 있을까?

비밀은 바로 다음 파일에 있다.

sql-error-codes.xml

```
<bean id="H2" class="org.springframework.jdbc.support.SQLErrorCodes">
  <property name="badSqlGrammarCodes">
      <value>42000,42001,42101,42102,42111,42112,42121,42122,42132</value>
  </property>
  <property name="duplicateKeyCodes">
      <value>23001,23505</value>
  </property>
</bean>
<bean id="MySQL" class="org.springframework.jdbc.support.SQLErrorCodes">
  <property name="badSqlGrammarCodes">
      <value>1054,1064,1146</value>
  </property>
  <property name="duplicateKeyCodes">
      <value>1062</value>
  </property>
</bean>
```

- `org.springframework.jdbc.support.sql-error-codes.xml`
- 스프링 SQL 예외 변환기는 SQL ErrorCode를 이 파일에 대입해서 어떤 스프링 데이터 접근 예외로 전환해야 할지 찾아낸다. 예를 들어서 H2 데이터베이스에서 `42000`이 발생하면 `badSqlGrammarCodes`이기 때문에 `BadSqlGrammarException`을 반환한다.

해당 파일을 확인해보면 10개 이상의 우리가 사용하는 대부분의 관계형 데이터베이스를 지원하는 것을 확인할 수 있다.

정리

- 스프링은 데이터 접근 계층에 대한 일관된 예외 추상화를 제공한다.
- 스프링은 예외 변환기를 통해서 `SQLException`의 `ErrorCode`에 맞는 적절한 스프링 데이터 접근 예외로 변환해준다.
- 만약 서비스, 컨트롤러 계층에서 예외 처리가 필요하면 특정 기술에 종속적인 `SQLException` 같은 예외를 직접 사용하는 것이 아니라, 스프링이 제공하는 데이터 접근 예외를 사용하면 된다.
- 스프링 예외 추상화 덕분에 특정 기술에 종속적이지 않게 되었다. 이제 JDBC에서 JPA같은 기술로 변경되어도 예외로 인한 변경을 최소화 할 수 있다. 향후 JDBC에서 JPA로 구현 기술을 변경하더라도, 스프링은 JPA 예외를 적절한 스프링 데이터 접근 예외로 변환해준다.
- 물론 스프링이 제공하는 예외를 사용하기 때문에 스프링에 대한 기술 종속성은 발생한다.
  - 스프링에 대한 기술 종속성까지 완전히 제거하려면 예외를 모두 직접 정의하고 예외 변환도 직접 하면 되지만, 실용적인 방법은 아니다.

스프링 예외 추상화 적용

이제 우리가 만든 애플리케이션에 스프링이 제공하는 데이터 접근 예외 추상화와 SQL 예외 변환기를 적용해보자.

MemberRepositoryV4_2

```
package hello.jdbc.repository;

import hello.jdbc.domain.Member;
import lombok.extern.slf4j.Slf4j;
import org.springframework.jdbc.datasource.DataSourceUtils;
import org.springframework.jdbc.support.JdbcUtils;
import org.springframework.jdbc.support.SQLErrorCodeSQLExceptionTranslator;
import org.springframework.jdbc.support.SQLExceptionTranslator;

import javax.sql.DataSource;
import java.sql.*;
import java.util.NoSuchElementException;

/**
 * SQLExceptionTranslator 추가
 */
@Slf4j
```

```java
public class MemberRepositoryV4_2 implements MemberRepository {

    private final DataSource dataSource;
    private final SQLExceptionTranslator exTranslator;

    public MemberRepositoryV4_2(DataSource dataSource) {
        this.dataSource = dataSource;
        this.exTranslator = new SQLErrorCodeSQLExceptionTranslator(dataSource);
    }

    @Override
    public Member save(Member member) {
        String sql = "insert into member(member_id, money) values(?, ?)";

        Connection con = null;
        PreparedStatement pstmt = null;

        try {
            con = getConnection();
            pstmt = con.prepareStatement(sql);
            pstmt.setString(1, member.getMemberId());
            pstmt.setInt(2, member.getMoney());
            pstmt.executeUpdate();
            return member;
        } catch (SQLException e) {
            throw exTranslator.translate("save", sql, e);
        } finally {
            close(con, pstmt, null);
        }
    }

    @Override
    public Member findById(String memberId) {
        String sql = "select * from member where member_id = ?";

        Connection con = null;
        PreparedStatement pstmt = null;
        ResultSet rs = null;
```

```
        try {
            con = getConnection();
            pstmt = con.prepareStatement(sql);
            pstmt.setString(1, memberId);

            rs = pstmt.executeQuery();

            if (rs.next()) {
                Member member = new Member();
                member.setMemberId(rs.getString("member_id"));
                member.setMoney(rs.getInt("money"));
                return member;
            } else {
                throw new NoSuchElementException("member not found memberId=" +
memberId);
            }

        } catch (SQLException e) {
            throw exTranslator.translate("findById", sql, e);
        } finally {
            close(con, pstmt, rs);
        }

    }

    @Override
    public void update(String memberId, int money) {

        String sql = "update member set money=? where member_id=?";

        Connection con = null;
        PreparedStatement pstmt = null;

        try {
            con = getConnection();
            pstmt = con.prepareStatement(sql);
            pstmt.setInt(1, money);
            pstmt.setString(2, memberId);
            pstmt.executeUpdate();
```

```
        } catch (SQLException e) {
            throw exTranslator.translate("update", sql, e);
        } finally {
            close(con, pstmt, null);
        }
    }

    @Override
    public void delete(String memberId) {

        String sql = "delete from member where member_id=?";

        Connection con = null;
        PreparedStatement pstmt = null;

        try {
            con = getConnection();
            pstmt = con.prepareStatement(sql);
            pstmt.setString(1, memberId);
            pstmt.executeUpdate();

        } catch (SQLException e) {
            throw exTranslator.translate("delete", sql, e);
        } finally {
            close(con, pstmt, null);
        }
    }

    private void close(Connection con, Statement stmt, ResultSet rs) {
        JdbcUtils.closeResultSet(rs);
        JdbcUtils.closeStatement(stmt);
        DataSourceUtils.releaseConnection(con, dataSource);
    }

    private Connection getConnection() {
        Connection con = DataSourceUtils.getConnection(dataSource);
        log.info("get connection={} class={}", con, con.getClass());
        return con;
```

```
    }

}
```

기존 코드에서 스프링 예외 변환기를 사용하도록 변경되었다.

```
} catch (SQLException e) {
    throw exTranslator.translate("save", sql, e);
}
```

MemberServiceV4Test - 수정

```
@Bean
MemberRepository memberRepository() {
    //return new MemberRepositoryV4_1(dataSource); //단순 예외 변환
    return new MemberRepositoryV4_2(dataSource); //스프링 예외 변환
}
```

- `MemberRepository` 인터페이스가 제공되므로 스프링 빈에 등록할 빈만 `MemberRepositoryV4_1` 에서 `MemberRepositoryV4_2` 로 교체하면 리포지토리를 변경해서 테스트를 확인할 수 있다.

정리

드디어 예외에 대한 부분을 깔끔하게 정리했다.

스프링이 예외를 추상화해준 덕분에, 서비스 계층은 특정 리포지토리의 구현 기술과 예외에 종속적이지 않게 되었다. 따라서 서비스 계층은 특정 구현 기술이 변경되어도 그대로 유지할 수 있게 되었다. 다시 DI를 제대로 활용할 수 있게 된 것이다.

추가로 서비스 계층에서 예외를 잡아서 복구해야 하는 경우, 예외가 스프링이 제공하는 데이터 접근 예외로 변경되어서 서비스 계층에 넘어오기 때문에 필요한 경우 예외를 잡아서 복구하면 된다.

JDBC 반복 문제 해결 - JdbcTemplate

지금까지 서비스 계층의 순수함을 유지하기 위해 수 많은 노력을 했고, 덕분에 서비스 계층의 순수함을 유지하게 되었다. 이번에는 리포지토리에서 JDBC를 사용하기 때문에 발생하는 반복 문제를 해결해보자.

JDBC 반복 문제

- 커넥션 조회, 커넥션 동기화
- `PreparedStatement` 생성 및 파라미터 바인딩
- 쿼리 실행
- 결과 바인딩
- 예외 발생시 스프링 예외 변환기 실행
- 리소스 종료

리포지토리의 각각의 메서드를 살펴보면 상당히 많은 부분이 반복된다. 이런 반복을 효과적으로 처리하는 방법이 바로 템플릿 콜백 패턴이다.
스프링은 JDBC의 반복 문제를 해결하기 위해 `JdbcTemplate` 이라는 템플릿을 제공한다.
`JdbcTemplate` 에 대한 자세한 사용법은 뒤에서 설명하겠다. 지금은 전체 구조와, 이 기능을 사용해서 반복 코드를 제거할 수 있다는 것에 초점을 맞추자.

MemberRepositoryV5

```
package hello.jdbc.repository;

import hello.jdbc.domain.Member;
import lombok.extern.slf4j.Slf4j;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;

import javax.sql.DataSource;

/**
 * JdbcTemplate 사용
 */
@Slf4j
public class MemberRepositoryV5 implements MemberRepository {

    private final JdbcTemplate template;

    public MemberRepositoryV5(DataSource dataSource) {
        template = new JdbcTemplate(dataSource);
    }

    @Override
```

```
    public Member save(Member member) {
        String sql = "insert into member(member_id, money) values(?, ?)";
        template.update(sql, member.getMemberId(), member.getMoney());
        return member;
    }

    @Override
    public Member findById(String memberId) {
        String sql = "select * from member where member_id = ?";
        return template.queryForObject(sql, memberRowMapper(), memberId);
    }

    @Override
    public void update(String memberId, int money) {
        String sql = "update member set money=? where member_id=?";
        template.update(sql, money, memberId);
    }

    @Override
    public void delete(String memberId) {
        String sql = "delete from member where member_id=?";
        template.update(sql, memberId);
    }

    private RowMapper<Member> memberRowMapper() {
        return (rs, rowNum) -> {
            Member member = new Member();
            member.setMemberId(rs.getString("member_id"));
            member.setMoney(rs.getInt("money"));
            return member;
        };
    }

}
```

MemberServiceV4Test - 수정

```
@Bean
```

```
MemberRepository memberRepository() {
    //return new MemberRepositoryV4_1(dataSource); //단순 예외 변환
    //return new MemberRepositoryV4_2(dataSource); //스프링 예외 변환
    return new MemberRepositoryV5(dataSource); //JdbcTemplate
}
```

`MemberRepository` 인터페이스가 제공되므로 등록하는 빈만 `MemberRepositoryV5` 로 변경해서 등록하면 된다.

`JdbcTemplate` 은 JDBC로 개발할 때 발생하는 반복을 대부분 해결해준다. 그 뿐만 아니라 지금까지 학습했던, **트랜잭션을 위한 커넥션 동기화**는 물론이고, 예외 발생시 **스프링 예외 변환기**도 자동으로 실행해준다.

> **참고**
> 템플릿 콜백 패턴에 대해서 지금은 자세히 이해하지 못해도 괜찮다. 스프링이 `JdbcTemplate` 이라는 편리한 기능을 제공하는 구나 정도로 이해해도 된다. 템플릿 콜백 패턴에 대한 자세한 내용은 **스프링 핵심 원리 - 고급편** 강의를 참고하자.

정리

완성된 코드를 확인해보자.

- 서비스 계층의 순수성
  - 트랜잭션 추상화 + 트랜잭션 AOP 덕분에 서비스 계층의 순수성을 최대한 유지하면서 서비스 계층에서 트랜잭션을 사용할 수 있다.
  - 스프링이 제공하는 예외 추상화와 예외 변환기 덕분에, 데이터 접근 기술이 변경되어도 서비스 계층의 순수성을 유지하면서 예외도 사용할 수 있다.
  - 서비스 계층이 리포지토리 인터페이스에 의존한 덕분에 향후 리포지토리가 다른 구현 기술로 변경되어도 서비스 계층을 순수하게 유지할 수 있다.
- 리포지토리에서 JDBC를 사용하는 반복 코드가 `JdbcTemplate` 으로 대부분 제거되었다.