

1. JDBC 이해

#1.인강/7.스프링 DB 1/강의#

- /프로젝트 생성
- /H2 데이터베이스 설정
- /JDBC 이해
- /JDBC와 최신 데이터 접근 기술
- /데이터베이스 연결
- /JDBC 개발 - 등록
- /JDBC 개발 - 조회
- /JDBC 개발 - 수정, 삭제
- /정리

프로젝트 생성

사전 준비물

- Java 17 이상 설치
- IDE: IntelliJ 또는 Eclipse 설치

스프링 부트 스타터 사이트로 이동해서 스프링 프로젝트 생성

<https://start.spring.io>

- 프로젝트 선택
 - Project: **Gradle - Groovy** Project
 - Language: Java
 - Spring Boot: 3.x.x
- Project Metadata
 - Group: hello
 - Artifact: jdbc
 - Name: jdbc
 - Package name: **hello.jdbc**
 - Packaging: **Jar**
 - Java: 17 또는 21
- Dependencies: **JDBC API, H2 Database, Lombok**

주의! - 스프링 부트 3.x 버전 선택 필수

start.spring.io 사이트에서 스프링 부트 2.x에 대한 지원이 종료되어서 더는 선택할 수 없습니다.

이제는 스프링 부트 3.0 이상을 선택해주세요.

스프링 부트 3.0을 선택하게 되면 다음 부분을 꼭 확인해주세요.

- **1. Java 17 이상**을 사용해야 합니다.
- **2. javax 패키지 이름을 jakarta로 변경**해야 합니다.
 - 오라클과 자바 라이선스 문제로 모든 javax 패키지를 jakarta로 변경하기로 했습니다.
- **3. H2 데이터베이스를 2.1.214 버전 이상** 사용해주세요.

패키지 이름 변경 예)

- **JPA 애노테이션**
 - javax.persistence.Entity → jakarta.persistence.Entity
- **스프링에서 자주 사용하는 @PostConstruct 애노테이션**
 - javax.annotation.PostConstruct → jakarta.annotation.PostConstruct
- **스프링에서 자주 사용하는 검증 애노테이션**
 - javax.validation → jakarta.validation

스프링 부트 3.0 관련 자세한 내용은 다음 링크를 확인해주세요: <https://bit.ly/springboot3>

build.gradle

```
plugins {  
    id 'org.springframework.boot' version '2.6.4'  
    id 'io.spring.dependency-management' version '1.0.11.RELEASE'  
    id 'java'  
}  
  
group = 'hello'  
version = '0.0.1-SNAPSHOT'  
sourceCompatibility = '11'  
  
configurations {  
    compileOnly {  
        extendsFrom annotationProcessor  
    }  
}
```

```

}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-jdbc'
    compileOnly 'org.projectlombok:lombok'
    runtimeOnly 'com.h2database:h2'
    annotationProcessor 'org.projectlombok:lombok'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'

    //테스트에서 lombok 사용
    testCompileOnly 'org.projectlombok:lombok'
    testAnnotationProcessor 'org.projectlombok:lombok'
}

tasks.named('test') {
    useJUnitPlatform()
}

```

테스트에서도 **lombok**을 사용하기 위해 다음 코드를 추가하자.

(위 코드에는 추가해두었다.)

```

build.gradle
dependencies {
    ...
    //테스트에서 lombok 사용
    testCompileOnly 'org.projectlombok:lombok'
    testAnnotationProcessor 'org.projectlombok:lombok'
}

```

이 설정을 추가해야 테스트 코드에서 `@Slf4j` 같은 롬복 애노테이션을 사용할 수 있다.

- 동작 확인
 - 기본 메인 클래스 실행(`JdbcApplication.main()`)
 - 콘솔에 `Started JdbcApplication` 로그가 보이면 성공이다.

H2 데이터베이스 설정

H2 데이터베이스는 개발이나 테스트 용도로 사용하기 좋은 가볍고 편리한 DB이다. 그리고 SQL을 실행할 수 있는 웹 화면을 제공한다.

- <https://www.h2database.com>
- 다운로드 및 설치
- **h2 데이터베이스 버전은 스프링 부트 버전에 맞춘다.**
 - 스프링 부트 2.x를 사용하면 **1.4.200 버전**을 다운로드 받으면 된다.
 - 스프링 부트 3.x를 사용하면 **2.1.214 버전 이상** 사용해야 한다.
 - 다음 링크에 가면 다양한 H2 다운로드 버전을 확인할 수 있다.
 - <https://www.h2database.com/html/download-archive.html>

MAC, 리눅스 사용자

- 디렉토리 이동 : `cd bin`
- 권한 주기: `chmod 755 h2.sh`
- 실행: `./h2.sh`

윈도우 사용자


- 실행: `h2.bat`
- 데이터베이스 파일 생성 방법
 - 사용자명은 `sa` 입력
 - JDBC URL에 다음 입력,
 - `jdbc:h2:~/test` (최초 한번) → 이 경우 `연결 시험`을 호출하면 오류가 발생한다. `연결`을 직접 눌러주어야 한다.
 - `~/test.mv.db` 파일 생성 확인
 - 이후부터는 `jdbc:h2:tcp://localhost/~/test` 이렇게 접속

참고: H2 데이터베이스가 정상 생성되지 않을 때

JDBC URL 에 `jdbc:h2:~/test` 를 입력해도 다음과 같은 오류 메시지가 나오며 H2 데이터베이스가 생성되지 않는 경우가 있다.


해결방안은 다음과 같다.

1. H2 데이터베이스를 종료하고, 다시 시작한다.
2. 웹 브라우저가 자동 실행되면 주소창에 다음과 같이 되어있다.(100.1.2.3이 아니라 임의의 숫자가 나온다.)

 100.1.2.3:8082/login.jsp?jsessionId=d1e9d744ce01f475cb1c0bce286be109

3. 다음과 같이 앞 부분만 100.1.2.3 → localhost 로 변경하고 Enter를 입력한다. 나머지 부분은 절대 변경하면 안된다.

주의! 특히 뒤에 jsessionId 부분이 변경되면 안된다.

 localhost:8082/login.jsp?jsessionId=d1e9d744ce01f475cb1c0bce286be109

4. 이제 JDBC URL에 jdbc:h2:~/test 를 입력하면, 데이터베이스가 정상 생성된다.
5. 이후에는 jdbc:h2:tcp://localhost/~/test 로 접속하자.

참고

만약 그래도 데이터베이스에 접근할 수 없다면 다음 URL에 접근한 다음 H2 데이터베이스 접속 오류 부분을 확인해보자.

<https://bit.ly/3fX6ygg>

테이블 생성하기

테이블 관리를 위해 프로젝트 루트에 sql/schema.sql 파일을 생성하자

```
drop table member if exists cascade;
create table member (
    member_id varchar(10),
    money integer not null default 0,
    primary key (member_id)
);
```

```
insert into member(member_id, money) values ('hi1',10000);
insert into member(member_id, money) values ('hi2',20000);
```

- H2 데이터베이스 웹 콘솔에 접근하자.
 - H2 데이터베이스 웹 콘솔 접근 URL: <http://localhost:8082/>
- 방금 작성한 SQL을 H2 데이터베이스 웹 콘솔에서 실행해서 `member` 테이블 생성한다.

다음 쿼리를 실행해서 저장한 데이터가 잘 나오는지 결과를 확인하자.

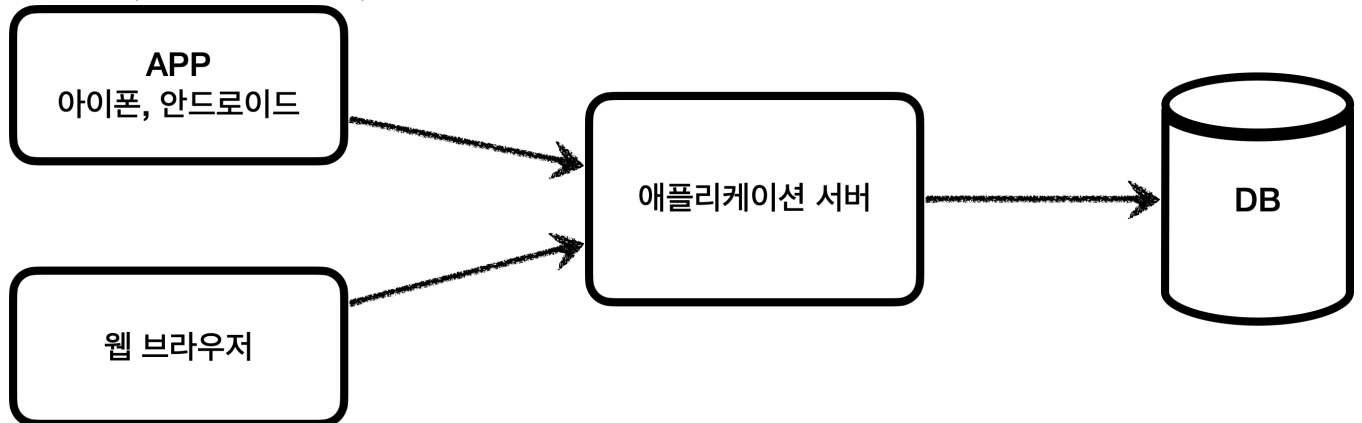
```
select * from member;
```

JDBC 이해

JDBC 등장 이유

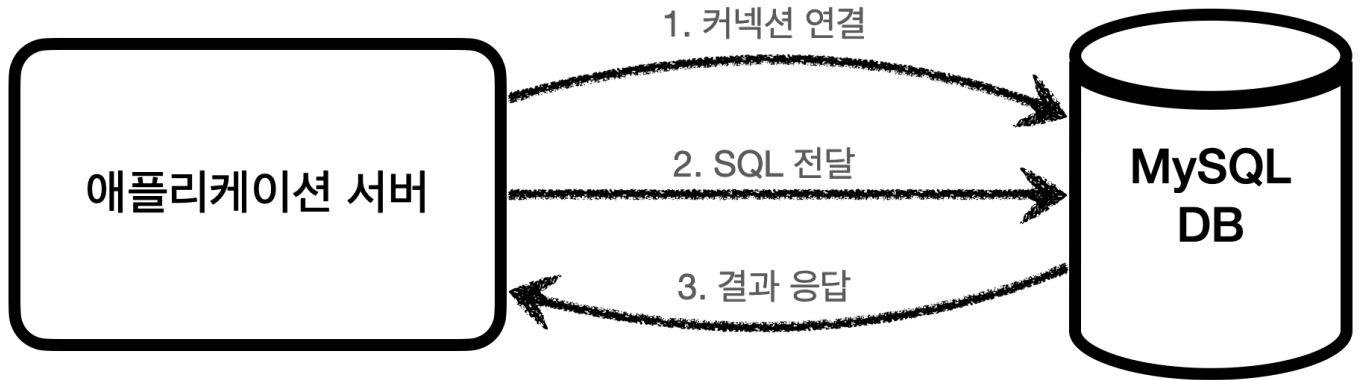
애플리케이션을 개발할 때 중요한 데이터는 대부분 데이터베이스에 보관한다.

클라이언트, 애플리케이션 서버, DB



클라이언트가 애플리케이션 서버를 통해 데이터를 저장하거나 조회하면, 애플리케이션 서버는 다음 과정을 통해서 데이터베이스를 사용한다.

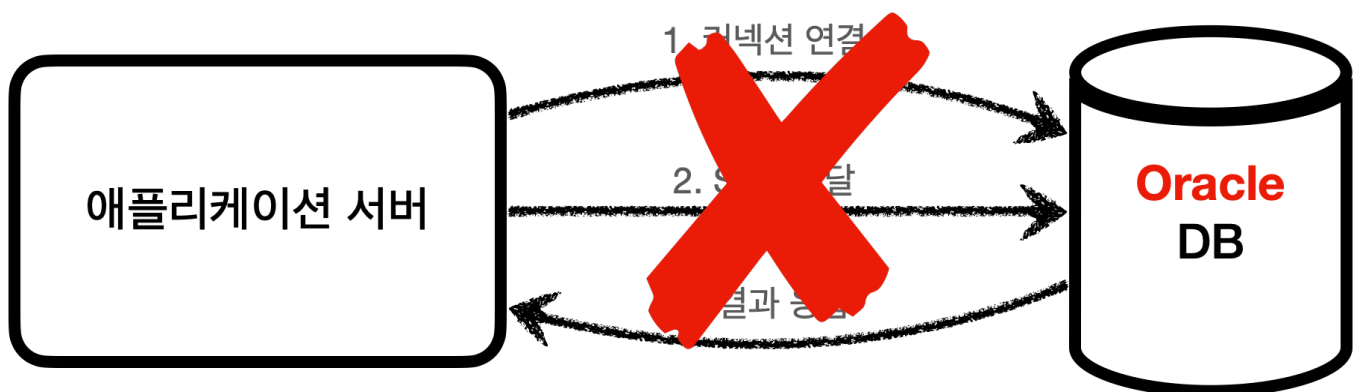
애플리케이션 서버와 DB - 일반적인 사용법



1. 커넥션 연결: 주로 TCP/IP를 사용해서 커넥션을 연결한다.
2. SQL 전달: 애플리케이션 서버는 DB가 이해할 수 있는 SQL을 연결된 커넥션을 통해 DB에 전달한다.
3. 결과 응답: DB는 전달된 SQL을 수행하고 그 결과를 응답한다. 애플리케이션 서버는 응답 결과를 활용한다.

애플리케이션 서버와 DB - DB 변경

각각의 데이터베이스마다 사용법이 다르다.



문제는 각각의 데이터베이스마다 커넥션을 연결하는 방법, SQL을 전달하는 방법, 그리고 결과를 응답 받는 방법이 모두 다르다는 점이다.

참고로 관계형 데이터베이스는 수십개가 있다.

여기에는 2가지 큰 문제가 있다.

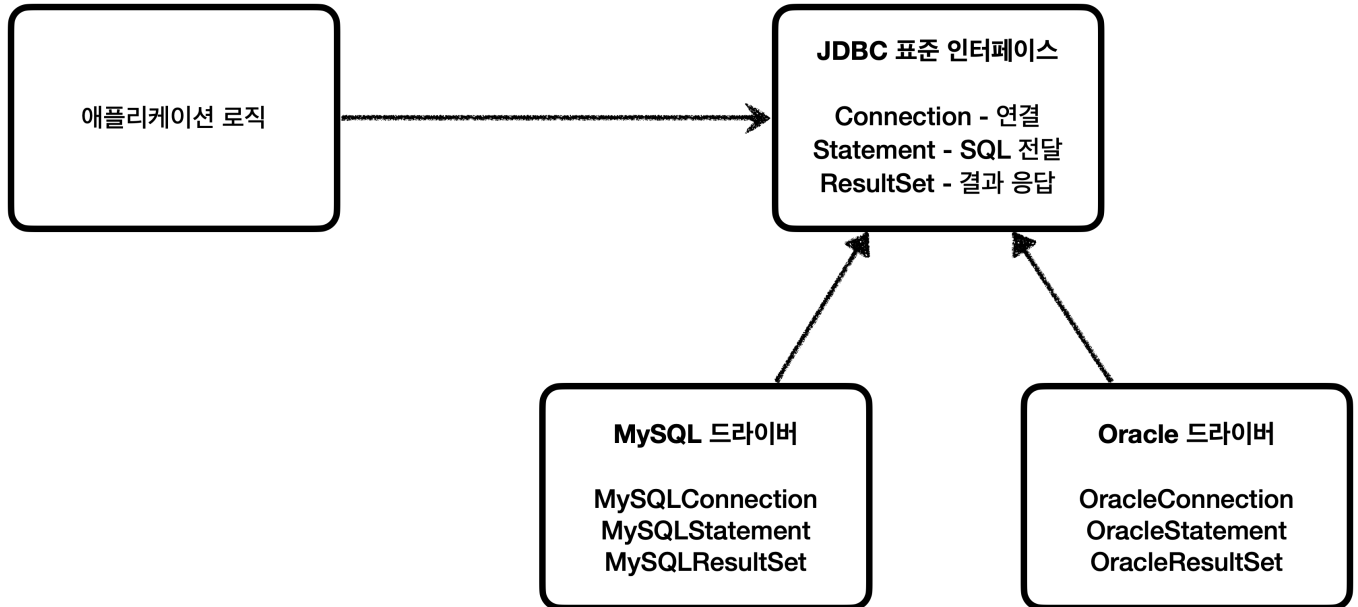
1. 데이터베이스를 다른 종류의 데이터베이스로 변경하면 애플리케이션 서버에 개발된 데이터베이스 사용 코드도 함께 변경해야 한다.
2. 개발자가 각각의 데이터베이스마다 커넥션 연결, SQL 전달, 그리고 그 결과를 응답 받는 방법을 새로 학습해야 한다.

이런 문제를 해결하기 위해 JDBC라는 자바 표준이 등장한다.

JDBC 표준 인터페이스

JDBC(Java Database Connectivity)는 자바에서 데이터베이스에 접속할 수 있도록 하는 자바 API다. JDBC는 데이터베이스에서 자료를 쿼리하거나 업데이트하는 방법을 제공한다. - 위키백과

JDBC 표준 인터페이스

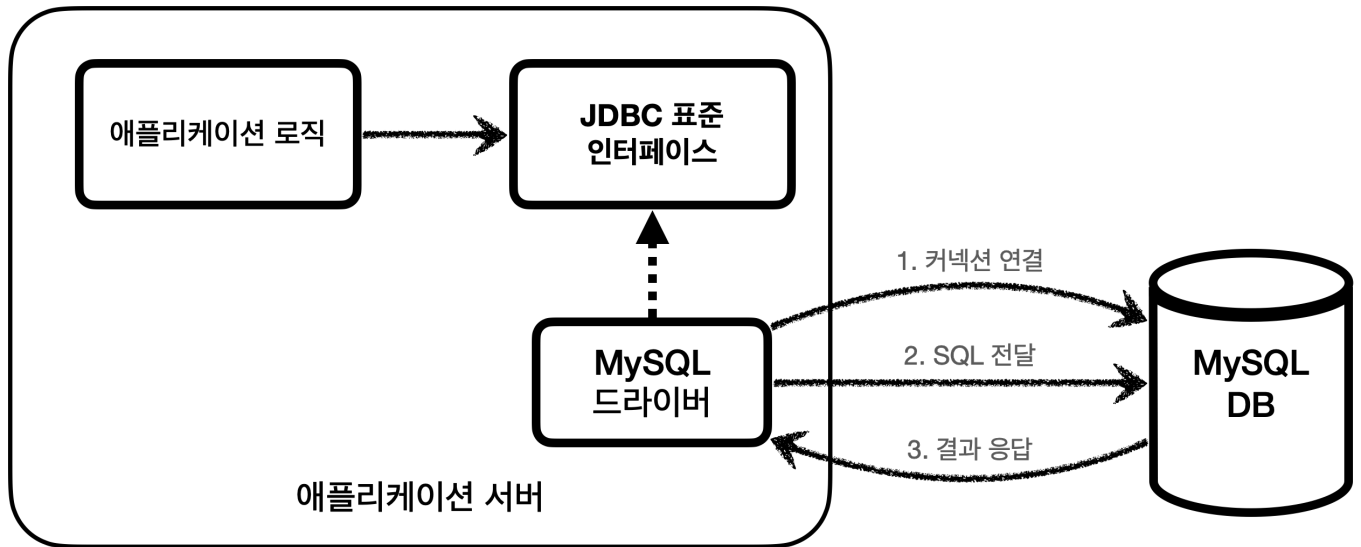


대표적으로 다음 3가지 기능을 표준 인터페이스로 정의해서 제공한다.

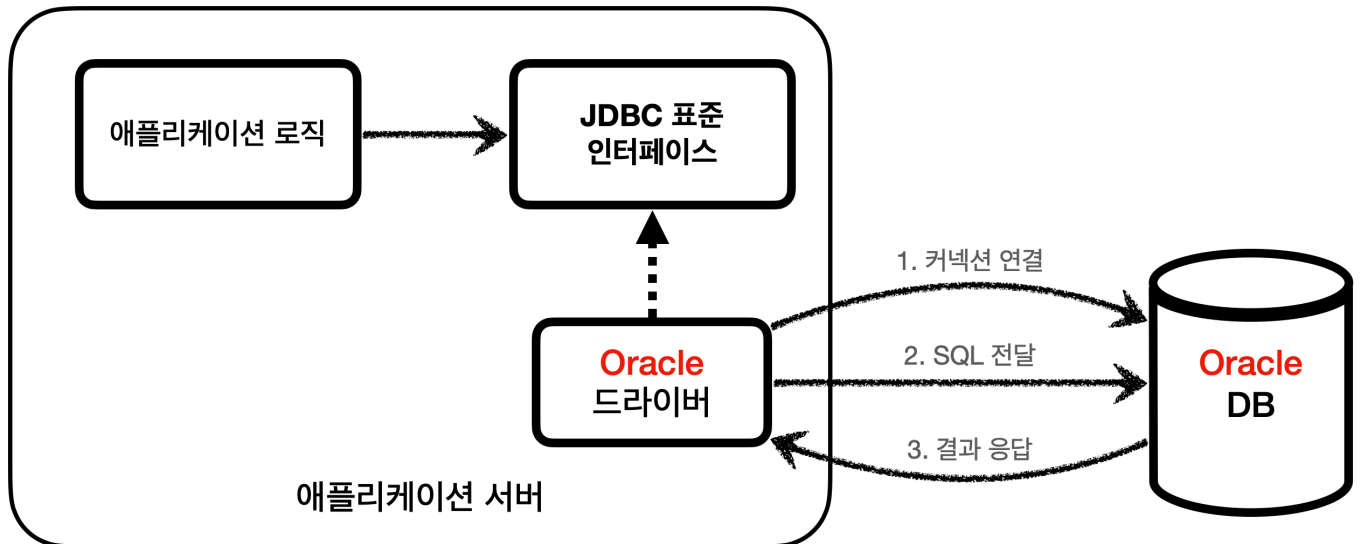
- `java.sql.Connection` - 연결
- `java.sql.Statement` - SQL을 담은 내용
- `java.sql.ResultSet` - SQL 요청 응답

자바는 이렇게 표준 인터페이스를 정의해두었다. 이제부터 개발자는 이 표준 인터페이스만 사용해서 개발하면 된다. 그런데 인터페이스만 있다고해서 기능이 동작하지는 않는다. 이 JDBC 인터페이스를 각각의 DB 벤더(회사)에서 자신의 DB에 맞도록 구현해서 라이브러리로 제공하는데, 이것을 JDBC 드라이버라 한다. 예를 들어서 MySQL DB에 접근할 수 있는 것은 MySQL JDBC 드라이버라 하고, Oracle DB에 접근할 수 있는 것은 Oracle JDBC 드라이버라 한다.

MySQL 드라이버 사용



Oracle 드라이버 사용



정리

JDBC의 등장으로 다음 2가지 문제가 해결되었다.

1. 데이터베이스를 다른 종류의 데이터베이스로 변경하면 애플리케이션 서버의 데이터베이스 사용 코드도 함께 변경해야 하는 문제
 - 애플리케이션 로직은 이제 JDBC 표준 인터페이스에만 의존한다. 따라서 데이터베이스를 다른 종류의 데이터베이스로 변경하고 싶으면 JDBC 구현 라이브러리만 변경하면 된다. 따라서 다른 종류의 데이터베이스로 변경해도 애플리케이션 서버의 사용 코드를 그대로 유지할 수 있다.
2. 개발자가 각각의 데이터베이스마다 커넥션 연결, SQL 전달, 그리고 그 결과를 응답 받는 방법을 새로 학습해야 하는 문제
 - 개발자는 JDBC 표준 인터페이스 사용법만 학습하면 된다. 한번 배워두면 수십개의 데이터베이스에 모두 동일하게 적용할 수 있다.

참고 - 표준화의 한계

JDBC의 등장으로 많은 것이 편리해졌지만, 각각의 데이터베이스마다 SQL, 데이터타입 등의 일부 사용법 다르다. ANSI SQL이라는 표준이 있기는 하지만 일반적인 부분만 공통화했기 때문에 한계가 있다. 대표적으로 실무에서 기본으로 사용하는 페이징 SQL은 각각의 데이터베이스마다 사용법이 다르다.

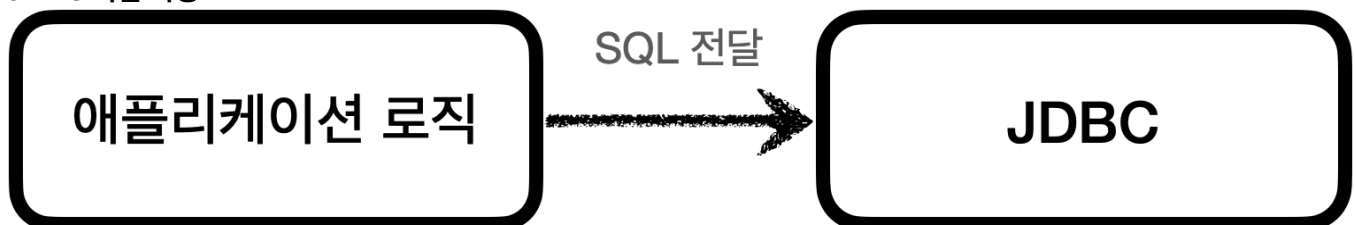
결국 데이터베이스를 변경하면 JDBC 코드는 변경하지 않아도 되지만 SQL은 해당 데이터베이스에 맞도록 변경해야 한다.

참고로 JPA(Java Persistence API)를 사용하면 이렇게 각각의 데이터베이스마다 다른 SQL을 정의해야 하는 문제도 많은 부분 해결할 수 있다.

JDBC와 최신 데이터 접근 기술

JDBC는 1997년에 출시될 정도로 오래된 기술이고, 사용하는 방법도 복잡하다. 그래서 최근에는 JDBC를 직접 사용하기 보다는 JDBC를 편리하게 사용하는 다양한 기술이 존재한다. 대표적으로 SQL Mapper와 ORM 기술로 나눌 수 있다.

JDBC 직접 사용



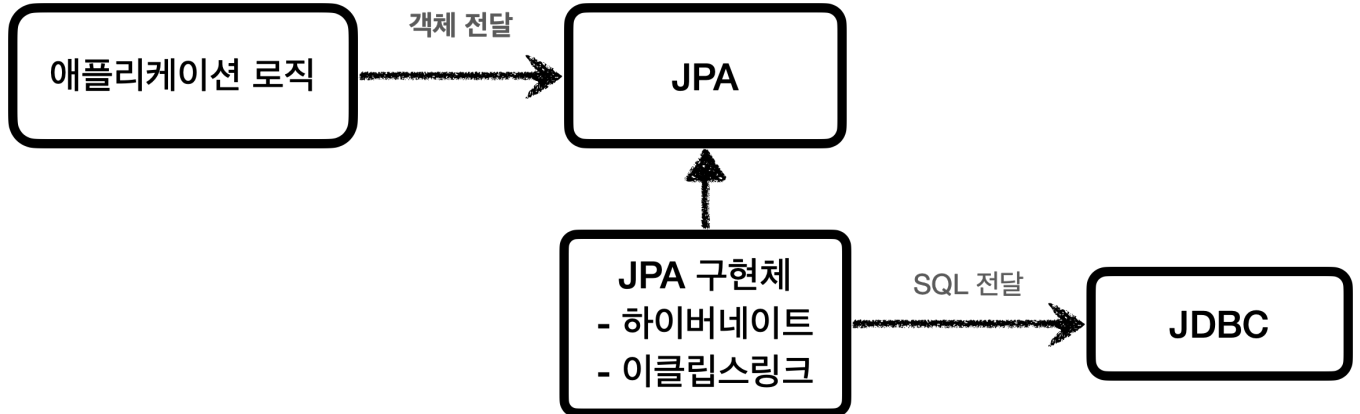
SQL Mapper



- SQL Mapper
 - 장점: JDBC를 편리하게 사용하도록 도와준다.
 - ◆ SQL 응답 결과를 객체로 편리하게 변환해준다.
 - ◆ JDBC의 반복 코드를 제거해준다.

- 단점: 개발자가 SQL을 직접 작성해야한다.
- 대표 기술: 스프링 JdbcTemplate, MyBatis

ORM 기술



- ORM 기술
 - ORM은 객체를 관계형 데이터베이스 테이블과 매핑해주는 기술이다. 이 기술 덕분에 개발자는 반복적인 SQL을 직접 작성하지 않고, ORM 기술이 개발자 대신에 SQL을 동적으로 만들어 실행해준다. 추가로 각각의 데이터베이스마다 다른 SQL을 사용하는 문제도 중간에서 해결해준다.
 - 대표 기술: JPA, 하이버네이트, 이클립스링크
 - JPA는 자바 진영의 ORM 표준 인터페이스이고, 이것을 구현한 것으로 하이버네이트와 이클립스 링크 등의 구현 기술이 있다.

SQL Mapper vs ORM 기술

SQL Mapper와 ORM 기술 둘다 각각 장단점이 있다.

쉽게 설명하자면 SQL Mapper는 SQL만 직접 작성하면 나머지 번거로운 일은 SQL Mapper가 대신 해결해준다.

SQL Mapper는 SQL만 작성할 줄 알면 금방 배워서 사용할 수 있다.

ORM기술은 SQL 자체를 작성하지 않아도 되어서 개발 생산성이 매우 높아진다. 편리한 반면에 쉬운 기술은 아니므로 실무에서 사용하려면 깊이있게 학습해야 한다.

강의 뒷 부분에서 다양한 데이터 접근 기술을 설명하는데, 그때 SQL Mapper인 JdbcTemplate과 MyBatis를 학습하고 코드로

활용해본다. 그리고 ORM의 대표 기술인 JPA도 학습하고 코드로 활용해본다. 이 과정을 통해서 각각의 기술들의 장단점을 파악하고, 어떤 기술을 언제 사용해야 하는지 자연스럽게 이해하게 될 것이다.

중요

이런 기술들도 내부에서는 모두 JDBC를 사용한다. 따라서 JDBC를 직접 사용하지는 않더라도, JDBC가 어떻게 동작하는지 기본 원리를 알아두어야 한다. 그래야 해당 기술들을 더 깊이있게 이해할 수 있고, 무엇보다 문제가 발생했을 때 근본적인 문제를 찾아서 해결할 수 있다 **JDBC는 자바 개발자라면 꼭 알아두어야 하는 필수 기본 기술**이다.

데이터베이스 연결

애플리케이션과 데이터베이스를 연결해보자.

주의

H2 데이터베이스 서버를 먼저 실행해두자.

ConnectionConst

```
package hello.jdbc.connection;

public abstract class ConnectionConst {
    public static final String URL = "jdbc:h2:tcp://localhost/~/test";
    public static final String USERNAME = "sa";
    public static final String PASSWORD = "";
}
```

데이터베이스에 접속하는데 필요한 기본 정보를 편리하게 사용할 수 있도록 상수로 만들었다.

이제 JDBC를 사용해서 실제 데이터베이스에 연결하는 코드를 작성해보자.

DBConnectionUtil

```
package hello.jdbc.connection;

import lombok.extern.slf4j.Slf4j;

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

import static hello.jdbc.connection.ConnectionConst.*;

@Slf4j
public class DBConnectionUtil {

    public static Connection getConnection() {
```

```

    try {
        Connection connection = DriverManager.getConnection(URL, USERNAME,
PASSWORD);
        log.info("get connection={}, class={}", connection,
connection.getClass());
        return connection;
    } catch (SQLException e) {
        throw new IllegalStateException(e);
    }
}
}

```

데이터베이스에 연결하려면 JDBC가 제공하는 `DriverManager.getConnection(..)` 를 사용하면 된다. 이렇게 하면 라이브러리에 있는 데이터베이스 드라이버를 찾아서 해당 드라이버가 제공하는 커넥션을 반환해준다. 여기서는 H2 데이터베이스 드라이버가 작동해서 실제 데이터베이스와 커넥션을 맺고 그 결과를 반환해준다.

간단한 학습용 테스트 코드를 만들어서 실행해보자.

DBConnectionUtilTest

```

package hello.jdbc.connection;

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

import java.sql.Connection;

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

@Slf4j
class DBConnectionUtilTest {

    @Test
    void connection() {
        Connection connection = DBConnectionUtil.getConnection();
        assertThat(connection).isNotNull();
    }
}

```

주의!

실행전에 H2 데이터베이스 서버를 실행해두어야 한다. (`h2.sh`, `h2.bat`)

실행 결과

```
DBConnectionUtil - get connection=conn0: url=jdbc:h2:tcp://localhost/~/test
user=SA, class=class org.h2.jdbc.JdbcConnection
```

실행 결과를 보면 `class=class org.h2.jdbc.JdbcConnection` 부분을 확인할 수 있다. 이것이 바로 H2 데이터베이스 드라이버가 제공하는 H2 전용 커넥션이다. 물론 이 커넥션은 JDBC 표준 커넥션 인터페이스인 `java.sql.Connection` 인터페이스를 구현하고 있다.

참고 - 오류

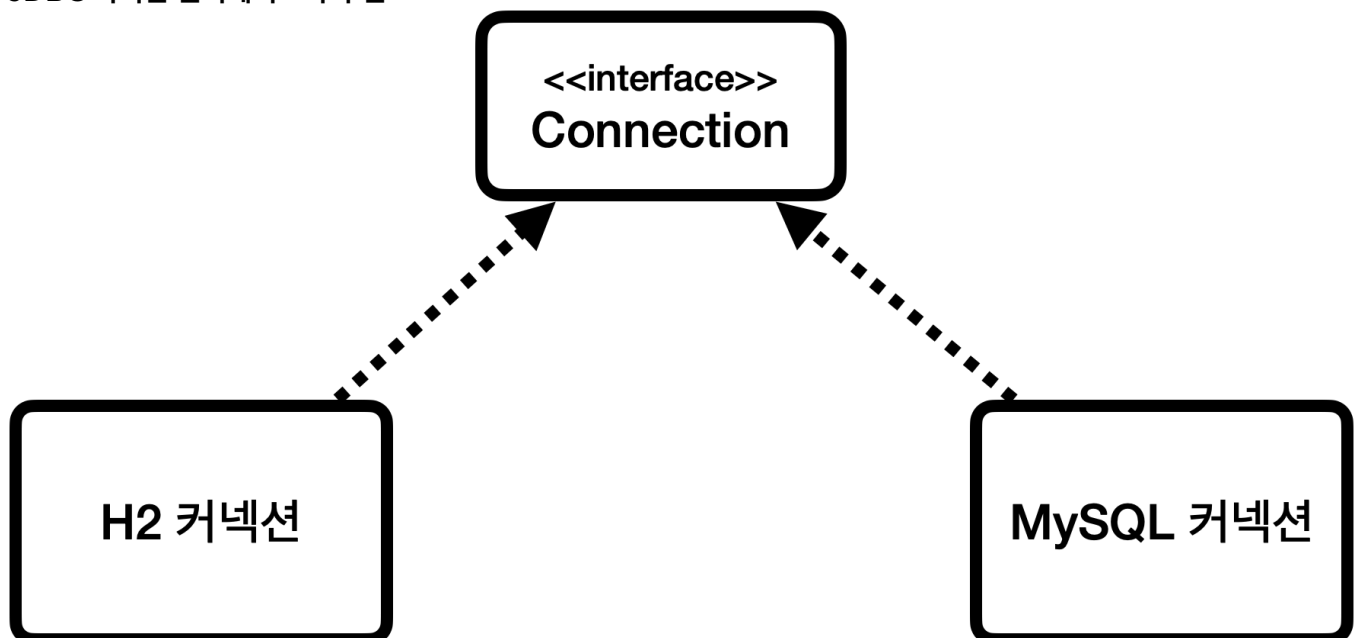
다음과 같은 오류가 발생하면 H2 데이터베이스가 실행되지 않았거나, 설정에 오류가 있는 것이다. H2 데이터베이스 설정 부분을 다시 확인하자.

```
Connection is broken: "java.net.ConnectException: Connection refused (Connection
refused): localhost" [90067-200]
```

JDBC DriverManager 연결 이해

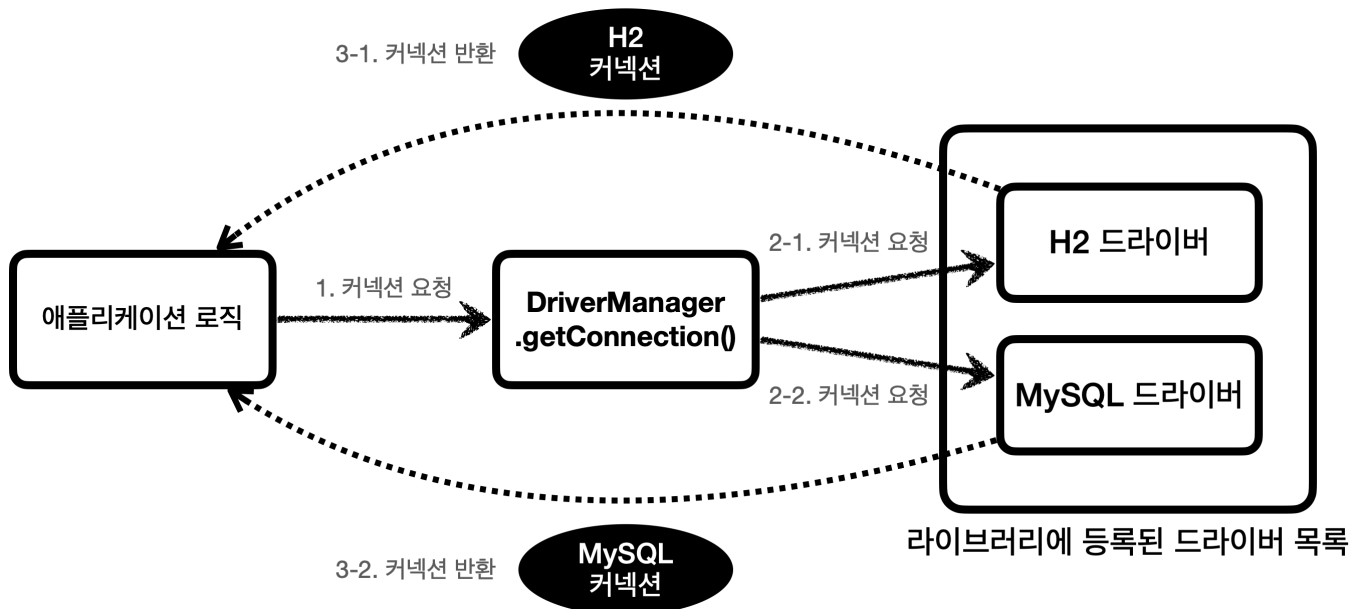
지금까지 코드로 확인해본 과정을 좀 더 자세히 알아보자.

JDBC 커넥션 인터페이스와 구현



- JDBC는 `java.sql.Connection` 표준 커넥션 인터페이스를 정의한다.
- H2 데이터베이스 드라이버는 JDBC Connection 인터페이스를 구현한 `org.h2.jdbc.JdbcConnection` 구현체를 제공한다.

DriverManager 커넥션 요청 흐름



JDBC가 제공하는 `DriverManager`는 라이브러리에 등록된 DB 드라이버들을 관리하고, 커넥션을 획득하는 기능을 제공한다.

1. 애플리케이션 로직에서 커넥션이 필요하면 `DriverManager.getConnection()`을 호출한다.
2. `DriverManager`는 라이브러리에 등록된 드라이버 목록을 자동으로 인식한다. 이 드라이버들에게 순서대로 다음 정보를 넘겨서 커넥션을 획득할 수 있는지 확인한다.
 - URL: 예) `jdbc:h2:tcp://localhost/~/test`
 - 이름, 비밀번호 등 접속에 필요한 추가 정보
 - 여기서 각각의 드라이버는 URL 정보를 체크해서 본인이 처리할 수 있는 요청인지 확인한다. 예를 들어서 URL이 `jdbc:h2`로 시작하면 이것은 h2 데이터베이스에 접근하기 위한 규칙이다. 따라서 H2 드라이버는 본인이 처리할 수 있으므로 실제 데이터베이스에 연결해서 커넥션을 획득하고 이 커넥션을 클라이언트에 반환한다. 반면에 URL이 `jdbc:h2`로 시작했는데 MySQL 드라이버가 먼저 실행되면 이 경우 본인이 처리할 수 없다는 결과를 반환하게 되고, 다음 드라이버에게 순서가 넘어간다.
3. 이렇게 찾은 커넥션 구현체가 클라이언트에 반환된다.

우리는 H2 데이터베이스 드라이버만 라이브러리에 등록했기 때문에 H2 드라이버가 제공하는 H2 커넥션을 제공받는다. 물론 이 H2 커넥션은 JDBC가 제공하는 `java.sql.Connection` 인터페이스를 구현하고 있다.

H2 데이터베이스 드라이버 라이브러리

```
runtimeOnly 'com.h2database:h2' //h2-x.x.xxx.jar
```

JDBC 개발 - 등록

이제 본격적으로 JDBC를 사용해서 애플리케이션을 개발해보자.

여기서는 JDBC를 사용해서 회원(Member) 데이터를 데이터베이스에 관리하는 기능을 개발해보자.

주의!

H2 데이터베이스 설정 마지막에 있는 테이블과 샘플 데이터 만들기를 통해서 member 테이블을 미리 만들어두어야 한다.

schema.sql

```
drop table member if exists cascade;
create table member (
    member_id varchar(10),
    money integer not null default 0,
    primary key (member_id)
);
```

Member

```
package hello.jdbc.domain;

import lombok.Data;

@Data
public class Member {

    private String memberId;
    private int money;

    public Member() {
    }

    public Member(String memberId, int money) {
        this.memberId = memberId;
        this.money = money;
    }
}
```

회원의 ID와 해당 회원이 소지한 금액을 표현하는 단순한 클래스이다. 앞서 만들어진 member 테이블에 데이터를 저

장하고 조회할 때 사용한다.

가장 먼저 JDBC를 사용해서 이렇게 만든 회원 객체를 데이터베이스에 저장해보자.

MemberRepositoryV0 - 회원 등록

```
package hello.jdbc.repository;

import hello.jdbc.connection.DBConnectionUtil;
import hello.jdbc.domain.Member;
import lombok.extern.slf4j.Slf4j;

import java.sql.*;

/**
 * JDBC - DriverManager 사용
 */
@Slf4j
public class MemberRepositoryV0 {

    public Member save(Member member) throws SQLException {
        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) {
            log.error("db error", e);
            throw e;
        } finally {
            close(con, pstmt, null);
        }
    }

    private void close(Connection con, Statement stmt, ResultSet rs) {
        if (rs != null) {

```

```

        try {
            rs.close();
        } catch (SQLException e) {
            log.info("error", e);
        }
    }

    if (stmt != null) {
        try {
            stmt.close();
        } catch (SQLException e) {
            log.info("error", e);
        }
    }

    if (con != null) {
        try {
            con.close();
        } catch (SQLException e) {
            log.info("error", e);
        }
    }
}

private Connection getConnection() {
    return DBConnectionUtil.getConnection();
}
}

```

커넥션 획득

- `getConnection()`: 이전에 만들어둔 `DBConnectionUtil`를 통해서 데이터베이스 커넥션을 획득한다.

save() - SQL 전달

- `sql`: 데이터베이스에 전달할 SQL을 정의한다. 여기서는 데이터를 등록해야 하므로 `insert sql`을 준비했다.
- `con.prepareStatement(sql)`: 데이터베이스에 전달할 SQL과 파라미터로 전달할 데이터들을 준비한다.
 - `sql: insert into member(member_id, money) values(?, ?)"`
 - `pstmt.setString(1, member.getMemberId())`: SQL의 첫번째 ?에 값을 지정한다. 문자이므로 `setString`을 사용한다.
 - `pstmt.setInt(2, member.getMoney())`: SQL의 두번째 ?에 값을 지정한다. `Int`형 숫자이므로 `setInt`를 지정한다.

- `pstmt.executeUpdate()`: `Statement`를 통해 준비된 SQL을 커넥션을 통해 실제 데이터베이스에 전달한다. 참고로 `executeUpdate()`은 `int`를 반환하는데 영향받은 DB row 수를 반환한다. 여기서는 하나의 row를 등록했으므로 1을 반환한다.

`executeUpdate()`

```
int executeUpdate() throws SQLException;
```

리소스 정리

쿼리를 실행하고 나면 리소스를 정리해야 한다. 여기서는 `Connection`, `PreparedStatement`를 사용했다. 리소스를 정리할 때는 항상 역순으로 해야한다. `Connection`을 먼저 획득하고 `Connection`을 통해 `PreparedStatement`를 만들었기 때문에 리소스를 반환할 때는 `PreparedStatement`를 먼저 종료하고, 그 다음에 `Connection`을 종료하면 된다. 참고로 여기서 사용하지 않은 `ResultSet`은 결과를 조회할 때 사용한다. 조금 뒤에 조회 부분에서 알아보자.

주의

리소스 정리는 꼭! 해주어야 한다. 따라서 예외가 발생하든, 하지 않은 항상 수행되어야 하므로 `finally` 구문에 주의해서 작성해야한다. 만약 이 부분을 놓치게 되면 커넥션이 끊어지지 않고 계속 유지되는 문제가 발생할 수 있다. 이런 것을 리소스 누수라고 하는데, 결과적으로 커넥션 부족으로 장애가 발생할 수 있다.

참고

`PreparedStatement`는 `Statement`의 자식 타입인데, ?를 통한 파라미터 바인딩을 가능하게 해준다. 참고로 SQL Injection 공격을 예방하려면 `PreparedStatement`를 통한 파라미터 바인딩 방식을 사용해야 한다.

이제 테스트 코드를 사용해서 JDBC로 회원을 데이터베이스에 등록해보자.

MemberRepositoryV0Test - 회원 등록

```
package hello.jdbc.repository;

import hello.jdbc.domain.Member;
import org.junit.jupiter.api.Test;

import java.sql.SQLException;

class MemberRepositoryV0Test {

    MemberRepositoryV0 repository = new MemberRepositoryV0();
```

```

@Test
void crud() throws SQLException {
    //save
    Member member = new Member("memberV0", 10000);
    repository.save(member);
}
}

```

실행 결과

데이터베이스에서 `select * from member` 쿼리를 실행하면 데이터가 저장된 것을 확인할 수 있다.

참고로 이 테스트는 2번 실행하면 PK 중복 오류가 발생한다. 이 경우 `delete from member` 쿼리로 데이터를 삭제한 다음에 다시 실행하자.

PK 중복 오류

```

org.h2.jdbc.JdbcSQLIntegrityConstraintViolationException: Unique index or
primary key violation: "PUBLIC.PRIMARY_KEY_8 ON PUBLIC.MEMBER(MEMBER_ID) VALUES
9"; SQL statement:

```

JDBC 개발 - 조회

이번에는 JDBC를 통해 이전에 저장한 데이터를 조회하는 기능을 개발해보자.

MemberRepositoryV0 - 회원 조회 추가

```

public Member findById(String memberId) throws SQLException {
    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) {
        log.error("db error", e);
        throw e;
    } finally {
        close(con, pstmt, rs);
    }
}

```

다음 코드도 추가하자

```
import java.util.NoSuchElementException;
```

findById() - 쿼리 실행

- sql: 데이터 조회를 위한 select SQL을 준비한다.
- rs = pstmt.executeQuery() 데이터를 변경할 때는 executeUpdate() 를 사용하지만, 데이터를 조회할 때는 executeQuery() 를 사용한다. executeQuery() 는 결과를 ResultSet 에 담아서 반환한다.

executeQuery()

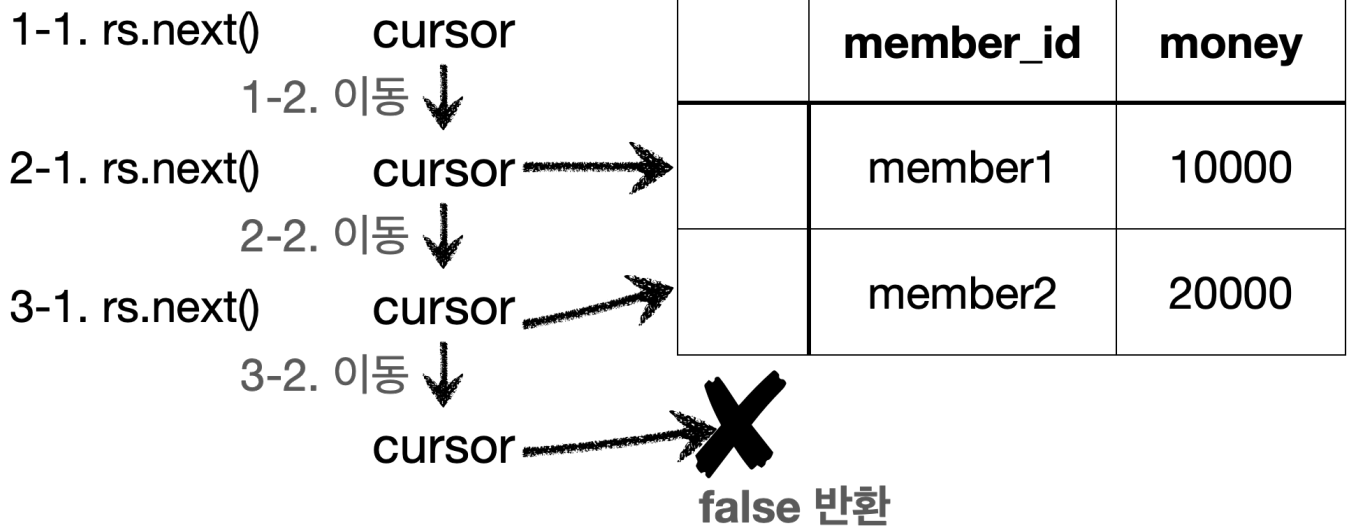
```
ResultSet executeQuery() throws SQLException;
```

ResultSet

- ResultSet 은 다음과 같이 생긴 데이터 구조이다. 보통 select 쿼리의 결과가 순서대로 들어간다.
 - 예를 들어서 select member_id, money 라고 지정하면 member_id, money 라는 이름으로 데이터가 저장된다.
 - 참고로 select * 을 사용하면 테이블의 모든 컬럼을 다 지정한다.
- ResultSet 내부에 있는 커서(cursor)를 이동해서 다음 데이터를 조회할 수 있다.
- rs.next(): 이것을 호출하면 커서가 다음으로 이동한다. 참고로 최초의 커서는 데이터를 가리키고 있지 않기 때문에 rs.next() 를 최초 한번은 호출해야 데이터를 조회할 수 있다.
 - rs.next() 의 결과가 true 면 커서의 이동 결과 데이터가 있다는 뜻이다.

- `rs.next()`의 결과가 `false`면 더이상 커서가 가리키는 데이터가 없다는 뜻이다.
- `rs.getString("member_id")`: 현재 커서가 가리키고 있는 위치의 `member_id` 데이터를 `String` 타입으로 반환한다.
- `rs.getInt("money")`: 현재 커서가 가리키고 있는 위치의 `money` 데이터를 `int` 타입으로 반환한다.

ResultSet 결과 예시



참고로 이 `ResultSet`의 결과 예시는 회원이 2명 조회되는 경우이다.

- 1-1에서 `rs.next()`를 호출한다.
- 1-2의 결과로 `cursor`가 다음으로 이동한다. 이 경우 `cursor`가 가리키는 데이터가 있으므로 `true`를 반환한다.
- 2-1에서 `rs.next()`를 호출한다.
- 2-2의 결과로 `cursor`가 다음으로 이동한다. 이 경우 `cursor`가 가리키는 데이터가 있으므로 `true`를 반환한다.
- 3-1에서 `rs.next()`를 호출한다.
- 3-2의 결과로 `cursor`가 다음으로 이동한다. 이 경우 `cursor`가 가리키는 데이터가 없으므로 `false`를 반환한다.

`findById()`에서는 회원 하나를 조회하는 것이 목적이다. 따라서 조회 결과가 항상 1건이므로 `while` 대신에 `if`를 사용한다. 다음 SQL을 보면 PK인 `member_id`를 항상 지정하는 것을 확인할 수 있다.

SQL: `select * from member where member_id = ?`

MemberRepositoryV0Test - 회원 조회 추가

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

```

import hello.jdbc.domain.Member;
import lombok.extern.slf4j.Slf4j;
import org.junit.jupiter.api.Test;

import java.sql.SQLException;

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

@Slf4j
class MemberRepositoryV0Test {

    MemberRepositoryV0 repository = new MemberRepositoryV0();

    @Test
    void crud() throws SQLException {
        //save
        Member member = new Member("memberV0", 10000);
        repository.save(member);

        //findById
        Member findMember = repository.findById(member.getMemberId());
        log.info("findMember={}", findMember);
        assertThat(findMember).isEqualTo(member);
    }
}

```

실행 결과

MemberRepositoryV0Test - findMember=Member(memberId=memberV0, money=10000)

- 회원을 등록하고 그 결과를 바로 조회해서 확인해보았다.
- 참고로 실행 결과에 member 객체의 참조 값이 아니라 실제 데이터가 보이는 이유는 롬복의 @Data가 toString()을 적절히 오버라이딩 해서 보여주기 때문이다.
- isEqualTo(): findMember.equals(member)를 비교한다. 결과가 참인 이유는 롬복의 @Data는 해당 객체의 모든 필드를 사용하도록 equals()를 오버라이딩 하기 때문이다.

참고

이 테스트는 2번 실행하면 PK 중복 오류가 발생한다. 이 경우 delete from member 쿼리로 데이터를 삭제한 다음에 다시 실행하자.

PK 중복 오류

org.h2.jdbc.JdbcSQLIntegrityConstraintViolationException: Unique index or

```
primary key violation: "PUBLIC.PRIMARY_KEY_8 ON PUBLIC.MEMBER(MEMBER_ID) VALUES
9"; SQL statement:
```

JDBC 개발 - 수정, 삭제

수정과 삭제는 등록과 비슷하다. 등록, 수정, 삭제처럼 데이터를 변경하는 쿼리는 `executeUpdate()` 를 사용하면 된다.

MemberRepositoryV0 - 회원 수정 추가

```
public void update(String memberId, int money) throws SQLException {
    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);
        int resultSize = pstmt.executeUpdate();
        log.info("resultSize={}", resultSize);
    } catch (SQLException e) {
        log.error("db error", e);
        throw e;
    } finally {
        close(con, pstmt, null);
    }
}
```

`executeUpdate()` 는 쿼리를 실행하고 영향받은 row수를 반환한다. 여기서는 하나의 데이터만 변경하기 때문에 결과로 1이 반환된다. 만약 회원이 100명이고, 모든 회원의 데이터를 한번에 수정하는 update sql을 실행하면 결과는 100이 된다.

MemberRepositoryV0Test - 회원 수정 추가

```
@Test
void crud() throws SQLException {
```



```

//save
Member member = new Member("memberV0", 10000);
repository.save(member);

//findById
Member findMember = repository.findById(member.getMemberId());
assertThat(findMember).isEqualTo(member);

//update: money: 10000 -> 20000
repository.update(member.getMemberId(), 20000);
Member updatedMember = repository.findById(member.getMemberId());
assertThat(updatedMember.getMoney()).isEqualTo(20000);
}

```

회원 데이터의 money 를 10000 → 20000으로 수정하고, DB에서 데이터를 다시 조회해서 20000으로 변경 되었는지 검증한다.

실행 로그

MemberRepositoryV0 - resultSize=1
 pstmt.executeUpdate() 의 결과가 1인 것을 확인할 수 있다. 이것은 해당 SQL에 영향을 받은 로우 수가 1개라는 뜻이다.

데이터베이스에서 조회하면 memberV0의 money가 20000으로 변경된 것을 확인할 수 있다.

```
select * from member;
```

참고

이 테스트는 2번 실행하면 PK 중복 오류가 발생한다. 이 경우 delete from member 쿼리로 데이터를 삭제한 다음에 다시 실행하자.

이번에는 회원을 삭제해보자.

MemberRepositoryV0 - 회원 삭제 추가

```

public void delete(String memberId) throws SQLException {
    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) {
        log.error("db error", e);
        throw e;
    } finally {
        close(con, pstmt, null);
    }
}

```

쿼리만 변경되고 내용은 거의 같다.

MemberRepositoryV0Test - 회원 삭제 추가

```

package hello.jdbc.repository;

import hello.jdbc.domain.Member;
import lombok.extern.slf4j.Slf4j;
import org.junit.jupiter.api.Test;

import java.sql.SQLException;
import java.util.NoSuchElementException;

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

@Slf4j
class MemberRepositoryV0Test {

    MemberRepositoryV0 repository = new MemberRepositoryV0();

    @Test
    void crud() throws SQLException {
        //save
        Member member = new Member("memberV0", 10000);
        repository.save(member);

        //findById
        Member findMember = repository.findById(member.getMemberId());
        log.info("findMember={}", findMember);
        assertThat(findMember).isEqualTo(member);
    }
}

```

```

        //update: money: 10000 -> 20000
        repository.update(member.getMemberId(), 20000);
        Member updatedMember = repository.findById(member.getMemberId());
        assertThat(updatedMember.getMoney()).isEqualTo(20000);

        //delete
        repository.delete(member.getMemberId());
        assertThatThrownBy(() -> repository.findById(member.getMemberId()))
            .isInstanceOf(NoSuchElementException.class);
    }
}

```

회원을 삭제한 다음 `findById()` 를 통해서 조회한다. 회원이 없기 때문에 `NoSuchElementException` 이 발생한다. `assertThatThrownBy` 는 해당 예외가 발생해야 검증에 성공한다.

참고

마지막에 회원을 삭제하기 때문에 테스트가 정상 수행되면, 이제부터는 같은 테스트를 반복해서 실행할 수 있다. 물론 테스트 중간에 오류가 발생해서 삭제 로직을 수행할 수 없다면 테스트를 반복해서 실행할 수 없다. 트랜잭션을 활용하면 이 문제를 깔끔하게 해결할 수 있는데, 자세한 내용은 뒤에서 설명한다.

정리