

5. ADVANCED SQL

JDBC Programming

JDBC CODE

```
public static void JDBCexample(String dbid, String userid, String passwd)
{
    try {
        Class.forName ("oracle.jdbc.driver.OracleDriver");
        Connection conn = DriverManager.getConnection(
            "jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);
        Statement stmt = conn.createStatement();
        ... Do Actual Work ....
        stmt.close();
        conn.close();
    }
    catch (SQLException sqle) {
        System.out.println("SQLException : " + sqle);
    }
}
```

1.DB에 접속해서 connection 열기

2. connection 받기

- JDBC의 역할 : 자바 프로그램이 DB와 관련된 작업을 처리할 수 있게 도움
- Java나 SQL 패키지 import 하기 (코드엔 없음) `// JDBC 사용을 위해`

JDBC CODE

```
public static void JDBCexample(String dbid, String userid, String passwd)
{
    try {
        Class.forName ("oracle.jdbc.driver.OracleDriver");
        Connection conn = DriverManager.getConnection(
            "jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);
        Statement stmt = conn.createStatement();
        ... Do Actual Work ....
        stmt.close();
        conn.close();
    }
    catch (SQLException sqle) {
        System.out.println("SQLException : " + sqle);
    }
}
```

Class.forName 메소드 사용 → JDBC 드라이버 로드

JDBC CODE

```
public static void JDBCexample(String dbid, String userid, String passwd)
{
    try {
        Class.forName ("oracle.jdbc.driver.OracleDriver");
        Connection conn = DriverManager.getConnection(
            "jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);
        Statement stmt = conn.createStatement();
        ... Do Actual Work ....
        stmt.close();
        conn.close();
    }
    catch (SQLException sqle) {
        System.out.println("SQLException : " + sqle);
    }
}
```

Connection 객체 생성 →

1) 참조 변수(conn) 선언

2) DriverManager.getConnection 메소드 사용 (매개변수 3개)

JDBC CODE

```
public static void JDBCexample(String dbid, String userid, String passwd)
{
    try {
        Class.forName ("oracle.jdbc.driver.OracleDriver");
        Connection conn = DriverManager.getConnection(
            "jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);
        Statement stmt = conn.createStatement();
        ... Do Actual Work ....
        stmt.close();
        conn.close();
    }
    catch (SQLException sqle) {
        System.out.println("SQLException : " + sqle);
    }
}
```

첫 번째 매개변수(url)는 4부분으로 나뉨

- | | |
|------------------------|--------------------------|
| - jdbc : oracle : thin | // 프로토콜 |
| - @db.yale.edu | // DBMS가 있는 url 또는 머신 이름 |
| - 2000 | // 시스템이 사용하는 포트 번호 |
| - univdb | // 사용할 DB 이름 |

각 부분이 :로 구분되어 있음

매개변수 3개 (url, userid, passwd)

JDBC CODE

```
public static void JDBCexample(String dbid, String userid, String passwd)
{
    try {
        Class.forName ("oracle.jdbc.driver.OracleDriver");
        Connection conn = DriverManager.getConnection(
            "jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);
        Statement stmt = conn.createStatement();
        ... Do Actual Work ....
        stmt.close();
        conn.close();
    }
    catch (SQLException sqle) {
        System.out.println("SQLException : " + sqle);
    }
}
```

userid, passwd = DB 시스템에 접속하기 위한 사용자의 ID, 패스워드

JDBC CODE

```
public static void JDBCexample(String dbid, String userid, String passwd)
{
    try {
        Class.forName ("oracle.jdbc.driver.OracleDriver");
        Connection conn = DriverManager.getConnection(
            "jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);
        Statement stmt = conn.createStatement();
        ... Do Actual Work ....
        stmt.close();
        conn.close();
    }
    catch (SQLException sqle) {
        System.out.println("SQLException : " + sqle);
    }
}
```

- SQL 문장 실행 →
 1. connection 객체의 createStatement 메소드 사용
 2. Statement 객체 생성


JDBC CODE

```
public static void JDBCexample(String dbid, String userid, String passwd)
{
    try {
        Class.forName ("oracle.jdbc.driver.OracleDriver");
        Connection conn = DriverManager.getConnection(
            "jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);
        Statement stmt = conn.createStatement();
        ... Do Actual Work ....
        stmt.close();
        conn.close();
    }
    catch (SQLException sqle) {
        System.out.println("SQLException : " + sqle);
    }
}
```

- 실질적으로 데이터를 다루는 작업
- INSERT, DELETE, UPDATE, QUERY(질의) 등

JDBC CODE

```
public static void JDBCexample(String dbid, String userid, String passwd)
{
    try {
        Class.forName ("oracle.jdbc.driver.OracleDriver");
        Connection conn = DriverManager.getConnection(
            "jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);
        Statement stmt = conn.createStatement();
        ... Do Actual Work ....
        stmt.close();
        conn.close();
    }
    catch (SQLException sqle) {
        System.out.println("SQLException : " + sqle);
    }
}
```



< 사용한 자원 시스템에 돌려주기 >

Statement 객체 반납 → Connection 객체

왜 필요한가?

1. 컴퓨터에게 프로그램의 종료 시점을 알려주기 위해

2. Connection 사용 후 계속 끊지 않으면, 나중엔 더 이상 connection 요청 못 함

JDBC CODE

```
public static void JDBCexample(String dbid, String userid, String passwd)
{
    try {
        Class.forName ("oracle.jdbc.driver.OracleDriver");
        Connection conn = DriverManager.getConnection(
            "jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);
        Statement stmt = conn.createStatement();
        ... Do Actual Work ....
        stmt.close();
        conn.close();
    }
    catch (SQLException sqle) {
        System.out.println("SQLException : " + sqle);
    }
}
```

try – catch 문으로 구현할 코드 감싸기 (exception 에러 핸들링)

JDBC CODE – EXECUTEUPDATE

1. DB 업데이트 - executeUpdate

```
try {  
    stmt.executeUpdate(  
        "insert into instructor values('77987', 'Kim', 'Physics', 98000)");  
} catch (SQLException sqle)  
{  
    System.out.println("Could not insert tuple. " + sqle);  
}
```

- INSERT, DELETE, UPDATE 문 실행
- DDL 사용 시 executeUpdate 문 사용
- SQL 문 실행 시 결과로 릴레이션 돌려받지 X

JDBC CODE – EXECUTEQUERY

2. 쿼리 수행 후 결과 출력 - executeQuery

```
ResultSet rset = stmt.executeQuery(  
    "select dept_name, avg (salary)  
    from instructor  
    group by dept_name");  
while (rset.next()) {  
    System.out.println(rset.getString("dept_name") + " " +  
        rset.getFloat(2));  
}
```

- 학과별 salary를 구하는 쿼리 → ResultSet의 객체의 참조변수가 결과 릴레이션의 첫 번째 튜플의 앞부분 가리킴
- while 문에서 next 메소드 사용 → 결과 출력 (튜플이 있으면 true, 출력)

JDBC CODE – EXECUTE

1. DB 업데이트 - `executeUpdate`

2. 쿼리 수행 후 결과 출력 – `executeQuery`

- `select` 절 : 데이터를 질의하여 릴레이션 돌려받음
- `ResultSet`의 커서 기능 : 위치를 가리키는 포인터 역할
- `ResultSet`의 `next` 메소드 : 커서 위치를 다음 튜플로 옮기고 그 튜플에 값이 있는지 체크

`executeUpdate` = 결과 set 돌려받지 X

`executeQuery` = 결과 set 돌려받음

JDBC CODE DETAILS

예) result 필드에 있는 값 가져오기

```
rs.getString("dept_name") and rs.getString(1) → (첫번째)
```

속성 이름 몇번째 속성인지

dept_name 속성이 결과의 첫 번째 속성이라면, and 좌우에서 가져오는 값은 같은 값임

예) Null 값 다루기

```
int a = rs.getInt("a");  
if (rs.isNull()) Systems.out.println("Got null value");
```

isNull 메소드는 a 속성이 null 인지 판단해줌

PREPARED STATEMENT

- PreparedStatement 메소드 = SQL 문이 인자를 가짐

```
PreparedStatement pstmt = conn.prepareStatement(  
    "insert into instructor values(?,?,?,?)");  
pstmt.setString(1, "88877");    pstmt.setString(2, "Perry");  
pstmt.setString(3, "Finance");  pstmt.setInt(4, 125000);  
pstmt.executeUpdate();  
pstmt.setString(1, "88878");  
pstmt.executeUpdate();
```

- SQL 문을 실제로 실행하기 위해 executeUpdate 또는 executeQuery 메소드 사용해야 함
- preparedStatement의 SQL 문을 실행하려면, ? 부분의 매개변수 값을 정해야 함
→ set 메소드 사용 (값 정해줌)

PREPARED STATEMENT

```
PreparedStatement pstmt = conn.prepareStatement(  
    "insert into instructor values(?,?,?,?)");  
pstmt.setString(1, "88877");    pstmt.setString(2, "Perry");  
pstmt.setString(3, "Finance");  pstmt.setInt(4, 125000);  
pstmt.executeUpdate();  
pstmt.setString(1, "88878");  
pstmt.executeUpdate();
```

- ? 부분의 매개변수 값 정해주기
- 형식 = set데이터타입(몇번째, "값");

PREPARED STATEMENT

```
PreparedStatement pstmt = conn.prepareStatement(
    "insert into instructor values(?,?,?,?)");
pstmt.setString(1, "88877");    pstmt.setString(2, "Perry");
pstmt.setString(3, "Finance");  pstmt.setInt(4, 125000);
pstmt.executeUpdate();
pstmt.setString(1, "88878");
pstmt.executeUpdate();
```

- executeUpdate 메소드 실행 : SQL의 insert 문 실행
→ 윗줄에서 입력한 대로 릴레이션 ?의 각 자리에 튜플의 값이 들어감
- Set으로 값이 다시 지정되기 전까지는 기존 값 유지함 (88877, Perry, Finance, 125000)

PREPARED STATEMENT

```
PreparedStatement pstmt = conn.prepareStatement(
    "insert into instructor values(?,?,?,?)");
pstmt.setString(1, "88877");    pstmt.setString(2, "Perry");
pstmt.setString(3, "Finance");  pstmt.setInt(4, 125000);
pstmt.executeUpdate();
pstmt.setString(1, "88878");
pstmt.executeUpdate();
```

- *instructor* 릴레이션의 첫번째 튜플 값이 88877 → 88878로 바뀜

PREPARED STATEMENT

- PreparedStatement 객체도 ResultSet을 돌려주는 executeQuery 메소드로 질의 가능
- 왜 PreparedStatement 사용?

반복적인 일을 할 경우, 효과적으로 실행!

(쿼리 1번만 컴파일 → 다른 매개변수 값으로 여러 번 실행)

- 사용자의 입력을 쿼리에 추가할 때는 반드시 prepared statement 사용
- 안 그러면, 아래와 같은 오류 발생

```
"insert into instructor values(" + ID + " ',' " + name + " ',' " +  
                                dept name + " ',' " balance + ")"
```

- 위의 취약점은 SQL injection 기술을 사용하여 데이터를 훔치거나 DB에 손상을 주는 데 사용될 수 있음 π (그러니, 반드시 입력은 prepared statement 사용하기)

METADATA

- DB의 메타데이터 가져오는 법

예) ResultSet의 모든 column의 타입과 이름 출력하기

```
ResultSetMetaData rsmd = rs.getMetaData();  
for(int i = 1; i <= rsmd.getColumnCount(); i++) {  
    System.out.println(rsmd.getColumnName(i));  
    System.out.println(rsmd.getColumnTypeName(i));  
}
```

- 주의! JDBC에서는 인덱스가 1부터 시작함

METADATA

- **ResultSet** 인터페이스의 **getMetaData** 메소드
: 결과 집합에 대한 메타데이터를 제공하는 **ResultSetMetaData** 객체 반환
- 메타데이터 정보를 가져오는 메소드
 1. **getColumnCount** : **ResultSetMetaData** 인터페이스에 속하는 결과 개수
 2. **getColumnName** : 명시된 열의 이름
 3. **getColumnTypeName** : 명시된 열의 데이터 타입
- 메타데이터 정보 = 릴레이션의 구조를 알아야 하는 작업에 유용

TRANSACTION CONTROL (JDBC)

- JDBC에서는 각각의 SQL 문을 개별적인 트랜잭션으로 취급
 - 각각의 SQL 문을 자동적으로 커밋시킴 (여러 업데이트로 이루어진 트랜잭션에서는 안 좋음)
- Connection에 있는 auto commit 끄기 = **conn.setAutoCommit(false);**
- 트랜잭션의 자동 커밋을 끄면, 반드시 명시적으로 커밋 or 롤백 작업 해야 함
 - `conn.commit();` // 영구적으로 DB에 반영
 - `conn.rollback();` // 오류 발생 → 이전 상태로 복구
- auto commit 켜기 = **conn.setAutoCommit(true);**
- try 블록 안에 트랜잭션 넣기 (exception 에러 발생할 수도 있으니까)
- catch 안에 rollback 호출 넣기 (예외 처리)

JDBC 특징

1. DB 시스템에 저장된 프로시저나 함수 호출 지원 – **CallableStatement** 인터페이스 사용
 - CallableStatement 객체는 PreparedStatement 객체를 상속받음

예) 리턴값이 매개변수 안으로 들어감

```
CallableStatement cStmt1 = conn.prepareCall("{? = call some  
function(?)})");
```

예) 리턴값이 없는 함수나 프로시저 호출

```
CallableStatement cStmt2 = conn.prepareCall("{call some  
procedure(?,?)})");
```

JDBC 특징

2. Large Object Type을 다룰 수 있음 – **getBlob**, **getClob** 메소드 사용

- `getBlob()`, `getClob()`과 `getString()`의 차이

 - : 각각 Blob, Clob 형식의 객체(포인터) 반환

- `getBytes()` : 실제로 데이터를 DB로부터 가져올 때 사용

- `setBlob/Clob()` : 데이터타입이 Blob/Clob인 column을 입력 스트림에 연결

예) `blob.setBlob(int parameterIndex, InputStream inputStream).`