21. JDBC (Java Database Connectivity)

드라이버 로딩과 연결

Java에서 데이터베이스와 연결하려면 JDBC(Java Database Connectivity)를 사용해야 한다. JDBC는 자바 프로그램에서 데이터베이스에 접근할 수 있게 해주는 **API** 집합이고, 그 첫 번째 단계가 바로 **드라이버 로딩과 연결(connection)** 단계이다. 아래에 그 과정을 구체적으로 설명한다.

✅ 1. JDBC 드라이버란?

- DBMS(Oracle, MySQL, PostgreSQL 등)마다 JDBC 드라이버가 필요함
- 드라이버는 자바와 DB 사이에서 **통신을 중개**하는 라이브러리 (.jar 파일)
- 보통 다음 경로에서 제공됨:
  - MySQL: `mysql-connector-java`
  - PostgreSQL: `postgresql.jar`
  - Oracle: `ojdbc11.jar`

✅ 2. 드라이버 로딩 방식

📌 방법 1: `Class.forName()` 사용 (전통적인 방식)

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

- JVM이 클래스 로딩 시 드라이버 등록 (`DriverManager.registerDriver()` 호출됨)
- Java 6 이후는 생략 가능 (서비스 프로바이더 메커니즘이 대신 처리)

📌 방법 2: 자동 로딩 (Java 6 이상)

JDBC 4.0 이후부터는 드라이버 `.jar` 파일에 `META-INF/services/java.sql.Driver` 파일이 존재하면, JVM이 자동으로 드라이버 클래스를 로딩함.

- 즉, `Class.forName()` 생략 가능
- 단, 드라이버 `.jar` 파일이 **classpath** 또는 **모듈 경로**에 포함돼야 함

✅ 3. 연결(Connection) 열기

📌 `DriverManager.getConnection()` 사용

```
String url = "jdbc:mysql://localhost:3306/mydb";
String user = "root";
String password = "1234";

Connection conn = DriverManager.getConnection(url, user, password);
```

- 연결 성공 시 `Connection` 객체 반환
- DB URL은 다음 형식을 따름:

DBMS	URL 형식 예시
MySQL	`jdbc:mysql://host:port/dbname`
PostgreSQL	`jdbc:postgresql://host:port/dbname`
Oracle	`jdbc:oracle:thin:@host:port:sid` or `:service`

✅ 4. 연결에 실패하는 경우

- 드라이버 `.jar`가 누락된 경우 → `ClassNotFoundException`
- DB 접근 불가, URL 오타 → `SQLException`
- 인증 실패 → `Access denied` 오류

✅ 5. 전체 예제

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DBConnectExample {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/mydb";
        String user = "root";
        String password = "1234";

        try {
            // Class.forName("com.mysql.cj.jdbc.Driver"); // 생략 가능 (JDBC 4.0 이상)
            Connection conn = DriverManager.getConnection(url, user, password);
            System.out.println("✅ 연결 성공!");
            conn.close();
        } catch (SQLException e) {
            System.out.println("❌ 연결 실패: " + e.getMessage());
        }
    }
}
```

✅ 6. 연결 후 해야 할 일

1. **Statement/PreparedStatement** 생성
2. **SQL 실행**
3. **ResultSet 처리**
4. **자원 해제 (`close()`)**

→ 이 부분은 이후 단계인 `SQL 실행`, `ResultSet 처리`, `트랜잭션 관리`에서 자세히 설명할게.

SQL 실행 (`Statement`, `PreparedStatement`)

Java에서 SQL을 실행하려면 `Statement` 또는 `PreparedStatement` 객체를 사용한다.
이 두 가지는 JDBC에서 **SQL 명령어를 DB로 전송하고 결과를 받아오는 핵심 도구**이다.

✅ 1. `Statement`

개요

- SQL을 문자열로 작성해서 실행
- 동적으로 쿼리를 생성할 수 있지만, **보안상 위험(SQL Injection)** 존재

예시

```
Statement stmt = conn.createStatement();
String sql = "SELECT * FROM users WHERE username = 'admin'";
ResultSet rs = stmt.executeQuery(sql);
```

주요 메서드

메서드	설명
`executeQuery(String sql)`	SELECT 문 실행, `ResultSet` 반환
`executeUpdate(String sql)`	INSERT, UPDATE, DELETE 실행, 영향받은 행 수 반환
`execute(String sql)`	모든 SQL 실행 가능. 결과가 `ResultSet`이면 `true` 반환

✅ 2. `PreparedStatement`

개요

- **미리 컴파일된 SQL 템플릿을 사용**, 파라미터 바인딩 가능
- SQL Injection 방지, 성능 개선, 가독성 향상

문법

```
String sql = "SELECT * FROM users WHERE username = ?";
PreparedStatement pstmt = conn.prepareStatement(sql);
pstmt.setString(1, "admin");
ResultSet rs = pstmt.executeQuery();
```

> `?`는 **1-based index**로 바인딩됨 (`setXXX(index, value)`)

다양한 타입 바인딩

메서드	설명
`setString(int, String)`	문자열 바인딩
`setInt(int, int)`	정수 바인딩
`setDouble(int, double)`	실수 바인딩
`setDate(int, java.sql.Date)`	날짜 바인딩
`setObject(int, Object)`	범용 타입 바인딩

✅ 3. 예제: `INSERT`, `UPDATE`, `DELETE`

```
String sql = "INSERT INTO users (username, age) VALUES (?, ?)";
PreparedStatement pstmt = conn.prepareStatement(sql);
pstmt.setString(1, "alice");
pstmt.setInt(2, 25);

int affectedRows = pstmt.executeUpdate();
System.out.println("● 삽입된 행 수: " + affectedRows);
```

✅ 4. 예제: SELECT + 결과 처리

```
String sql = "SELECT * FROM users WHERE age > ?";
PreparedStatement pstmt = conn.prepareStatement(sql);
pstmt.setInt(1, 18);

ResultSet rs = pstmt.executeQuery();
while (rs.next()) {
    int id = rs.getInt("id");
    String name = rs.getString("username");
    int age = rs.getInt("age");
    System.out.printf("ID: %d, 이름: %s, 나이: %d\n", id, name, age);
}
```

✅ 5. Statement vs PreparedStatement 요약 비교

항목	`Statement`	`PreparedStatement`
SQL 작성 방식	문자열 조합	템플릿 + 파라미터
SQL Injection 방지	❌ 위험함	✅ 안전
성능	쿼리마다 컴파일	쿼리 재사용 가능

항목	`Statement`	`PreparedStatement`
가독성	낮음	높음
추천 용도	단순 테스트	실무 전반에 사용

✅ 6. 참고: 트랜잭션 처리와 함께 사용하기

```
conn.setAutoCommit(false);

try {
    PreparedStatement pstmt1 = conn.prepareStatement("INSERT INTO orders VALUES (?, ?)");
    pstmt1.setInt(1, 1);
    pstmt1.setString(2, "book");
    pstmt1.executeUpdate();

    PreparedStatement pstmt2 = conn.prepareStatement("UPDATE stock SET qty = qty - 1 WHERE item = ?");
    pstmt2.setString(1, "book");
    pstmt2.executeUpdate();

    conn.commit();
} catch (SQLException e) {
    conn.rollback();
}
```

트랜잭션 처리

Java에서 **트랜잭션(Transaction)** 은 데이터베이스의 일련의 작업이 **모두 성공하거나 모두 실패해야 하는 원자적 단위**로 처리되는 것을 의미한다. JDBC에서 트랜잭션을 다룰 때는 `Connection` 객체를 통해 제어하며, **자동 커밋을 끄고 수동으로 commit/rollback**을 수행해야 한다.

✅ 1. 트랜잭션 기본 구조

```
Connection conn = DriverManager.getConnection(...);
conn.setAutoCommit(false); // 자동 커밋 해제

try {
    // 여러 SQL 작업 수행
    PreparedStatement pstmt1 = conn.prepareStatement(...);
    pstmt1.executeUpdate();

    PreparedStatement pstmt2 = conn.prepareStatement(...);
    pstmt2.executeUpdate();

    conn.commit(); // 모든 작업이 성공하면 커밋
} catch (SQLException e) {
    conn.rollback(); // 에러 발생 시 롤백
```

```
} finally {
    conn.close();
}
```

✅ 2. `setAutoCommit(false)` 의미

- 기본적으로 JDBC는 `setAutoCommit(true)`로 설정되어 있음 → 쿼리 실행마다 자동 커밋됨
- `false`로 바꾸면 `commit()` 호출 전까지는 실제로 DB에 반영되지 않음
- **rollback()** 호출 시, 모든 변경 사항을 원래대로 되돌림

✅ 3. 예제: 주문 처리 트랜잭션

```
String orderSQL = "INSERT INTO orders (user_id, product_id) VALUES (?, ?)";
String stockSQL = "UPDATE products SET stock = stock - 1 WHERE id = ?";

Connection conn = DriverManager.getConnection(url, user, password);
conn.setAutoCommit(false); // 트랜잭션 시작

try {
    // 주문 추가
    PreparedStatement orderStmt = conn.prepareStatement(orderSQL);
    orderStmt.setInt(1, 1);
    orderStmt.setInt(2, 101);
    orderStmt.executeUpdate();

    // 재고 감소
    PreparedStatement stockStmt = conn.prepareStatement(stockSQL);
    stockStmt.setInt(1, 101);
    stockStmt.executeUpdate();

    conn.commit(); // 성공 시 커밋
    System.out.println("✅ 주문 처리 완료");
} catch (SQLException e) {
    conn.rollback(); // 실패 시 롤백
    System.out.println("❌ 주문 실패, 롤백 처리됨");
} finally {
    conn.setAutoCommit(true); // 다시 기본값으로 복원
    conn.close();
}
```

✅ 4. 예외 발생 시 주의할 점

- `SQLException`만 잡지 말고, 넓은 범위의 예외도 고려하는 게 좋음
- `conn.rollback()`은 반드시 `catch` 블록에서 안전하게 호출할 수 있도록 설계해야 함

✅ 5. savepoint (중간 지점 설정)

여러 단계 중 일부만 롤백하고 싶을 때 사용

```
conn.setAutoCommit(false);
Savepoint savepoint1 = conn.setSavepoint();

try {
    // 첫 작업
    ...

    // 두 번째 작업
    Savepoint savepoint2 = conn.setSavepoint();

    // 오류 발생
    conn.rollback(savepoint2); // savepoint2 이후 작업만 취소

    conn.commit(); // 정상 처리된 것만 반영
} catch (SQLException e) {
    conn.rollback(); // 전체 롤백
}
```

✅ 6. 트랜잭션이 필요한 대표적인 작업들

작업 종류	트랜잭션 필요 여부
사용자 가입	✅ (회원정보 + 초기 설정 등록 등)
상품 주문	✅ (주문 + 재고 감소 + 결제 내역 등록)
게시판 글 삭제	✅ (글 + 댓글 + 첨부파일 일괄 삭제)

✅ 7. JDBC 외의 고급 트랜잭션 처리 도구

도구	설명
Spring TransactionManager	선언적 트랜잭션 지원 (`@Transactional`)
JTA (Java Transaction API)	분산 트랜잭션 처리 지원
Connection Pool (HikariCP 등)	트랜잭션과 커넥션의 효율적 관리 지원

ResultSet 처리

JDBC에서 `ResultSet` 은 `SELECT` 쿼리의 결과를 담고 있는 객체로, **DB에서 가져온 데이터 행(row)들을 탐색할 수 있도록** 해준다. 결과 집합의 각 행은 커서(cursor)로 하나씩 순회하며 접근한다.

✅ 1. ResultSet의 기본 구조

```
String sql = "SELECT id, name, email FROM users";
Connection conn = DriverManager.getConnection(...);
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(sql); // SELECT 쿼리 실행

while (rs.next()) {
    int id = rs.getInt("id");
    String name = rs.getString("name");
    String email = rs.getString("email");

    System.out.println(id + ", " + name + ", " + email);
}

rs.close();
stmt.close();
conn.close();
```

✅ 2. 커서의 개념

- `rs.next()` 호출 시, 커서가 다음 행으로 이동
- 커서 위치는 처음엔 0번째(첫 행 이전)에 있음
- `next()` 는 **boolean** 반환: 다음 행이 있으면 `true`, 없으면 `false`

✅ 3. 값 가져오기 메서드

메서드	설명
`getInt("컬럼명")`	정수 값 가져오기
`getString("컬럼명")`	문자열 가져오기
`getBoolean("컬럼명")`	불리언 가져오기
`getDouble("컬럼명")`	실수 값 가져오기
`getDate("컬럼명")`	`java.sql.Date` 객체 가져오기
`getObject("컬럼명")`	다형적인 방식으로 객체형으로 가져오기

> 컬럼 이름 대신 인덱스도 가능 (1부터 시작): `getString(2)`

✅ 4. 예외와 주의점

- `rs.getXXX("컬럼명")`에서 **컬럼명이 잘못되면** `SQLException` **발생**
- 널값 처리 시 `wasNull()` 사용:

```
int age = rs.getInt("age");
if (rs.wasNull()) {
    age = -1; // 또는 null 처리
}
```

✅ 5. ResultSet 타입 종류 (옵션 설정)

기본적으로는 단방향으로 한 번만 순회 가능하지만, 다음과 같은 설정도 가능하다:

```
Statement stmt = conn.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_READ_ONLY
);
```

옵션	설명
`TYPE_FORWARD_ONLY`	(기본) 순방향만 가능
`TYPE_SCROLL_INSENSITIVE`	커서 자유 이동 가능, DB 변경 반영 안 됨
`TYPE_SCROLL_SENSITIVE`	커서 자유 이동 가능, DB 변경 반영됨
`CONCUR_READ_ONLY`	읽기 전용
`CONCUR_UPDATABLE`	수정 가능 ResultSet

✅ 6. 커서 이동 메서드 (Scroll 가능한 경우)

```
rs.absolute(5); // 5번째 행으로 이동
rs.beforeFirst(); // 처음으로 이동
rs.afterLast();   // 마지막 다음으로 이동
rs.previous();    // 이전 행으로 이동
rs.first();       // 첫 행으로 이동
rs.last();        // 마지막 행으로 이동
```

✅ 7. ResultSet → 객체 매핑 예시 (DTO 매핑)

```
class User {
    int id;
    String name;
    String email;
```

```
    // 생성자, getter, setter 등
}

List<User> users = new ArrayList<>();

while (rs.next()) {
    User user = new User();
    user.setId(rs.getInt("id"));
    user.setName(rs.getString("name"));
    user.setEmail(rs.getString("email"));
    users.add(user);
}
```

✅ 8. 마무리: 리소스 해제 필수

```
rs.close();
stmt.close();
conn.close();
```

자원을 닫지 않으면 커넥션 누수 발생 → 커넥션 풀 사용 시 심각한 장애 유발 가능

✅ 참고: ResultSetMetaData

결과의 컬럼 개수나 컬럼명 등을 동적으로 확인할 때 사용

```
ResultSetMetaData meta = rs.getMetaData();
int columnCount = meta.getColumnCount();

for (int i = 1; i <= columnCount; i++) {
    System.out.println(meta.getColumnName(i));
}
```

DB 커넥션 풀 (HikariCP 등)

✅ DB 커넥션 풀(Connection Pool)이란?

📌 개념

DB 커넥션 풀은 **DB 연결 객체(Connection)** 를 미리 생성해두고,
필요할 때마다 꺼내 쓰고 다시 반납하는 **풀(pool) 구조**를 말한다.

```
기존 방식: 매 요청마다 DB 연결 → 성능 저하
풀 방식: 미리 만들어둔 커넥션 재사용 → 고성능
```

📌 장점

- DB 연결/해제 오버헤드 제거
- 트래픽 급증 대응력 향상
- DB 연결 수 제한 설정 가능
- 안정적인 커넥션 관리

✅ 주요 커넥션 풀 라이브러리

이름	특징
HikariCP	매우 빠르고 가볍고, Spring Boot 기본 채택
Apache DBCP2	안정적이지만 성능은 HikariCP보다 낮음
C3P0	설정이 다양하지만 오래됨

> 최근 프로젝트는 거의 대부분 **HikariCP** 사용

✅ HikariCP 사용법

1. Maven 의존성 추가

```
<dependency>
    <groupId>com.zaxxer</groupId>
    <artifactId>HikariCP</artifactId>
    <version>5.1.0</version>
</dependency>
```

> Gradle:

```
implementation 'com.zaxxer:HikariCP:5.1.0'
```

---

2. 기본 사용 예시 (순수 Java)

```
import com.zaxxer.hikari.HikariConfig;
import com.zaxxer.hikari.HikariDataSource;

import java.sql.Connection;

public class HikariExample {
    public static void main(String[] args) throws Exception {
        HikariConfig config = new HikariConfig();
        config.setJdbcUrl("jdbc:mysql://localhost:3306/testdb");
        config.setUsername("root");
        config.setPassword("password");
        config.setMaximumPoolSize(10);
```

```
13
14          HikariDataSource ds = new HikariDataSource(config);
15
16          try (Connection conn = ds.getConnection()) {
17              System.out.println("DB 연결 성공");
18          }
19
20          ds.close(); // 반드시 해제
21      }
22  }
```

---

3. 주요 설정 항목

설정	설명	기본값
`jdbcUrl`	DB 접속 URL	없음
`username`	DB 사용자명	없음
`password`	비밀번호	없음
`maximumPoolSize`	최대 커넥션 수	10
`minimumIdle`	최소 유휴 커넥션 수	same as max
`idleTimeout`	유휴 커넥션 유지 시간 (ms)	600000 (10분)
`connectionTimeout`	커넥션 얻기 대기 시간 (ms)	30000 (30초)
`validationTimeout`	커넥션 유효성 검사 시간	5000
`connectionTestQuery`	유효성 검사용 쿼리 (보통 `SELECT 1`)	DB별 자동 추론

✅ Spring Boot에서 HikariCP 사용하기

Spring Boot 2.x 이상은 기본으로 HikariCP가 포함됨
→ 별도 설정 없이 `spring.datasource.*` 로 자동 구성됨

`application.yml` 예시

```yaml
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/testdb
    username: root
    password: password
    driver-class-name: com.mysql.cj.jdbc.Driver
    hikari:
      maximum-pool-size: 10
      minimum-idle: 5
      idle-timeout: 600000
      connection-timeout: 30000
```

✅ 커넥션 풀 테스트 코드

```java
@Autowired
DataSource dataSource;

@Test
void 커넥션풀_확인() throws Exception {
    Connection conn = dataSource.getConnection();
    System.out.println("커넥션 클래스: " + conn.getClass());
    conn.close();
}
```

출력 예시:

```
커넥션 클래스: class com.zaxxer.hikari.pool.HikariProxyConnection
```

🧠 성능 측면에서 주의할 점

- `maximumPoolSize`를 너무 작게 잡으면 대기 시간 증가
- 너무 크게 잡으면 DB 연결 과부하
- 적절한 설정은 **TPS, DB 동시 처리 능력, CPU/메모리 여유** 고려 필요

📦 요약

항목	설명
HikariCP	가장 빠르고 경량의 커넥션 풀
사용 방식	`HikariDataSource` 직접 생성 또는 Spring Boot 자동 설정
설정 중요값	`maximumPoolSize`, `idleTimeout`, `connectionTimeout` 등

항목	설명
성능 향상	DB 연결 비용 절감, 처리량 증가
사용 환경	Spring Boot, JDBC, JPA 등 모든 Java DB 연동 시스템