JDBC와 Spring 데이터베이스 연동 지원

JDBC 개념과 역할

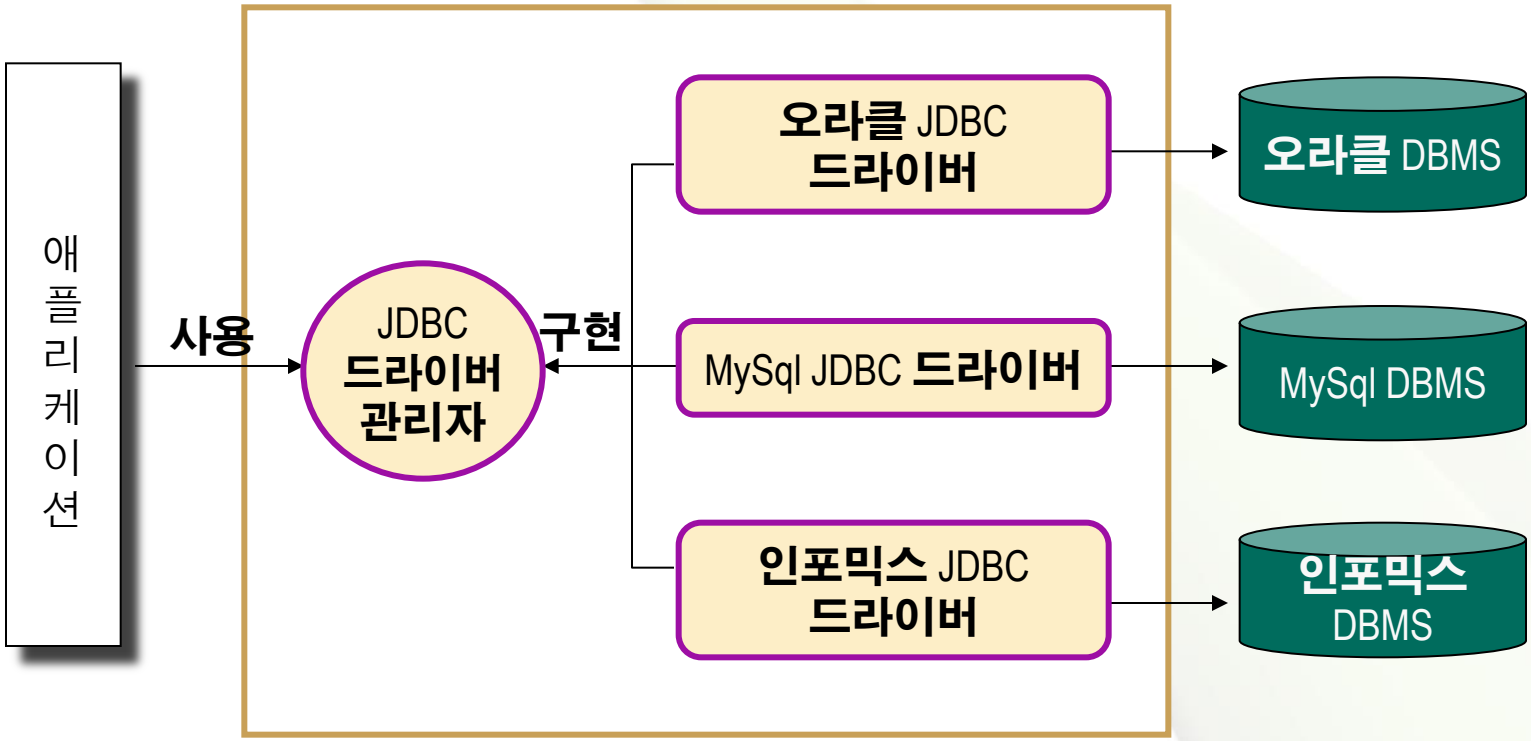
- **Java Database Connectivity**

- 자바애플리케이션에서 표준화된 데이터베이스 접근 제공.

- 각 데이터베이스 접속에 대한 상세한 정보를 추상화.

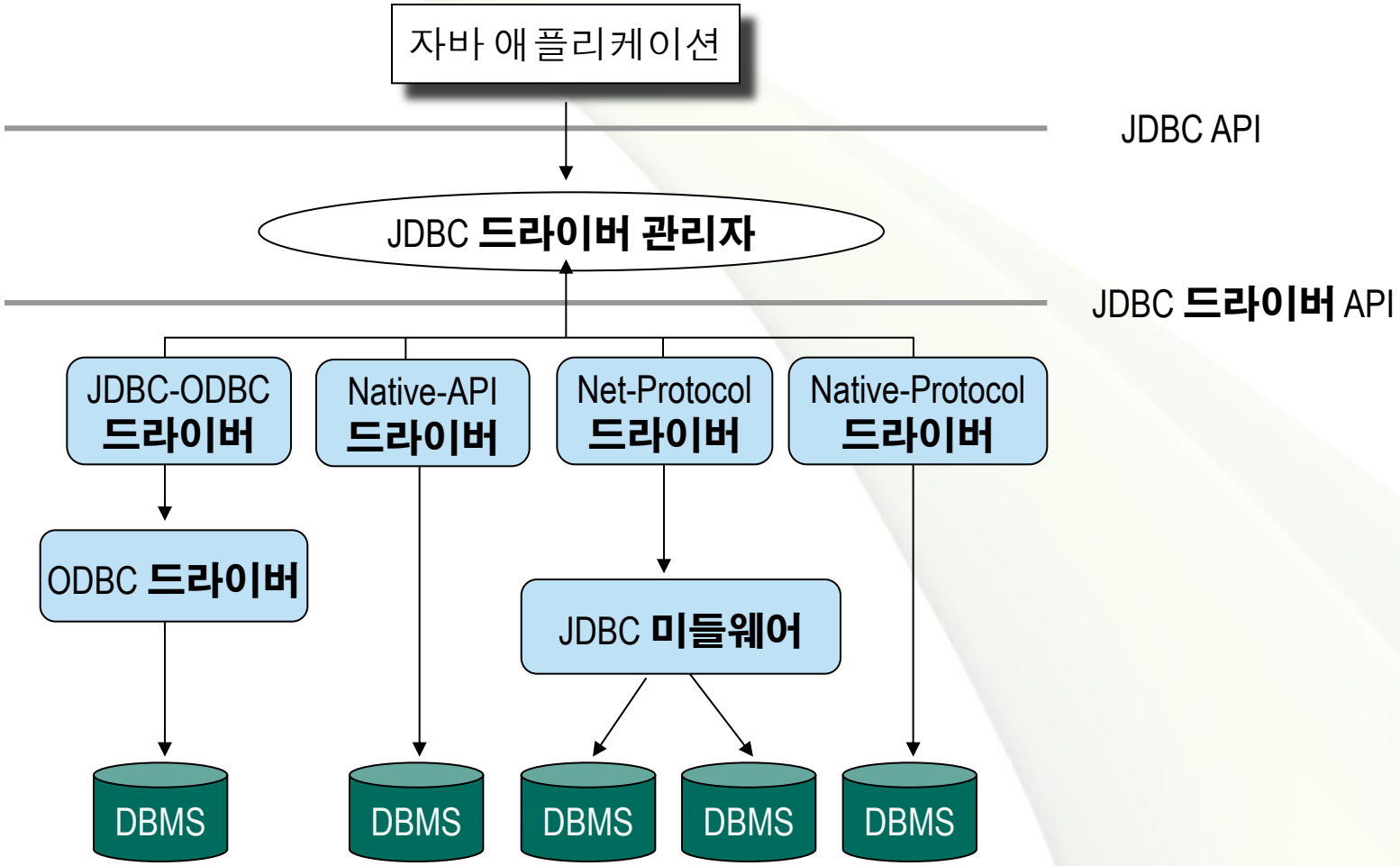
- 이론적으로는 개발된 애플리케이션에서 **DB** 변경시 **JDBC** 드라이버 교체만으로 가능

JDBC 구성


애플리케이션
사용
JDBC 드라이버 관리자
구현
오라클 JDBC 드라이버
MySql JDBC 드라이버
인포믹스 JDBC 드라이버
오라클 DBMS
MySql DBMS
인포믹스 DBMS

JDBC 드라이버 유형

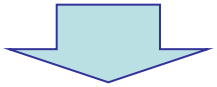
- JDBC 드라이버 구성도



JDBC 드라이버 설치

- **JDBC** 드라이버 선택
  - **JDBC** 드라이버는 사용하고자 하는 데이터베이스 벤더 별로 제공 됨

- 설치 디렉터리(다음 중 한 가지를 이용)
  - **JDK**설치디렉터리**\jre\lib\ext** 에 복사하는 방법.
  - 톰캣설치디렉터리**\common\lib** 폴더에 복사하는 방법
  - 이클립스 프로젝트의 **WebContent\WEB-INF\lib** 폴더에 복사하는 방법



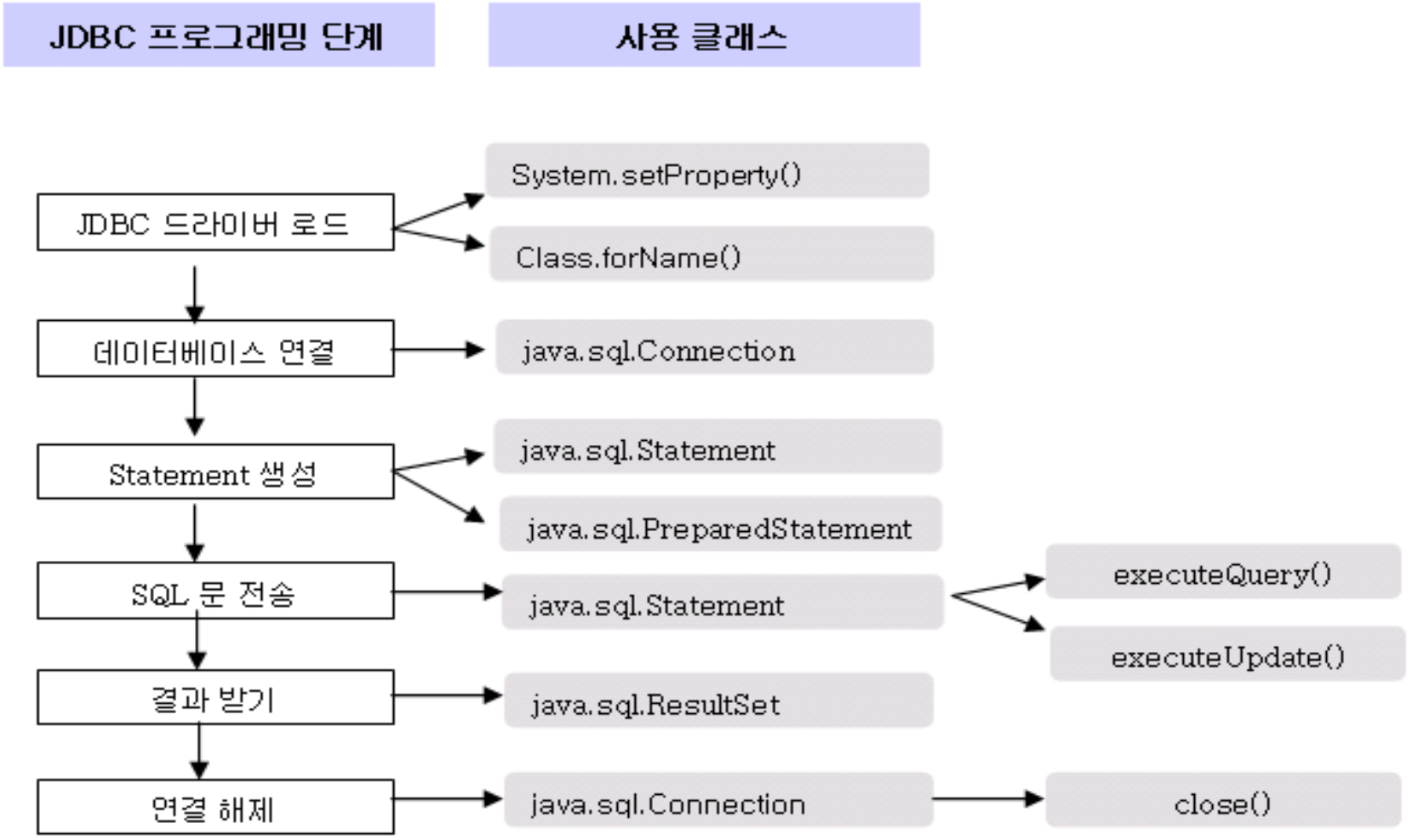
ebContent₩WEB-INF₩lib 폴더에 에 설치

  - **Maven**을 사용하는 경우 **pom.xml** 파일에 의존성 패키지 등록

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.33</version>
</dependency>
```

JDBC 프로그래밍 과정

▪JDBC 프로그래밍 단계



JDBC 프로그래밍 단계

- 데이터베이스 드라이버 로드

```
• DriverManager.RegisterDriver(new com.mysql.jdbc.Driver());
• Class.forName( “com.mysql.jdbc.Driver” );
• Class.forName( “com.mysql.jdbc.Driver” ).newInstance();
• System.setProperty( “jdbc.drivers” , “com.mysql.jdbc.Driver” );
```

- 데이터베이스 연결

```
Connection conn =
 DriverManger.getConnection( “JDBC_url” , ” 아이디”, ” 비밀번호”);
```

  - JDBC_URL 구성 = jdbc:mysql://ip:port/db-name

JDBC 프로그래밍 단계

- **Statement** 생성 및 쿼리 실행
  - **Statement** 객체 생성후 **SQL** 문장을 변수 처리부와 함께 문자열로 구성
  - 쿼리가 복잡해질수록 성능 저하 및 관리에 어려움이 있음

```
Statement stmt = conn.createStatement();
stmt.executeUpdate(
            "insert into test values( ' " +
           request.getParameter("username") + "','" +
           request.getParameter("email")+"')");
```

JDBC 프로그래밍 단계

- **PreparedStatement** 생성 및 쿼리 실행
  - **PreparedStatement** 객체 생성시 **SQL** 문장을 미리 생성하고 **Parameter**는 별도의 메서드로 대입하는 방식으로 성능과 관리 면에서 모두 권장 되는 방식

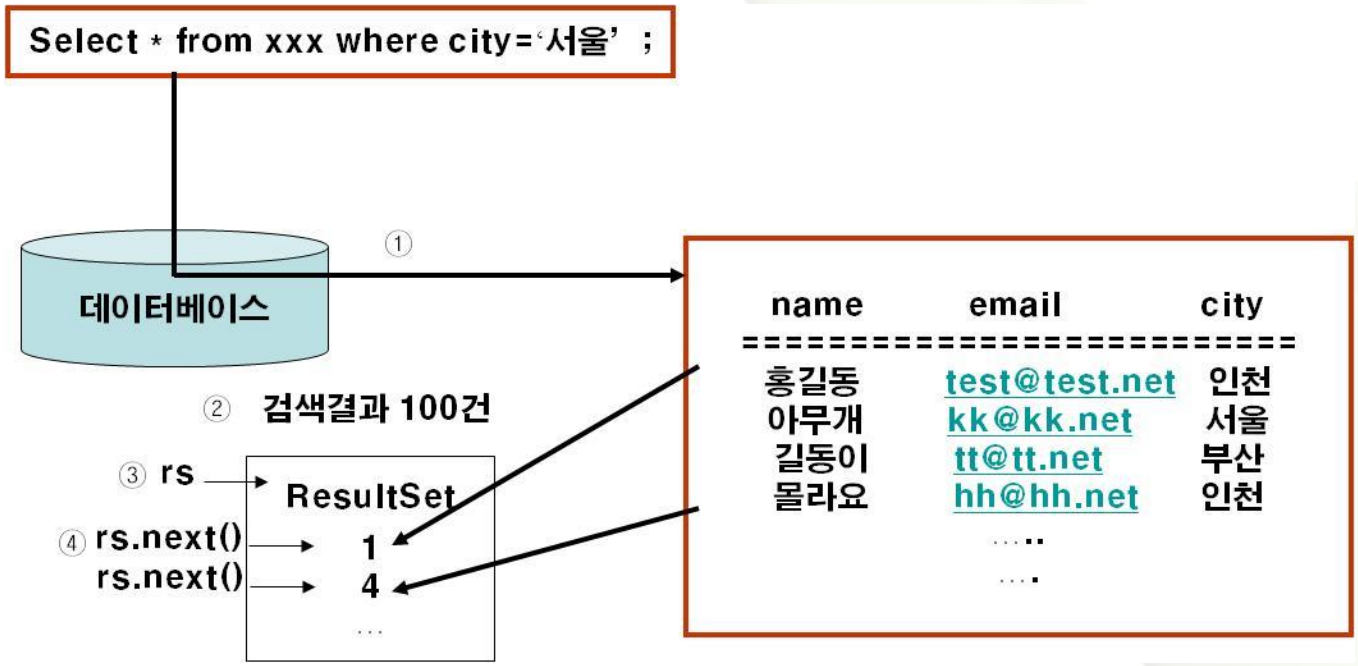
```
PreparedStatement pstmt =
        conn.prepareStatement( “insert into test values(?,?)” );
pstmt.setString(1,request.getParameter("username");
pstmt..setString(2, request.getParameter("email");
pstmt.executeUpdate();
```

JDBC 프로그래밍 단계

- 결과 받기

```
ResultSet rs = pstmt.executeQuery( );
```

  - **ResultSet**은 커서 개념의 연결 포인터
  - **next()**메서드를 통해 로우 이동



JDBC 프로그래밍 단계

- 결과 받기

```
ResultSet rs = pstmt.executeQuery();

while(rs.next()) {
    name = rs.getString(1); // or rs.getString( “name” );
    age = rs.getInt(2); // or rs.getInt( “email” );
}

rs.close();
```

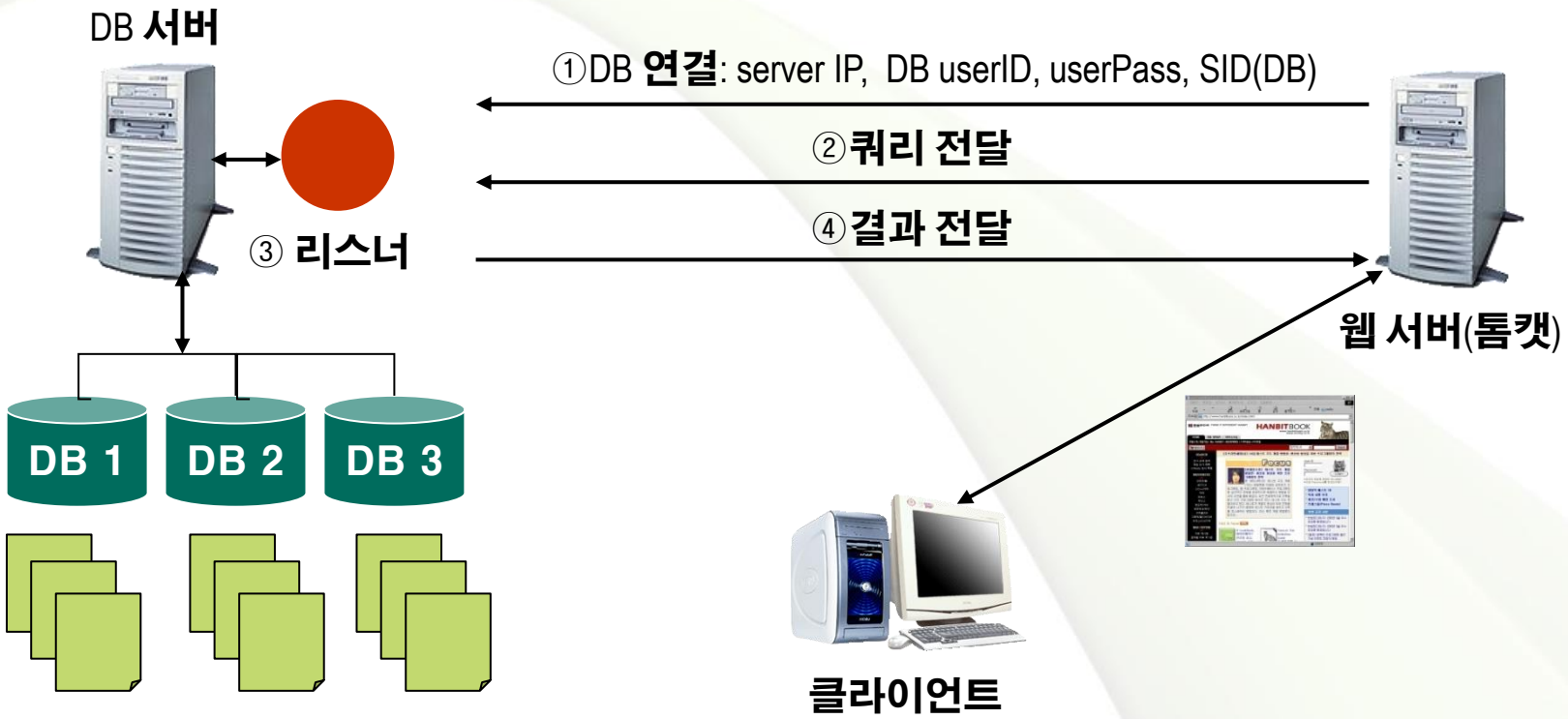
JDBC 프로그래밍 단계

- 연결해제
  - **Connection** 을 **close()**해주지 않으면 사용하지 않는 연결이 유지되어 **DB** 자원 낭비.

```
conn.close();
```

JDBC 프로그래밍 단계

- JDBC 프로그래밍 동작 과정 – 정리



저수준 JDBC 코드의 문제

- 구조 코드로 인한 코드량 증가
  - 예외처리 필수
  - 드라이버 등록, 연결 생성, 명령생성, 명령실행, 연결닫기 등의 표준 API 호출
  - 실제 변경되는 내용은 SQL과 전달인자 및 결과 처리 코드

- 데이터 구조 불일치로 인한 효율성 저하
  - 프로그램의 객체와 데이터베이스의 테이블 사이의 호환성 문제
    - 데이터 타입 불일치
    - 관계 불일치
    - 입자성 불일치
    - 상속성 불일치
    - 식별 불일치

Spring 데이터베이스 연동 지원

- 템플릿 클래스를 통한 데이터 접근 지원
  - 동일한 코드의 중복을 제거하고 필요한 최소한의 내용으로 데이터베이스 연동 코드 작성 가능
- 의미 있는 예외 클래스 제공
  - 데이터베이스 연동 과정 중에 발생하는 SQLException을 대체하고 오류의 원인을 예측할 수 있는 다양한 예외 클래스 제공
- 트랜잭션 처리 지원
  - 데이터베이스 연동 기술에 상관 없이 동일한 방식으로 트랙잭션 처리가 가능한 프로그래밍 기법 제공
  - 코드 기반 트랜잭션 및 선언적 트랜잭션 지원

Spring JDBC 설치

- 의존성 패키지 등록 (pom.xml)
  - jdbc 패키지

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>${org.springframework-version}</version>
</dependency>
```

  - 트랜잭션 패키지

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
    <version>${org.springframework-version}</version>
</dependency>
```

Spring DataSource 설정 (연결 설정)

- Spring은 템플릿 클래스 및 ORM 프레임워크 연동 클래스를 사용할 경우 DataSource를 통해 Connection 제공

- 제공 방식
  - 커넥션 풀을 이용한 DataSource 설정
  - JNDI를 이용한 DataSource 설정
  - DriverManager를 이용한 DataSource 설정

커넥션 풀을 이용한 DataSource 설정

- 스프링이 직접 커넥션 풀 구현 클래스를 제공하지는 않지만 Apache Commons DBCP와 같은 커넥션 풀 라이브러리를 이용해서 커넥션 풀 기반의 DataSource 설정 가능

- DBCP 의존성 패키지 등록

```
<dependency>
    <groupId>commons-dbcp</groupId>
    <artifactId>commons-dbcp</artifactId>
    <version>1.4</version>
</dependency>
```

- 스프링 빈 설정

```
<bean id="dataSource"
    class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost:3306/order_system" />
    <property name="username" value="root" />
    <property name="password" value="1234" />
</bean>
```

JNDI를 이용한 DataSource 설정

- WebLogic, Jboss와 같은 JEE 애플리케이션 서버 및 톰캣과 같은 웹 컨테이너 등이 지원하는 JNDI를 이용해서 DataSource 반환

- 컨테이너 데이터 소스 객체 등록 또는 활성화
  - 여기서는 Tomcat server.xml 파일의 컨텍스트 정보 수정

```
<Context docBase="SpringJdbc-05.SpringMVCOrder" path="/spring-jdbc-05-order" reloadable="true" so
  <Resource name="jdbc/order-system"
      auth="Container" type="javax.sql.DataSource"
      maxTotal="100" maxIdle="30" maxWaitMillis="10000"
      username="root" password="1234"
      driverClassName="com.mysql.jdbc.Driver" url="jdbc:mysql://localhost:3306/order-system"/>
</Context>
```

- Spring 빈 설정 파일에서 등록된 객체 참조

```
<jee:jndi-lookup jndi-name="/jdbc/order_system" id="dataSource"/>
```

DriverManager를 이용한 DataSource 설정

- 커넥션 풀이나 JNDI를 사용할 수 없을 경우 DriverManager를 이용해서 커넥션을 제공하는 DriverManagerDataSource 클래스 사용

```
<bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost:3306/order_system" />
    <property name="username" value="root" />
    <property name="password" value="1234" />
</bean>
```

데이터 소스 설정으로부터 정보 분리

- 데이터 소스에 직접 정보를 기록하는 것보다 설정 파일 등을 사용해서 데이터를 분리하는 것이 권장됨

- 설정 파일 작성 (environment.properties -- resources 폴더에 저장)

```
dataSource.driverClassName=com.mysql.jdbc.Driver
dataSource.url=jdbc:mysql://localhost:3306/order_system
dataSource.username=root
dataSource.password=1234
```

- 설정 파일 읽기
  - PropertyPlaceholderConfigurer 빈을 등록해서 설정 파일 읽기

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="location"  value="classpath:environment.properties" />
    <property fileEncoding="UTF-8" />
</bean>
```

  - <context:property-placeholder />요소를 사용해서 설정 파일 읽기

```
<context:property-placeholder file-encoding="UTF-8"
    location="classpath:environment.properties" />
```

데이터 소스 설정으로부터 정보 분리

- 설정 파일에서 읽은 데이터를 사용해서 데이터 소스 설정

```
<bean id="dataSource"
    class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName" value="${dataSource.driverClassName}"/>
    <property name="url"  value="${dataSource.url}" />
    <property name="username" value="${dataSource.username}" />
    <property name="password" value="${dataSource.password}" />
</bean>
```

DataSource로 부터 커넥션 구하기

- 사용할 클래스에 필드 선언 후 의존성 주입 처리

```
public class MyClass implements MyInterface {

    @Autowired
    private DataSource dataSource;
    public setDataSource (DataSource dataSource) {
        this.dataSource = dataSource;
    }

    public void testMethod() {
        Connection conn = null;
        try {
            conn = dataSource.getConnection(); //트랜잭션 활용 불가능
            conn = DataSourceUtils.getConnection(dataSource);//트랜잭션 활용 가능
            …
        } finally {
            JdbcUtils.closeConection(conn);// 트랜잭션 사용하지 않는 경우
            DataSourceUtils.releaseConnection(conn, dataSource); //트랜잭션 사용
        }
    }
}
```

스프링 템플릿을 이용한 JDBC 지원

- 연결객체 획득, 예외처리 등 중복코드를 제거하고 효과적인 데이터베이스 연동 코드 작성을 위해 템플릿 지원

- 종류
  - JdbcTemplate
    - SQL 실행을 위한 다양한 메서드 제공
    - 인덱스 기반 전달인자 사용
  - NamedParameterJdbcTemplate
    - 인덱스 기반 전달인자가 아닌 이름 기반의 전달인자 사용 지원
    - 이를 위해 Map이나 SqlParameterSource 등을 사용

JDBC 템플릿 의존성 주입

- 빈 설정 파일 또는 어노테이션을 이용한 의존성 주입

```
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate">
    <constructor-arg ref="dataSource"/>
</bean>

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

```
@Autowired
private JdbcTemplate jdbcTemplate;
```

- JdbcDaoSupport / NamedParameterJdbcDaoSupport 클래스 상속

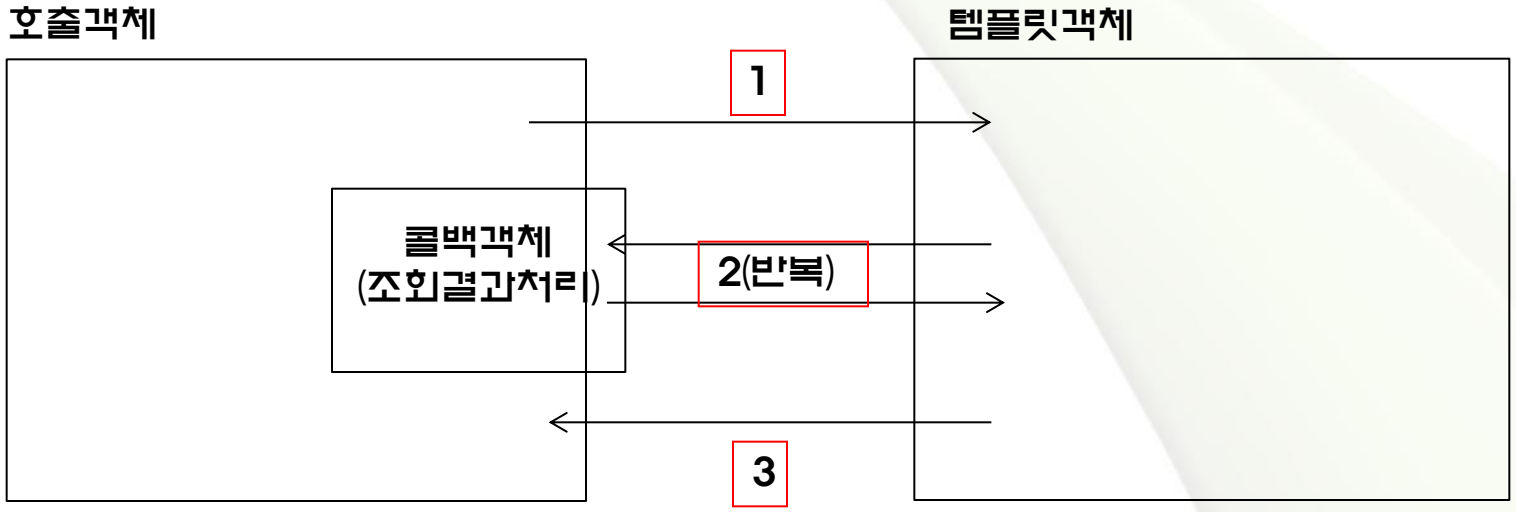
```
@Repository("customerRepository")
public class CustomerRepositoryJdbcDaoSupport extends NamedParameterJdbcDaoSupport
    implements CustomerRepository {

@Override
public CustomerEntity findOne(long id) {
    Map<String, Object> params = new HashMap<String, Object>();
    params.put("id", id);
    NamedParameterJdbcTemplate template = getNamedParameterJdbcTemplate();
    return template.queryForObject(SQL_GETBYID, params, new CustomerRowMapper());
}
```

템플릿 구조의 작동 원리

- 반복되는 구조를 분리해서 별도의 클래스로 정의하고 변경되는 내용을 전달해서 기능을 처리하는 기법

- SQL, Parameter 매핑 데이터, 조회 결과를 처리할 객체 참조를 전달인자로 제공하면 템플릿의 구조 코드에서 이 전달인자를 사용해서 전체 데이터 연동 코드 수행



스프링 템플릿을 이용한 JDBC 지원

- JdbcTemplate 사용 1 (insert, update, delete, select)

```
@Override
public int count() {
    return jdbcTemplate
     .queryForInt("select count(*) from GUESTBOOK_MESSAGE");
}
@Override
public List<GuestMessage> select(int begin, int end) {
    int startRowNum = begin - 1;
    int count = end - begin + 1;
    return jdbcTemplate
     .query(
             "select * from GUESTBOOK_MESSAGE order by MESSAGE_ID desc limit ?, ?",
             new Object[] { startRowNum, count },
             new GuestMessageRowMapper());
}
@Override
public int delete(int id) {
    return jdbcTemplate.update(
     "delete from GUESTBOOK_MESSAGE where MESSAGE_ID = ?", id);
}
@Override
public int update(GuestMessage message) {
    return jdbcTemplate
     .update(
             "update GUESTBOOK_MESSAGE set MESSAGE = ? where MESSAGE_ID = ?",
             new Object[] { message.getMessage(), message.getId() },
             new int[] { Types.VARCHAR, Types.INTEGER });
}
```

스프링 템플릿을 이용한 JDBC 지원

- JdbcTemplate 사용 2 (자동증가컬럼을 포함하는 insert)

```
@Override
public int insert(final GuestMessage message) {
    KeyHolder keyHolder = new GeneratedKeyHolder();
    int insertedCount = jdbcTemplate.update(new PreparedStatementCreator() {

        @Override
        public PreparedStatement createPreparedStatement(Connection con)
                throws SQLException {
            PreparedStatement pstmt = con
                    .prepareStatement(
                            "insert into GUESTBOOK_MESSAGE (GUEST_NAME, MESSAGE, REGISTRY_DATE) values (?, ?, ?)",
                            new String[] { "MESSAGE_ID" });
            pstmt.setString(1, message.getGuestName());
            pstmt.setString(2, message.getMessage());
            pstmt.setTimestamp(3, new Timestamp(message.getRegistryDate()
                    .getTime()));
            return pstmt;
        }
    }, keyHolder);
    Number keyNumber = keyHolder.getKey();
    message.setId(keyNumber.intValue());
    return insertedCount;
    // int insertedCount = jdbcTemplate
    // .update(
    // "insert into GUESTBOOK_MESSAGE (GUEST_NAME, MESSAGE, REGISTRY_DATE) values (?, ?, ?)",
    // message.getGuestName(), message.getMessage(), message
    // .getRegistryDate());
    // if (insertedCount > 0) {
    // int id = jdbcTemplate.queryForInt("select last_insert_id() ");
    // message.setId(id);
    // }
    // return insertedCount;
}
```

스프링 템플릿을 이용한 JDBC 지원

- NamedParameterJdbcTemplate 사용 (insert, update, delete)

```
@Override
public int delete(int id) {
    Map<String, Object> paramMap = new HashMap<String, Object>();
    paramMap.put("id", id);
    return template.update(
            "delete from GUESTBOOK_MESSAGE where MESSAGE_ID = :id",
            paramMap);
}
@Override
public int insert(GuestMessage message) {
    BeanPropertySqlParameterSource paramSource = new BeanPropertySqlParameterSource(
            message);
    int insertedCount = template.update(
            "insert into GUESTBOOK_MESSAGE (GUEST_NAME, MESSAGE, REGISTRY_DATE) values "
                    + "(:guestName, :message, :registryDate)", paramSource);
    if (insertedCount > 0) {
        int id = template.queryForInt("select last_insert_id() ",
                Collections.<String, Object> emptyMap());
        message.setId(id);
    }
    return insertedCount;
}
@Override
public int update(GuestMessage message) {
    MapSqlParameterSource paramSource = new MapSqlParameterSource();
    paramSource.addValue("message", message.getMessage());
    paramSource.addValue("id", message.getId(), Types.INTEGER);
    return template
            .update(
                    "update GUESTBOOK_MESSAGE set MESSAGE = :message where MESSAGE_ID = :id",
                    paramSource);
}
```

스프링 템플릿을 이용한 JDBC 지원

- NamedParameterJdbcTemplate 사용 (select)

```
@Override
public List<GuestMessage> select(int begin, int end) {
    Map<String, Object> paramMap = new HashMap<String, Object>();
    paramMap.put("startRowNum", begin - 1);
    paramMap.put("count", end - begin + 1);
    return template
            .query(
                    "select * from GUESTBOOK_MESSAGE order by MESSAGE_ID desc limit :startRowNum, :count",
                    paramMap, new RowMapper<GuestMessage>() {

                        @Override
                        public GuestMessage mapRow(ResultSet rs, int rowNum)
                                throws SQLException {
                            GuestMessage message = new GuestMessage();
                            message.setId(rs.getInt("MESSAGE_ID"));
                            message
                                    .setGuestName(rs
                                            .getString("GUEST_NAME"));
                            message.setMessage(rs.getString("MESSAGE"));
                            message.setRegistryDate(rs
                                    .getDate("REGISTRY_DATE"));
                            return message;
                        }

                    });
}
```