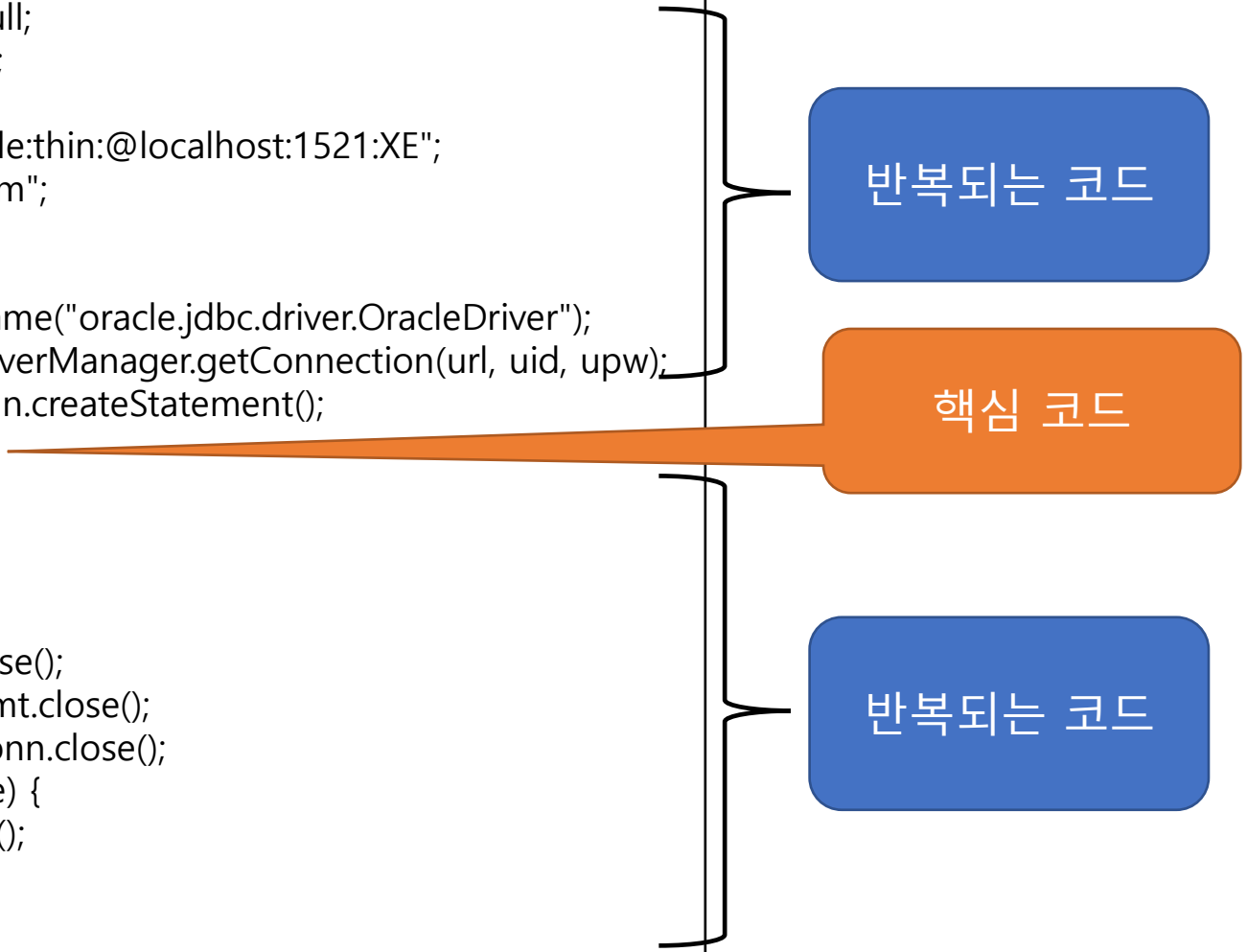
# 07강 DB 연동

# JDBC 프로그래밍의 단점을 보완하는 스프링

- 스프링 이전에 데이터베이스에 연동하기 위해서는 다음과 같은 코드를 작성해야 했다.

```
Connection conn = null;
Statement stmt = null;
ResultSet rs = null;
String url = "jdbc:oracle:thin:@localhost:1521:XE";
String uid = "hongteam";
String upw = "1234";
try {
        Class.forName("oracle.jdbc.driver.OracleDriver");
        conn = DriverManager.getConnection(url, uid, upw);
        stmt = conn.createStatement();
....
} catch (Exception e) {
    e.printStackTrace();
} finally {
    try {
      if (rs != null)rs.close();
      if (stmt != null)stmt.close();
      if (conn != null)conn.close();
    } catch (Exception e) {
       e.printStackTrace();
    }
}//finally의 끝
```



# JDBC 프로그래밍의 단점을 보완하는 스프링

- 위 코드에서 }로 표시 되는 부분은 실제 데이터 처리와는 관계없는 단순 연결을 위한 코드이다.
- 이런 구조적인 반복을 줄이기 위해 많이 사용되는 기법이 템플릿 메서드 패턴과 전략 패턴을 함께 사용하는 것인데 스프링은 이 두가지 기법을 하나로 엮은 jdbcTemplate클래스를 제공한다.

# JDBC 프로그래밍의 단점을 보완하는 스프링

- 또 얻을 수 있는 장점 – 트랜젝션 관리 부분
- 일반적으로 JDBC를 통해서 트랜젝션을 관리하려면

conn.setAutoCommit(false)를 통해서 자동 커밋을 비활성화하고

conn.commit()메서드나 conn.rollback()메서드를 통해서 트랜젝션을 통제해야 한다.

- 그러나 스프링에서는 트렌젝션을 적용하고 싶은 메서드에 @Transaction 애노테이션을 붙이면 된다.

# 프로젝트 준비

- POM.xmI에 다음관련 기능을 추가하고 메이븐 업데이트를 진행한다.

-> spring-jdbc : JDBCTemplate등 JDBC 연동에 필요한 기능

-> c3p0 : DB 커넥션 풀 기능을 제공

-> ojdbc6 : 오라클 DB에 연결하기 위한 JDBC 드라이버를 제공

추가로 트랜젝션 기능을 활용하려면 spring-tx도 필요하지만 spring-jdbc 와 의존 관계이므로 자동으로 다운로드가 된다.

# 프로젝트 준비

-spring.exception
    - AlreadyExistingMemberException.java
    - IdPasswordNotMatchingException.java
    - MemberNotFoundException.java
-spring.vo
    - Member.java
    - RegisterRequest.java
-spring.dao
    - MemberDao.java
-spring.service
    - MemberRegisterService.java
    - ChangPasswordService.java
-spring.printer
    - MemberPrinter.java
    - MemberInfoPrinter.java
    - MemberListPrinter.java
-spring.main

# 프로젝트 준비

- MemberDao.java

DAO 객체의 내부를 비운다.

```
public class MemberDao {

//  private static long nextId = 0;
//
//  private Map<String, Member> map = new HashMap<>();

    public Member selectByEmail(String email) {
        return null;
    }

    public void insert(Member member) {

    }

    public void update(Member member) {

    }

    public Collection<Member> selectAll() {
        return null;
    }
}
```

# 프로젝트 준비

- 데이터 테이블은 준비한다.

```
CREATE TABLE members(
    ID number primary key,
    EMAIL varchar2(255) unique,
    PASSWORD varchar2(100),
    NAME varchar2(100),
    REGDATE date
);

CREATE SEQUENCE members_seq
NOCACHE;
```

# 프로젝트 준비

- 샘플 테이터를 입력한다.

| 이메일 | 비밀번호 | 이름 | 가입일 |
|---|---|---|---|
| kim@naver.com | 1234 | 김길동 | 2020/05/13 |
| lee@naver.com | 1234 | 이길동 | 2020/04/04 |
| park@naver.com | 1234 | 박길동 | 2020/07/12 |
| hwang@naver.com | 1234 | 황길동 | 2020/10/03 |

# DataSource 설정

- 자바 JDBC는 데이터베이스와 연결을 위해서 DriverManager이외에도 DataSource를 이용해서 연결하는 방법을 정의한다.
- 스프링이 제공하는 DB 연동 기능은 DataSource를 활용해서 DBConnection은 구하도록 구현되어 있다.

- DataSource 기능을 제공하는 모듈로 dbcp나 c3p0등이 존재하는 데 이번에는 c3p0모듈을 활용해본다.

# DataSource 설정

- c3p0 모듈은 DataSource를 구현한 ComboPooledDataSource 클래스를 제공하므로 이 클래스를  스프링 빈으로 등록해서 사용한다.
- c3p0의 주요 프로퍼티

| 프로퍼티 | 설명 |
|---|---|
| initalPoolSize | 초기 커넥션 풀의 크기, 기본값 3 |
| maxPoolSize | 커넥션 풀의 최대 크기 기본값 15 |
| minPoolSize | 커넥션 풀의 최소 크기 기본값 3 |
| maxIdleTime | 지정한 시간동안 사용되지 않는 커넥션을 제거한다. 단위는 초, 기본값은 0 |
| checkoutTimeout | 풀에서 커넥션을 가져올 때 대기 시간, 1/1000초 기본값 0 (무한히 대기)<br>지정한 시간동안 커넥션을 가져오지 못하는 경우 SQLException발생 |
| idleConnectionTestPeriod | 풀 속에 있는 커넥션 테스트 주기, 단위는 초,<br>기본값은 0(0인 경우 검사하지 않는다. |

# jdbcTemplate를 이용한 쿼리 실행

- DataSource설정후 스프링이 제공하는 jdbcTemplate 클래스를 이용해서 DB 연동을 처리한다.

```
<!--  DataSource : 1. 연결 드라이버, 2.DB서버주소, 3. 계정ID, 4.  계정 PW -->
    <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource" destroy-method="close">
        <property name="driverClass" value="oracle.jdbc.OracleDriver"/>
        <property name="jdbcUrl" value="jdbc:oracle:thin:@db.interstander.com:41521:XE"/>
        <property name="user" value="green99"/>
        <property name="password" value="1234"/>
        <property name="maxPoolSize" value="30"/>
    </bean>
```

# jdbcTemplate를 이용한 쿼리 실행

- DAO 객체에 jdbcTemplate 생성 코드를 추가한다.

```
public class MemberDao {

    private JdbcTemplate jdbcTemplate;

    //생성자 주입
    public MemberDao(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }
```

# jdbcTemplate를 이용한 쿼리 실행

- 그후에 DAO 객체를 스프링 설정 파일에 등록한다.

```
<bean id="dao" class="spring.dao.MemberDao">
    <constructor-arg ref="dataSource" />
</bean>
```

# jdbcTemplate를 이용한 쿼리 실행

- JdbcTemplate 클래스는 SELECT 쿼리 실행을 위한 query() 메서드를 제공한다.
- 그중 자주 사용되는 쿼리는 다음과 같다.

```
• List<T> query(String sql, RowMapper<T> rowMapper)
• List<T> query(String sql, Object[] args, RowMapper<T> rowMapper)
• List<T> query(String sql, RowMapper<T> rowMapper, Object… args)
```

- 복잡하게 보이지만 간단히 보면
- sql 쿼리를 매개로 받아서 실행하고 RowMapper를 이용해서 ResultSet 의 결과를 자바 객체로 변환한다.
- sql 쿼리가 PreparedStatement인 경우 args 파라미터를 통해서 각 ?를 채울 값을 전달한다.

# jdbcTemplate를 이용한 쿼리 실행

- RowMapper의 mapRow()메서드는 SQL실행 결과로 구한 ResultSet으로부터 한 행의 데이터를 읽어와서 자바 객체로 변환해 주는 매퍼 기능을 구현한다.

- 보통 익명구현 클래스를 이용해서 RowMapper객체를 생성해 query()메서드에 전달해주는 방법이 일반적이다.

# jdbcTemplate를 이용한 쿼리 실행

- MemberDao의 selectByEmail()메서드를 구해보자.

```
public Member selectByEmail(String email) {
    List<Member> result = jdbcTemplate.query(
            "SELECT * FROM members WHERE email=?",
            new RowMapper<Member>() {

                @Override
                public Member mapRow(ResultSet rs, int rowNum) throws SQLException {
                    Member member = new Member(
                            rs.getString("email"),
                            rs.getString("password"),
                            rs.getString("name"),
                            rs.getTimestamp("regdate")
                            );
                    member.setId(rs.getLong("id"));
                    return member;
                }
            },email);
    return result.isEmpty()?null:result.get(0);
}
```

# jdbcTemplate를 이용한 쿼리 실행

- 같은 방법으로 selectAll()메서드를 구해보자

```
public List<Member> selectAll() {
    List<Member> results = jdbcTemplate.query(
            "SELECT * FROM MEMBERS ORDER BY id ASC",
            new RowMapper<Member>() {
                @Override
                public Member mapRow(ResultSet rs, int rowNum) throws SQLException {
                    Member member = new Member(
                            rs.getString("email"),
                            rs.getString("password"),
                            rs.getString("name"),
                            rs.getTimestamp("regdate")
                            );
                    member.setId(rs.getLong("id"));
                    return member;
                }

            });
        return results;
}
```

# jdbcTemplate를 이용한 쿼리 실행

- 결과가 무조건 1개인 조회 쿼리문 : queryForObject()

- 조회 결과가 다수의 행이 나온 경우 query()메서드를 사용할 수 있다.
- 그러나 어짜피 단 하나의 조회 결과만 나온다면 List로 처리할 것이 아니라 Integer형으로 처리하는 것이 편리하다.

- 테이블의 레코드 개수를 구하는 메서드를 작성해보자.

```
public int count() {
    Integer cnt = jdbcTemplate.queryForObject(
            "SELECT count(*) FROM Members",
            Integer.class);
    return cnt;
}
```

# jdbcTemplate를 이용한 쿼리 실행

- JdbcTemplate 클래스는 데이터 변경 관련 쿼리 실행을 위한 update() 메서드를 제공한다.

  - update(String sql)
  - update(String sql, Object…args)

- update() 메서드의 반환값은 변경된 행의 개수을 반환한다.

# jdbcTemplate를 이용한 쿼리 실행

- MemberDao 클래스의  update를 구현해보자

```
public void update(Member member) {
    jdbcTemplate.update(
            "UPDATE members set name=?, password=? WHERE Email=?",
            member.getName(),
            member.getPassword(),
            member.getEmail()
            );
}
```

# jdbcTemplate를 이용한 쿼리 실행

PreparedStatementCreator를 이용한 쿼리 실행

- 앞서 살펴본 update()메서드는 쿼리에 사용할 값을 매개값으로 전달해 주었다.

```
update("update members set NAME = ?, PASSWORD = ? where EMAIL = ?",
                        member.getName(), member.getPassword(), member.getEmail()
);
```

- 그러나 경우에 따라서 Servlet때 사용했던 set 메서드를 사용해서 직접 파라미터 값을 설정해 주어야 할 때도 있다.
- PreparedStatementCreator객체를 사용하면 connection을 가져와서 직접 PreparedStatement객체를 만들고 쿼리를 완성해서 반환한다.

# jdbcTemplate를 이용한 쿼리 실행

- insert메서드를 구현해 보자 – 예제7

```
public void insert(Member member) {
    jdbcTemplate.update(
            new PreparedStatementCreator() {

                @Override
                public PreparedStatement createPreparedStatement(Connection con) throws SQLException {
                    PreparedStatement psmt = con.prepareStatement(
                            "INSERT INTO members VALUES(members_seq.nextval,?,?,?,?)",
                            new String[] {"id"});

                    psmt.setString(1,member.getEmail());
                    psmt.setString(2,member.getPassword());
                    psmt.setString(3,member.getName());
                    psmt.setTimestamp(4,
                            new Timestamp(member.getRegisterDate().getTime()));

                    return psmt;
                }
            });
}
```

# jdbcTemplate를 이용한 쿼리 실행

- 기본키로 사용되는 컬럼은 대체로 시퀀스를 이용한 자동 증가값을 사용한다.
- 그런데 생성된 시퀀스의 값을 알고 싶을 때는 어떻게 할까.
- 이때는 KeyHolder를 사용한다.

- keyHolder를 사용하면 insert()메서드를 구현할 때 Member 객체의 ID값을 구할 수 있다.

- insert메서드를 구현해 보자 – 예제8

# jdbcTemplate를 이용한 쿼리 실행

- insert메서드를 구현해 보자 – 예제8

```
public void insert(Member member) {
    KeyHolder keyHolder = new GeneratedKeyHolder();

    jdbcTemplate.update(// update(PreparedStatementCreator객체,keyHolder);
            new PreparedStatementCreator() {

                @Override
                public PreparedStatement createPreparedStatement(Connection con) throws SQLException {
                    PreparedStatement psmt = con.prepareStatement(
                            "INSERT INTO members VALUES(members_seq.nextval,?,?,?,?)",
                            new String[] {"id"});

                    psmt.setString(1,member.getEmail());
                    psmt.setString(2,member.getPassword());
                    psmt.setString(3,member.getName());
                    psmt.setTimestamp(4,
                            new Timestamp(member.getRegisterDate().getTime()));

                    return psmt;
                }
            },keyHolder);
    Number keyValue = keyHolder.getKey();
    member.setId(keyValue.longValue());
}
```

# MemberDao 테스트 해보기

- 이제까지 jdbcTemplate객체를 이용해서 MemberDao 클래스의 코드를 구현했다.
- 이 MemberDao 클래스가 정상 작동하는지 Main01을 작성해서 확인해 보자

- 예제 9

- 추가적으로  delete메서드도 만들어보자

# 트랜젝션 처리

- 트랜젝션이란 하나의 논리적인 작업의 단위를 의미한다.
- 여러 물리적인 작업을 하나로 묶어 주며, 중간에 하나의 쿼리라도 실패하면 전체 쿼리 실패로 간주하고 실패 이전의 모든 쿼리를 취소한다.

- 이렇게 쿼리 실행 결과를 취소하고 기존 상태로 되돌리는 것을  rollback(롤백)라고 한다.
- 반대로 모든 쿼리가 성공적으로 작동해서 쿼리의 내용을 실제 DB에 반영하는 것을 commit(커밋)이라고 한다.

# 트랜젝션 처리

- 트랜젝션이 한번 시작되면 커밋하거나 롤백할 때 까지 실행되는 모든 쿼리가 하나의 작업의 단위가 된다.

```
Connection conn = null;
try{
    conn = DriverManager.getConnection(url,user,pwd);
    conn.setAutoCommit(false); //트랜젝션 시작
    ....쿼리 실행
    conn.commit(); //커밋, 트랜젝션 종료
}catch(SQLException err){
    if(conn!=null){
        try{
            conn.rollback(); //롤백, 트랜젝션 종료
        }
    }
}
```

- 위와 같은 방식으로 직접 트랜젝션으로 관리하면 실수의 가능성이 높아진다.

# 트랜젝션 처리 -@Transactional을 이용한 트랜젝션 처리

- 스프링이 제공하는 @Transactional 애노테이션을 이용하면 트랜젝션 범위를 쉽게 지정할 수 있다

예제 10 :ChangePasswordService 클래스

```
@Transactional
public void changePassword(String email, String oldPwd, String newPwd) {
    Member member = memberDao.selectByEmail(email);
    if (member == null)
        throw new MemberNotFoundException();

    member.changePassword(oldPwd, newPwd);

    memberDao.update(member);
}
```

# 트랜젝션 처리 -@Transactional을 이용한 트랜젝션 처리

- 다만 @Transactional 애노테이션이 제대로 동작하려면 다음의 두 가지 내용을 스프링 설정에 추가해야 한다.

⇒TransactionManager 빈 설정

⇒ @Transactional 애노테이션 활성화

⇒설정의 예 11

```
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>

<tx:annotation-driven transaction-manager="transactionManager"/>
```

# 트랜젝션 처리 -@Transactional 을 이용한 트랜젝션 처리

- PlatformTransactionManager 는 스프링이 제공하는 트랜젝션 관리자를 위한 인터페이스이다.
- 스프링은 구현 기술에 관계없이 동일한 방식으로 트랜젝션을 처리하기 위하여 이 인터페이스를 사용한다.

- <tx:annotation-driven>태그는 @Transactional 애노테이션이 붙은 메서드를 트랜젝션 범위에서 실행하는 기능을 활성화 한다.

- 앞서 작성한 ChangePasswordService클래스의 @Transactional 애노테이션이 잘 작동하는 지 확인하기 위해 스프링 설정파일에 빈으로 등록한다.

예제12

```
<bean id="changePwdSvc" class="spring.service.ChangePasswordService">
    <constructor-arg ref="dao" />
</bean>
```

# 트랜젝션 처리 -@Transactional 을 이용한 트랜젝션 처리

- 실제 암호 변경하는 Main클래스를 작성해본다.

예제 13

```
public static void main(String[] args) {
    AbstractApplicationContext ctx =
            new GenericXmlApplicationContext("classpath:appCtx.xml");

    ChangePasswordService cps =
            ctx.getBean("changePwdSvc", ChangePasswordService.class);
    try {
        cps.changePassword("kim@naver.com", "1234", "1111");
        System.out.println("암호를 변경했습니다.");
    } catch (MemberNotFoundException e) {
        System.out.println("회원 데이터가 존재하지 않습니다.");
    } catch (IdPasswordNotMatchingException e) {
        System.out.println("암호가 올바르지 않습니다.");
    }

    ctx.close();
}
```

# 트랜젝션 처리 -@Transactional 을 이용한 트랜젝션 처리

- 다만 실행 해보면 암호가 변경되었다는 메시지만 나올 뿐 실제 트랜젝션이 작동했는지 알수 없다.

- 그러므로 스프링에서 만들어내는 로그 메시지를 확인하기 위해 로깅 프레임워크(Log4j)를 사용하기로 한다.

- POM.xml에 Log4j를 추가한다.
- log4j.xml을 추가한다.

예제 14

```
<!-- https://mvnrepository.com/artifact/log4j/log4j -->
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>
```

# 트랜젝션 처리 - @Transactional 을 이용한 트랜젝션 처리

- 누가 트랜젝션을 어떻게 처리하는가?

- 트랜젝션도 공통의 기능중 하나이다.
- 스프링은 트랜젝션을 처리하기 위해 내부적으로 AOP를 사용한다.

- 실제로 @Transactional 을 적용하기 위해 <tx:annotation-driven>태그를 사용하면 스프링은 내부적으로 @Transactional 가 적용된 빈 객체를 찾아서 그에 알맞은 프록시 객체를 생성한다.

# 트랜젝션 처리 - @Transactional 을 이용한 트랜젝션 처리

- 이때 프록시 객체는 @Transactional 가 붙은 메서드를 호출하면 PlatformTransactionManager 를 사용해서 트랜젝션을 시작한다.
- 트랜젝션을 시작한 다음 실제 객체의 메서드를 실행하고 성공적으로 실행시 커밋, 그렇지 않으면 롤백한다.

- 별도의 설정을 추가하지 않으면 RuntimeException과 그 하위 예외 발생시 롤백을 수행한다.