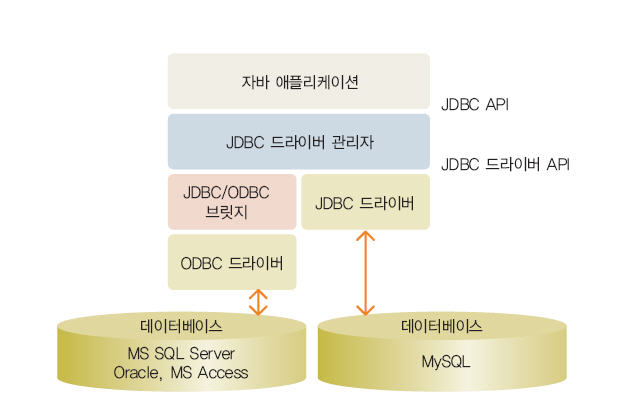
**데이터베이스 프로그래밍**

• 상업적인 프로그램을 작성하다 보면 항상 닥치는 문제가 데이터를 어디에 저장하느냐이다. 물론 간단한 프로그램 같으면 파일에 저장하면 되겠지만 업무용 프로그램은 데이터를 데이터베이스에 저장하여야 한다. 이번 장에서는 자바를 이용하여 데이터베이스에 접근하고 데이터를 추가, 수정, 삭제하는 방법을 살펴본다.

**학습목표**

**1. 자바와 데이터베이스**

**JDBC(Java DataBase Connectivity)**는 자바 API의 하나로서 데이터베이스에 연결되어서 데이터베이스 안의 데이터에 대하여 검색하고 데이터를 변경할 수 있게 한다. 자바의 개발진들은 초창기부터 자바로 데이터베이스를 접근할 수 있기를 원했다. 1995년에 표준 자바 라이브러리를 확장하여 데이터베이스를 접근하려 하였는데 처음 목표는 순수하게 자바만을 사용하여서 어떤 데이터베이스라도 접근하게 하는 것이었으나 시장에는 많은 종류의 데이터베이스가 있어서 이것은 처음부터 불가능한 일이었다. 따라서 Sun사는 애플리케이션 프로그램머를 위하여 JDBC API를 제공하고, 데이터베이스 업체들을 위해서는 JDBC 드라이버 API를 제공하여서 많은 데이터베이스 업체들이 그들의 데이터베이스를 위한 드라이버를 개발할 수 있도록 하였다. 즉 애플리케이션은 드라이버 관리자에게 요청을 하고 드라이버 관리자는 드라이버를 통하여 데이터베이스에게 요청을 하게 된다.



**1) 데이터베이스 프로그램 개발 절차**

① DBMS(DataBase Management System)를 설치하여야 한다.

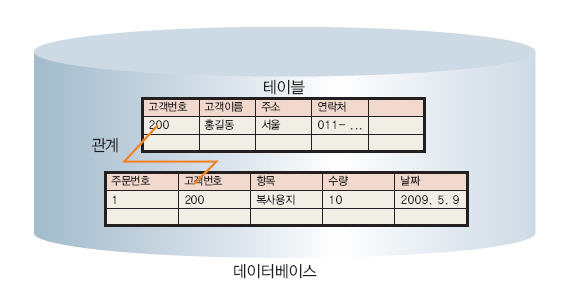
② 자신이 설치한 DBMS에 필요한 JDBC 드라이버를 설치한다.

③ JDBC가 제공하는 기능을 이용하여 데이터베이스 응용 프로그램을 개발한다. 자바의 JDBC API가 하는 일은 자바 프로그램에서 SQL 명령어들을 데이터베이스 관리 시스템으로 보낼 수 있도록 자바와 데이터베이스를 연결하는 것이다.

**2. 데이터베이스의 기초**

**1) 데이터베이스란?**

데이터베이스는 데이터가 빠르게 추출될 수 있도록 데이터를 조직화하여서 저장하는 방법이다. 데이터베이스 관리 시스템(DBMS)은 다수의 사용자를 위하여 데이터가 저장, 접근, 변경되는 기능을 구현한다. 현재 가장 인기있는 데이터베이스 시스템은 관계형 데이터베이스(Relational Database)이다. 관계형 데이터베이스에서는 여러 개의 테이블이 존재하고 테이블과 테이블 간에는 공통적인 데이터로 인하여 어떤 관계가 성립될 수 있다.

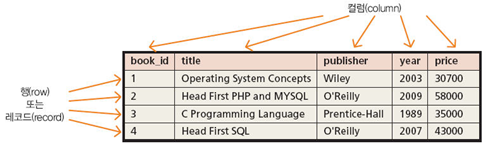


가장 많이 사용되는 DBMS는 오라클, 마이크로소프트의 SQL Server, 사이베이스, MySQL 등을 들 수 있다. 관계형 데이터베이스는 SQL이라고 불리는 언어로 사용할 수 있다. SQL(Standard Query Language)은 데이터베이스를 조작하는 국제적인 표준 언어이다.

**2) 테이블**

테이블의 하나의 행(row)은 레코드(record)라고 불린다. 레코드는 여러 개의 컬럼(column)으로 이루어져 있다. 테이블은 무결성 법칙을 따라서 작성되어야 하며 테이블에서 각 레코드는 중복하지 않아야 한다는 것이다. 그래서 대부분의 DBMS에서 사용자는 중복된 레코드가 허용되지 않도록 설정할 수 있다.

테이블에서는 하나의 컬럼을 이용하여 레코드와 레코드를 구분할 수 있다. 이러한 특정한 컬럼을 주요키(primary key)라고 한다. 주요키는 null이 되면 안 된다.



3) MYSQL

데이터베이스에 대하여 여러 가지 프로그래밍을 하려면 데이터베이스 시스템이 설치되어 있어야 한다. 가장 손쉽게 무료로 설치할 수 있는 데이터베이스는 MySQL이다.

MySQL 설치(<http://www.mysql.com>)

**3. SQL**

**1) SQL이란?**

SQL은 관계형 데이터베이스에서 사용하기 위하여 설계된 언어이다. 표준적인 SQL 명령어들이 있으며 이것은 모든 관계형 데이터베이스에 의하여 지원된다.



2) 데이터베이스 생성하기

데이터베이스에 데이터를 저장하기 전에 당연히 데이터베이스부터 생성하여야 한다. CREATE 구문을 사용하여 생성하면 되고 SQL의 모든 문장은 ;으로 끝나야 한다.



기존의 데이터베이스가 있으면 삭제

|  |
| --- |
| ① DROP DATABASE book\_db;  ② CREATE DATABASE book\_db;  데이터베이스 생성  ③ USE book\_db;  ④ CREATE TABLE books (  데이터베이스 사용  ⑤ book\_id int not null auto\_increment,  데이블 생성  ⑥ title varchar(50),  ⑦ publisher varchar(30),  ⑧ year varchar(10),  ⑨ price int,  ⑩ PRIMARY KEY(book\_id)  ⑪ ); |

**3) 레코드 추가하기**

데이터베이스를 생성하였으면 다음 작업은 레코드를 추가하는 것이다. 레코드 추가는 INSERT 명령어를 사용한다.



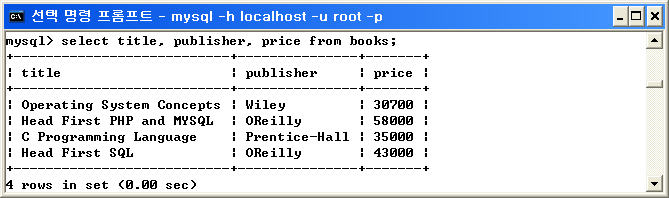
|  |
| --- |
| INSERT INTO books (title, publisher, year, price) VALUES ('Operating System Concepts', 'Wiley', '2003',30700);  INSERT INTO books (title, publisher, year, price) VALUES ('Head First PHP and MYSQL', 'OReilly', '2009', 58000);  INSERT INTO books (title, publisher, year, price) VALUES ('C Programming Language', 'Prentice-Hall', '1989', 35000);  INSERT INTO books (title, publisher, year, price) VALUES ('Head First SQL', 'OReilly', '2007', 43000); |

4) 레코드 검색하기

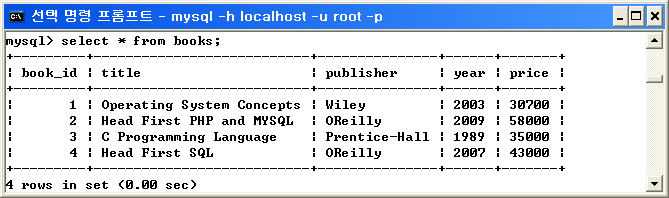
SELECT문장은 쿼리(Query)라고도 불리는데 테이블에서 정보를 얻어내기 위하여 사용된다. 이 명령어는 출력하고 싶은 컬럼과 테이블을 지정한다.



다음의 SELECT 명령어는 books라는 테이블에서 title, publisher, price 컬럼을 출력한다. FROM절은 테이블을 지정한다. (select \* from books;)



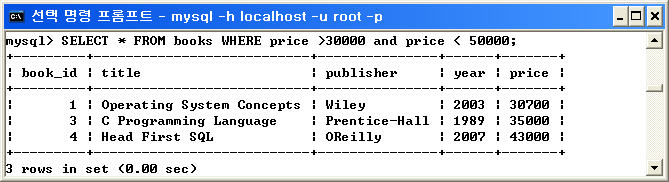
SELECT 명령어의 결과로 나오는 레코드들의 집합을 결과 집합(RESULT SET)이라고 한다. 테이블 안의 모든 레코드를 포함하는 결과 집합을 생성하기 한다. 이때 \*는 모든 컬럼을 선택함을 의미한다.



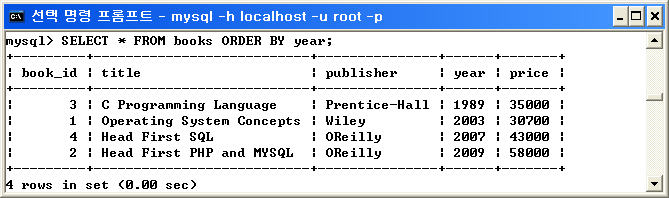
SELECT를 사용할 때 선택을 위한 조건을 명시할 수 있다. WHERE절은 레코드를 선택하는 기준을 제공한다.



키워드 LIKE는 문자열을 비교할 때 사용되면 이것은 와일드 카드를 가지고 있는 패턴을 사용할 수 있다. % 기호는 0개 이상의 문자를 더하는 것을 의미한다. 다른 와일드 카드로는 \_는 하나의 글자를 대신 사용할 수 있다.



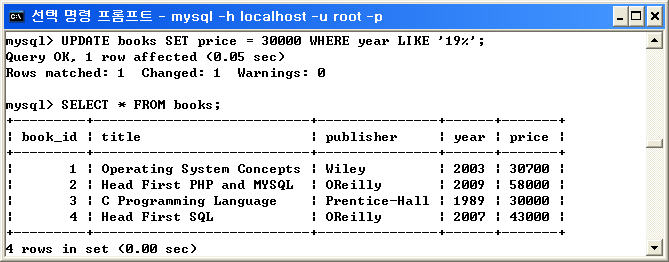
레코드들을 정렬하여 출력하려면 ORDER BY를 사용한다.



**5) 레코드 수정하기**

SQL의 UPDATE 명령어를 사용하면 레코드를 변경할 수 있다. UPDATE 명령어는 다음과 같은 구문을 가진다. 만약 조건을 주지 않으면 모든 레코드가 변경된다.

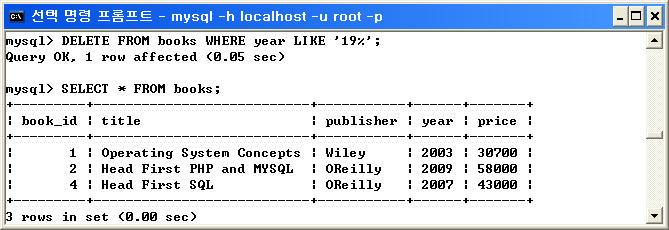




**6) 레코드 삭제하기**

SQL의 DELETE 명령어를 사용하면 현재의 레코드를 삭제할 수 있다.





**7) 결과 집합(Result Set)와 커서(Cursors)**

쿼리의 조건을 만족하는 레코드들의 집합이 **결과 집합(Result Set)**이다. 결과 집합에서 사용자는 커서를 사용하여 한 번에 한 레코드씩 데이터에 접근할 수 있다. **커서(Cursors)**는 결과 집합의 레코드들을 포함하고 있는 파일에 대한 포인터라고 간주할 수 있다. 이 포인터는 현재 접근되고 있는 레코드들을 가리킨다. 커서는 사용자로 하여금 결과 집합에서 각각의 레코드들을 처리할 수 있도록 도와준다. 커서는 레코드들에 대하여 반복 처리를 할 때에 이용된다. 대부분의 DBMS는 결과 집합이 생성될 때 커서가 자동적ㅇ로 만들어진다. 커서는 정방향이나 역방향으로 움직일 수 있고 특정한 레코드로 이용할 수 있다.

**4. JDBC를 이용한 프로그래밍**

**1) JDBC 드라이버 설치**

자바와 데이터베이스를 연결하는 형태는 여러 가지가 있지만 가장 간명한 것은 해당 데이터베이스 회사에서 제공하는 JDBC 드라이버를 설치하는 것이다. JDBC 드라이버란 자바가 특정한 회사 제품의 데이터베이스에 접근할 수 있도록 해주는 드라이버의 일종이다. 이때 해당 데이터베이스 회사가 제공하는 JDBC 드라이버가 있어야만이 자바가 그 데이터베이스를 사용할 수 있다.

① MySQL에 대한 드라이버는 www.mysql.com/products/connector로부터 드라이버를 다운로드받아서 압축을 푼다.

② 다음은 자바 가상 기계가 이 드라이버 파일을 찾을 수 있도록 하여야 한다.

오라클은 C:\app\jwsev\product\18.0.0\dbhomeXE\jdbc\lib 폴더에 있는 ojdbc8.jar 파일을

C:\Project\jdk-15.0.2\lib 폴더에 복사한다.

이클립스에서 Build Path에서 라이브러리를 추가한다.

**2) JDBC를 이용한 데이터베이스 사용 절차**

JDBC를 이용하여 데이터베이스를 사용하는 전형적인 절차는 다음과 같다.

① URL로 지정된 JDBC 드라이버를 적재(load)한다.

② 사용자 이름과 패스워드를 가지고 데이터베이스에 연결한다.

③ SQL문장을 작성하여서 전송하고 실행한다. SQL명령어의 결과로 생성되는 결과 집합을 얻는다.

④ 결과 집합을 화면에 표시하거나 결과 집합을 처리한다. 사용이 끝나면 연결을 해제한다.



**3) 드라이버 클래스 적재**

첫 번째 단계는 드라이버 클래스를 등록하여서 프로그램 안으로 드라이버 클래스 파일을 적재하여야 한다. Class 클래스의 forName()이라는 메소드를 사용할 수 있다.

|  |
| --- |
| try {  **Class.forName("oracle.jdbc.driver.OracleDriver");** //Class.forName("com.mysql.jdbc.Driver");  지정된 이름의 클래스를 찾아서 메모리로 적재한다.  } catch (ClassNotFoundException e) {  System.out.println("드라이버를 찾을 수 없습니다“);  } |

**4) 데이터베이스 연결**

드라이버를 적재하였으면 드라이버를 통하여 데이터베이스 시스템과 연결할 수 있다. 연결하기 위해서는 DriverManager 클래스의 정적 메소드인 getConnection()을 호출한다. 이 메소드는 데이터베이스 연결을 확립하게 하고 매개변수로 데이터베이스 URL, 사용자 아이디, 패스워드를 요구한다.

|  |
| --- |
| String url = **"jdbc:oracle:thin:@127.0.0.1:1521:xe";** // "jdbc:mysql://localhost/book\_db";  String user = **"hr";** // "root";  String password = **"1234";** // "1234";  con = **DriverManager.getConnection(url, user, password);** |

MySQL 접속할 때는 jdbc:subprotocol:subname으로 구성되는데 subprotocol는 데이터베이스 이름을 적으면 되고, 로컬컴퓨터에 있는 데이터베이스라면 //localhost/book\_db 와 같이 작성하면 된다. getConnection()메소드는 SQLException을 발생할 수 있다.

**JDBC를 이용한 데이터베이스 연결 방법**

1 단계 : import java.sql.\*;

2 단계 : 드라이버를 로드 한다.

3 단계 : Connection 객체를 생성한다.

4 단계 : Statement 객체를 생성한다.

5 단계 : SQL문에 결과물이 있다면 ResultSet 객체를 생성한다.

6 단계 : 모든 객체를 닫는다.

드라이버 다운 받기와 설정

JDBC API를 이용해서 DBMS에 접근하기 위해서는 DBMS에서 제공되는 드라이버를 내려 받아야 한다.

만약, 오라클이 설치되어 있다면 아래의 경로에서 드라이버를 제공하고 있다.

|  |
| --- |
| C:\app\jwsev\product\18.0.0\dbhomeXE\jdbc\lib\ojdbc8.jar |

C:\Project\jdk-15.0.2\lib\ojdbc8.jar 에 복사한다.

이클립스에서 JDBC 드라이버 추가하기(Build Path)

**JDBC 드라이버 로딩 & Connection 객체 생성**

데이터베이스에 연결하기 위해서는 2가지 과정이 필요하다. 가장 먼저 드라이버 클래스를 로딩을 해야 하고 그 다음은 DB와 접속을 시도하는 것이다. DriverManager는 Class.forName()[이 메소드는 인터페이스 드라이버를 구현하는 기능을 한다.] 메소드를 통해서 생성된다.

Class.forName()는 드라이버 로딩시 DriverManager.registerDriver()메소드를 자동으로 호출하여 레지스터하기 때문에 DriverManager.registerDriver() 메소드를 명시하여 레지스터 할 수 없다.

즉 Class.forName()처럼 특정 클래스를 로딩하면 자동으로 객체가 생성되고 DriverManager에 등록된다. 드라이버 클래스를 찾지 못할 경우 forName() 메소드는 ClassNotFoundException 예외를 발생시키므로 반드시 예외를 처리를 해야 한다.

일반적으로 드라이버 클래스들은 로드될 때 자신의 인스턴스(객체)를 생성하고, 자동적으로 DriverManager 클래스 메소드를 호출하여 그 인스턴스를 등록한다. DriverManager 클래스의 모든 메소드는 static이기 때문에 반드시 객체를 생성시킬 필요가 없다. DriverManager 클래스의 Connection 인터페이스의 구현 객체를 생성하는데 getConnection 메소드를 사용한다

특정 데이터원본에 대한 커넥션은 Connection 인터페이스가 구현된 클래스의 객체로 표현된다. 어떤 SQL문을 실행시키기 전에 우선 Connection 객체가 있어야 한다. Connection객체는 특정 데이터원본과 연결된 커넥션을 나타내고 특정한 SQL문을 정의하고 실행시킬 수 있는 Statement 객체를 생성할 때도

Connection 객체를 사용한다.

[형식1]

|  |
| --- |
| 1-1. Class.forName(“드라이버 클래스명”) //벤더가 제공한 클래스명 기재  또는 클래스패스에 설정되어져 있는 벤더측 JDBC 드라이버 클래스 객체를 통해서 로딩이 가능하다.  1-2. DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());  1.3 Connection conn = DriverManage.getConnection(“연결 url”, “id”, ”password”); |

[형식 2]

|  |
| --- |
| String url = “jdbc:oracle:thin:@127.0.0.1:1521:orcl”;  Properties langType = new Properties();  langType.put(“user”,”userid”);  langType.put(“password”,”passwd”);  langType.put(“charSet”,”8859\_1”);  Class.forName(“드라이버 클래스명”);  Connection conn = DriverManager.getConnection(url, langType); |

- 드라이별 데이터베이스 연결 정보

|  |  |  |
| --- | --- | --- |
| 드라이버 | 드라이버 클래스 | 연결 URL |
| JDBC-ODBC | sun.jdbc.odbc.JdbcOdbcDriver | “jdbc:odbc:<dbname>” |
| **Oracle Thin** | **oracle.jdbc.driver.OracleDriver** | **“jdbc:oracle:thin:@<serverip>:port:SID”** |
| Oracle OCI | oracle.jdbc.driver.OracleDriver | “jdbc:oracle:oci8:@<sid>” |
| MS-SQL | com.microsoft.jdbc.sqlserver.SQLServerDriver | “jdbc:microsoft:sqlserver://IP:1433;  Database=<db\_name>”,”id”,”pwd” |
| MYSQL | org.gjt.mm.mysql.Driver | “jdbc.mysql://<ip>:<port>/<dbname>” |
| DB2 | com.ibm.db2.jdbc.app.DB2Driver | “jdbc:db2://<host 주소>:port/db\_name” |
| Sybase | com.sybase.jdbc.jdbc.SybDriver | “jdbc:sybase.Tds//<ip>:<port>” |
| Informix | com.informix.jdbc.IfxDriver | “jdbc:Informix-sqli://<ip>:<port>/<db\_name>:INFORMIXSERVER” |

**- 각각의 DB 드라이버 연결 형식**

JDK(JDBC-ODBC)

|  |
| --- |
| String url = "jdbc:odbc:orcl";  String user = "userid";  String passwd = "passwd";  try {  Class.forName("jdbc.odbc.JdbcOdbcDriver");  Connection conn = DriverManager.getConnection(url, user, passwd);  }catch(Expception e) { } |

**Oracle Thin Driver**

|  |
| --- |
| String url = "jdbc:oracle:thin:@127.0.0.1:1521:xe";  String user = "userid";  String passwd = "passwd";  try {  Class.forName("oracle.jdbc.driver.OracleDriver");  Connection conn = DriverManager.getConnection(url, user, passwd);  }catch(Expception e) { } |

Oracle OCI Driver

|  |
| --- |
| String url = "jdbc:oracle:oci8:@orcl";  String user = "userid";  String passwd = "passwd";  try {  Class.forName("oracle.jdbc.driver.OracleDriver");  Connection conn = DriverManager.getConnection(url, user, passwd);  }catch(Expception e) { } |

MS-SQL Driver

|  |
| --- |
| String url = "jdbc:Microsoft:sqlserver://127.0.0.1:1433;DatabaseName=orcl";  String user = "userid";  String passwd = "passwd";  try {  Class.forName("com.microsoft.jdbc.sqlserver.SQLServerDriver");  Connection conn = DriverManager.getConnection(url, user, passwd);  }catch(Expception e) { } |

My-SQL Driver

|  |
| --- |
| String url = "jdbc:mysql://127.0.0.1:3306/orcl";  String user = "userid";  String passwd = "passwd";  try {  Class.forName("org.git.mm.mysql.Driver");  Connection conn = DriverManager.getConnection(url, user, passwd);  }catch(Expception e) { } |

**-Statement 객체 생성 및 질의**

드라이버 클래스 로딩과 Connection 객체(DB 연결 정보)를 얻는 방법에 대해서는 확인해 보았다. 이제부터는 원하는 질의문을 데이터베이스에 던질 수 있어야 하는데 이때 사용되는 인터페이스가 Statement interface이다. Connection 객체를 통해서 데이터베이스 서버 정보를 얻었기 때문에 우리가 실행할 SQL문도 데이터베이스 서버에 전달해 주어야 하기 때문에 데이터베이스 서버의 정보를 가지고 생성된 Connection 객체를 통해서 Statement 객체를 얻어야 한다. (Statement 객체는 항상 인수가 없는 Connection 클래스의 createStatement()메소드를 호출함으로써 얻어진다.)

[기본 형식]

|  |
| --- |
| Class.forName(“oracle.jdbc.driver.OracleDriver”);  Connection conn = DriverManager.getConnection(url, user, password);  Statement stmt = conn.createStatement(); |

Statement 객체를 생성하면 Statement 객체의 executeQuery()나 executeUpdate() 메소드를 호출하여 sql문을 실행시킬 수 있다. 이 때 단순한 질의문을 사용할 경우에 좋다.

[DML 실행]

|  |
| --- |
| stmt.executeUpdate(“create table table01(name varchar2(10), age number(3))”);  stmt.executeUpdate(“insert into table01 values(‘scott’,40)”);  stmt.executeUpdate(“insert into table01 values(‘kim’,20)”);  stmt.executeUpdate(“insert into table01 values(‘king’,32)”); |

<주의>

executeUpdate() 메소드의 매개변수의 자료형은 String이고, 문자열, 날짜형 데이터 표현은 더블쿼테이션(“)이 아닌 싱클쿼테이션(‘)으로 표현해야 한다. 데이터는 대소문자를 구분한다.

Statement 메소드

- ResultSet executeQuery()

결과물이 있는 select문을 실행할 때 사용하며 결과물은 Result 객체를 통해서 반환한다.

- int executeUpdate()

insert, update, delete, create, drop 등에 사용하며 결과는 갱신된 행의 수를 반환한다.

- ResultSet execute()

질의문이나 갱신문 모두 사용할 수 있으며, 결과셋을 얻기 위해서는 getResultSet()메소드를 사용해야 한다.

|  |
| --- |
| stmt.execute(“select name from table01”);  ResultSet rs = stmt.getResultSet(); |

**질의 ResultSet 얻기**

executeQuery(), execute() 메소드를 통해서 실행한 결과를 ResultSet 객체로 반환받으며, 질의명령을 행하면 얻어지는 결과물은 column과 row 형태로 가져오게 된다.

ResultSet을 통해서 얻은 결과물을 실제 자바에서 사용할 수 있게 값을 얻는 메소드 getXXX()[레코드셋에서 해당 레코드의 필드 값을 가져올 수 있다.]라는 식의 메소드를 통해서 이루어진다. 얻은 결과에 대해서 다음 행으로 커서를 이동하기 위해서는 next()라는 메소드를 통해서 이동하면 되고 단방향으로 행 이동 가능하다. JDBC 2.0이상에서는 양방향 스크롤이 가능하지만 Vendor마다 달라서 지원이 안되는 경우도 있다. next() 메소드의 리턴타입은 boolean으로서 true가 리턴되면 결과행이 존재하고, false가 리턴되면 더 이상 결과 행이 없다는 뜻이다.

|  |
| --- |
| ResultSet rs = stmt.executeQuery(“Select name, age from table01”); //질의문을 수행하여 결과물을 얻음  while(rs.next()) { //결과물의 수를 알 수 없기 때문에 while문을 사용 – 다음 커서로 이동  String n = rs.getString(“name”); //name 컬럼의 값을 얻어 냄  int a = rs.getInt(“age”); //age 컬럼의 값을 얻어 냄.  System.out.println(“이름 : ” + n + “ 나이 : ”+a);  } |

위의 예제에서 값을 얻는 과정을 보면, 결과물의 값이 문자열이면 rs.getString(“컬럼명”)형식으로 값을 얻었고, 숫자이면 rs.getInt(“컬럼명”) 형식으로 값을 얻는다. 다르게 사용할 수 있는데, name을 얻기 위한 문장을 rs.getString(1)와 같이, age를 얻기 위한 문장은 rs.getInt(2)와 같이 사용할 수 있다.

|  |
| --- |
| 체크 사항  - rs.next()를 맨 처음 한번 반드시 실행하여야 한다. 결과를 얻은 객체의 가장 처음 로우에 커서를 옮기는 작업으로 생각하면 된다. 만약 결과가 없다면 false를 리턴하게 된다.  - ResultSet 객체에 next()를 호출함으로써 해당 객체는 getXXX() 메소드를 호출할 준비를 하게 된다.  - getInt(), getString(), getDate() 등 여러 가지 데이터 형에 대한 메소드가 존재한다.  - 자바의 인덱스의 시작은 0부터 이지만, 데이터베이스는 1부터 시작한다. |

**작업 종료 및 객체 반환**

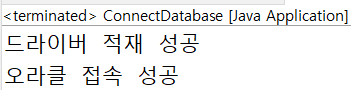
[형식]

|  |
| --- |
| try{  rs.close();  stmt.close();  conn.close()  }catch(Exception e){} |

프로그램 종료되면 자동으로 close 되지만, 명시적으로 close()를 호출함으로써 각각의 객체가 가지고 있는 자원을 반환하게 하여 안정적인 프로그램이 될 수 있게 작업을 해야 할 것이다.

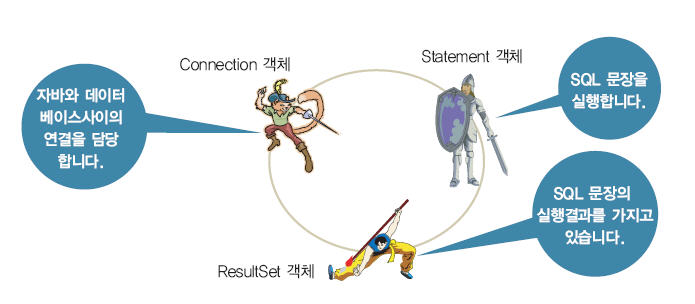
**[예제]**

|  |
| --- |
| ConnectDatabase.java |
| import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.SQLException;  public class ConnectDatabase {  public static Connection makeConnection() {  String **url =** **"jdbc:oracle:thin:@localhost:1521:xe";** // jdbc:mysql://localhost:3306/book\_db  String **id =** **"hr";** // root  String **password = "1234";** // 1234  **Connection con = null;**  try {  **Class.forName("oracle.jdbc.driver.OracleDriver");** // com.mysql.jdbc.Driver  System.out.println("드라이버 적재 성공");  **con = DriverManager.getConnection(url, id, password);**  System.out.println("오라클 접속 성공");  } catch (ClassNotFoundException e) {  System.out.println("드라이버를 찾을 수 없습니다.");  } catch (SQLException e) {  System.out.println("연결에 실패하였습니다.");  }  **return con;**  }  public static void main(String arg[]) throws SQLException {  Connection con = makeConnection();  }  } |



**5) SQL 문장 실행**

데이터베이스로 연결한 후에는 SELECT와 같은 SQL문장들을 실행할 수 있다. 이때 사용되는 것이 Connection, Statement, ResultSet 객체이다.



SELECT 문장을 실행시키려면 다음과 같은 문장을 사용한다.

|  |
| --- |
| **Statement s = con.createStatement();**  // 문장 객체 생성  String select = "SELECT \* FROM EMPLOYEES ORDER BY SALARY"; // SQL 문장 생성  **ResultSet rs = s.executeQuery(select);** // SQL 문장 실행 |

**6) 결과 집합에서 이동**

다음 단계에서는 결과 집합에서 레코드에 하나씩 접근하여서 작업을 하여야 한다. executeQuery 메소드에 의하여 반환된 ResultSet 객체에는 SELECT문장에 의하여 추출된 모든 레코드가 들어 있다. 하지만 한 번에 하나의 레코드만 접근할 수 있다. 이것을 위하여 커서라는 포인터가 제공되는데 이 커서를 움직이는 다양한 메소드가 제공된다.

결과 집합에서 레코드를 하나씩 처리하는 예

|  |
| --- |
| **while(rs.next())**  **{**  // 현재 레코드를 처리한다.  **}** |

ResultSet의 주요한 메소드

|  |  |
| --- | --- |
| **메소드** | **설명** |
| close() | 결과 집합을 닫는다 |
| last() | 커서를 마지막 레코드로 옮긴다. |
| getRow() | 현재 레코드 번호를 얻는다. |
| next() | 다음 레코드로 이동한다. |
| previous() | 이전 레코드로 이동한다. |
| absolute(int row) | 지정된 row로 커서를 이동한다. |
| isFirst() | 첫 레코드이면 true 반환한다. |
| isLast() | 마지막 레코드이면 true 반환한다. |

**7) 결과 집합 처리**

다음 단계는 레코드에서 컬럼의 값을 추출하는 단계이다. 그때 메소드의 매개변수로 컬럼을 이름으로 접근할 수 있고, 컬럼을 번호로 접근할 수 있다.

|  |
| --- |
| String empno = rs.getString("EMPLOYEE\_ID");  String ename = rs.getString("FIRST\_NAME");  또는  String empno = rs.getString(1);  String ename = rs.getString(2); |

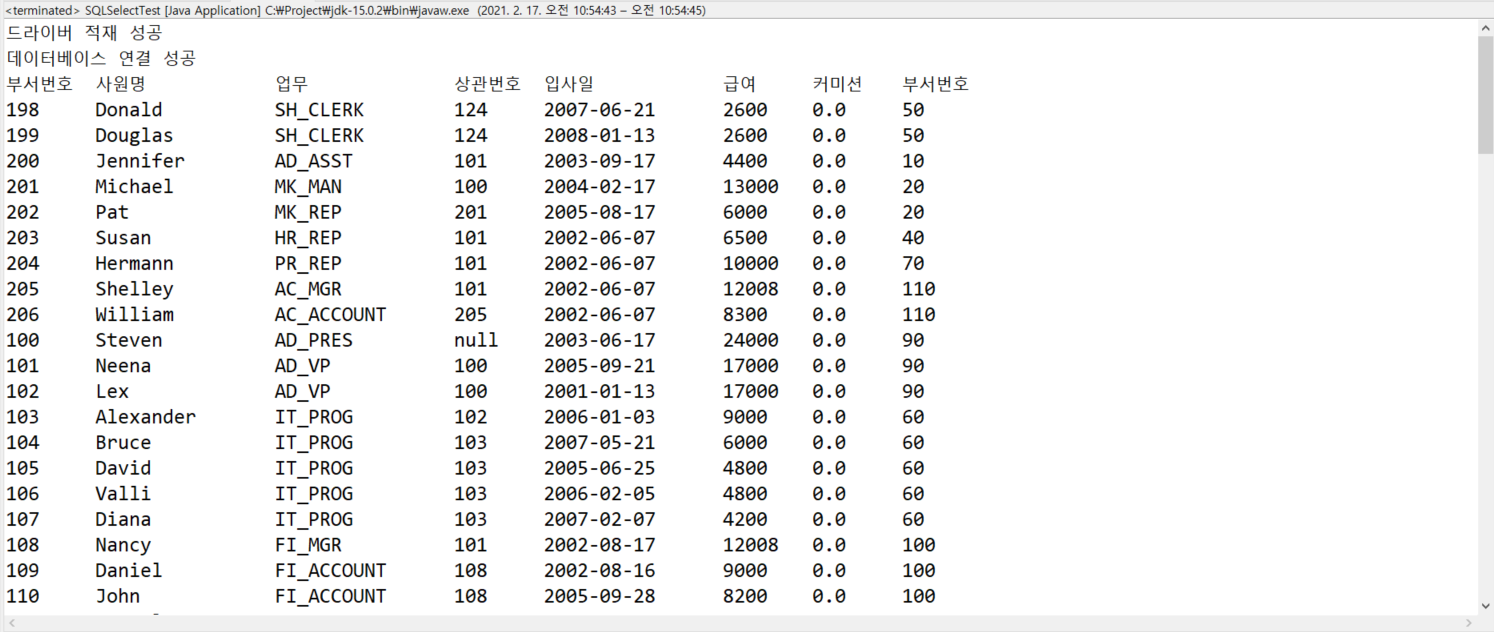
자바에서는 인덱스 번호가 0부터 시작하지만 **SQL에서는 1부터 시작**한다. 레코드에서 컬럼값을 추출하는 메소드는 getXXX()와 같은 형태를 가진다.

|  |  |
| --- | --- |
| **메소드** | **설명** |
| boolean getBoolean(String columnName) | 지정된 컬럼의 값을 boolean 타입으로 반환한다. |
| boolean getBoolean(int columnIndex) | 지정된 컬럼의 값을 boolean 타입으로 반환한다. |
| Date getDate(String columnName) | 지정된 컬럼의 값을 Date 타입으로 반환한다. |
| Date getDate(int columnIndex) | 지정된 컬럼의 값을 Date 타입으로 반환한다. |
| double getDouble(String columnName) | 지정된 컬럼의 값을 double 타입으로 반환한다. |
| double getDouble(int columnIndex) | 지정된 컬럼의 값을 double 타입으로 반환한다. |
| float getFloat(String columnName) | 지정된 컬럼의 값을 float 타입으로 반환한다. |
| float getFloat(int columnIndex) | 지정된 컬럼의 값을 float 타입으로 반환한다. |
| BigDecimal getBigDecimal(String columnName) | 지정된 컬럼의 값을 BigDecimal타입으로 반환한다. |
| BigDecimal getBigDecimal(int columnIndex) | 지정된 컬럼의 값을 BigDecimal타입으로 반환한다. |
| int getInt(String columnName) | 지정된 컬럼의 값을 int 타입으로 반환한다. |
| int getInt(int columnIndex) | 지정된 컬럼의 값을 int 타입으로 반환한다. |
| Long getLong(String columnName) | 지정된 컬럼의 값을 Long 타입으로 반환한다. |
| Long getLong(int columnIndex) | 지정된 컬럼의 값을 Long 타입으로 반환한다. |
| short getShort(String columnName) | 지정된 컬럼의 값을 short 타입으로 반환한다. |
| short getShort(int columnIndex) | 지정된 컬럼의 값을 short 타입으로 반환한다. |
| String getString(String columnName) | 지정된 컬럼의 값을 String 타입으로 반환한다. |
| String getString(int columnIndex) | 지정된 컬럼의 값을 String 타입으로 반환한다. |

접속 후 선택 예제

SQLSelectTest.java

|  |
| --- |
| import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.ResultSet;  import java.sql.SQLException;  import java.sql.Statement;  import java.util.Date;  public class SQLSelectTest {  public static Connection makeConnection() {  **String url = "jdbc:oracle:thin:@localhost:1521:xe";**  **String id = "hr";**  **String password = "1234";**  **Connection con = null;**  try {  **Class.forName("oracle.jdbc.driver.OracleDriver");**  System.out.println("드라이버 적재 성공");  **con = DriverManager.getConnection(url, id, password);**  System.out.println("데이터베이스 연결 성공");  } catch (ClassNotFoundException e) {  System.out.println("드라이버를 찾을 수 없습니다.");  } catch (SQLException e) {  System.out.println("연결에 실패하였습니다.");  }  return con;  }  public static void main(String[] args) **throws SQLException** {  **Connection con = makeConnection();**  **Statement stmt = con.createStatement();**  **ResultSet rs = stmt.executeQuery("SELECT \* FROM EMPLOYEES");**    String empNo;  String eName;  String job;  String mgr;  Date hireDate;  int sal;  double comm;  String deptNo;    System.out.println("부서번호\t사원명\t\t업무\t\t상관번호\t입사일\t\t급여\t커미션\t부서번호");    while (rs.next()) {  empNo = rs.getString("EMPLOYEE\_ID");  eName = rs.getString("FIRST\_NAME");  job = rs.getString("JOB\_ID");  mgr = rs.getString("MANAGER\_ID");  hireDate = rs.getDate("HIRE\_DATE");  sal = rs.getInt("SALARY");  comm = rs.getDouble("COMMISSION\_PCT");  deptNo = rs.getString("DEPARTMENT\_ID");  System.out.println(empNo + "\t" + eName + " \t" + job + " \t" + mgr + "\t" + hireDate + "\t" + sal + "\t" + comm + "\t" + deptNo);  }  }  } |

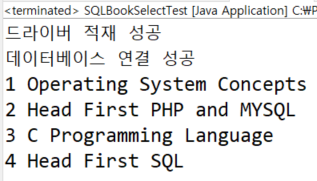


오라클

|  |
| --- |
| CREATE TABLE books (  book\_id number(4),  title varchar2(50),  publisher varchar2(30),  year varchar2(10),  price number(6),  PRIMARY KEY(book\_id)  );  CREATE SEQUENCE book\_id\_seq  START WITH 1  INCREMENT BY 1 ;  INSERT INTO books (book\_id, title, publisher, year, price) VALUES (book\_id\_seq.nextval, 'Operating System Concepts', 'Wiley', '2003',30700);  INSERT INTO books (book\_id, title, publisher, year, price) VALUES (book\_id\_seq.nextval, 'Head First PHP and MYSQL', 'OReilly', '2009', 58000);  INSERT INTO books (book\_id, title, publisher, year, price) VALUES (book\_id\_seq.nextval, 'C Programming Language', 'Prentice-Hall', '1989', 35000);  INSERT INTO books (book\_id, title, publisher, year, price) VALUES (book\_id\_seq.nextval, 'Head First SQL', 'OReilly', '2007', 43000);  commit; |

**[예제]**Oracle DB

|  |
| --- |
| SQLBookSelectTest.java |
| import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.ResultSet;  import java.sql.SQLException;  import java.sql.Statement;  public class SQLBookSelectTest {  public static Connection makeConnection() {  **String url = "jdbc:oracle:thin:@localhost:1521:xe";**  **String id = "hr";**  **String password = "1234";**  **Connection con = null;**  try {  **Class.forName("oracle.jdbc.driver.OracleDriver");**  System.out.println("드라이버 적재 성공");  **con = DriverManager.getConnection(url, id, password);**  System.out.println("데이터베이스 연결 성공");  } catch (ClassNotFoundException e) {  System.out.println("드라이버를 찾을 수 없습니다.");  } catch (SQLException e) {  System.out.println("연결에 실패하였습니다.");  }  **return con;**  }  public static void main(String arg[]) throws SQLException {  Connection con = makeConnection();  Statement stmt = con.createStatement();  ResultSet rs = stmt.executeQuery("SELECT \* FROM books");  while (**rs.next()**) {  결과 집합에서 다음 레코드로 이동한다  int id = **rs.getInt**("book\_id");  String title = **rs.getString**("title");  System.out.println(id + " " + title);  현재 레코드에서 필드의 값을 가져온다.  }  }  } |



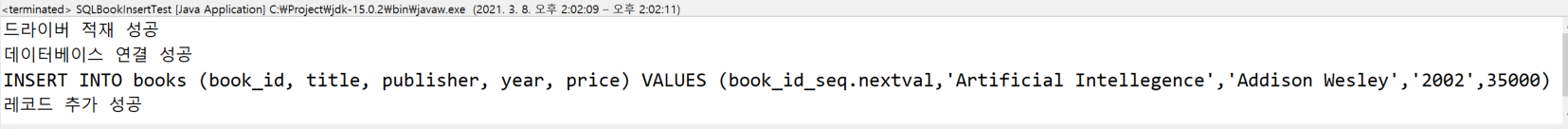
**8) 레코드 추가, 수정, 삭제**

만약 레코드를 추가, 수정, 삭제 시 executeUpdate() 메소드를 사용하여야 한다.

executeUpdate() 메소드는 얼마나 많은 레코드들이 변경되었는지를 반환한다. 이 반환값을 이용하여서 데이터들이 제대로 추가되었는지를 확인할 수 있다.

**[예제]**

|  |
| --- |
| SQLBookInsertTest.java |
| import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.SQLException;  import java.sql.Statement;  public class SQLBookInsertTest {  public static Connection makeConnection() {  String url = "jdbc:oracle:thin:@localhost:1521:xe";  String id = "hr";  String password = "1234";  Connection con = null;  try {  Class.forName("oracle.jdbc.driver.OracleDriver");  System.out.println("드라이버 적재 성공");  con = DriverManager.getConnection(url, id, password);  System.out.println("데이터베이스 연결 성공");  } catch (ClassNotFoundException e) {  System.out.println("드라이버를 찾을 수 없습니다.");  } catch (SQLException e) {  System.out.println("연결에 실패하였습니다.");  }  return con;  }  public static void main(String arg[]) {  addBook("Artificial Intellegence", "Addison Wesley", "2002", 35000);  }  private static void addBook(String title, String publisher, String year, int price) {  Connection con = makeConnection();  try {  Statement stmt = con.createStatement();  String s = "INSERT INTO books (book\_id, title, publisher, year, price) VALUES ";  s += "(" + "book\_id\_seq.nextval" + ",'" + title + "','" + publisher + "','" + year + "'," + price + ")";  System.out.println(s);  int i = stmt.executeUpdate(s);  if (i == 1)  System.out.println("레코드 추가 성공");  else  System.out.println("레코드 추가 실패");  } catch (SQLException e) {  System.out.println(e.getMessage());  System.exit(0);  }  }  } |



**5. 변경가능한 결과 집합**

JDBC 2.0버전부터는 데이터를 변경할 수 있는 또 하나의 방법이 추가로 제공된다. 이것은 변경가능한 결과 집합이라고 불린다. 만약 결과 집합이 변경 가능하다면 결과집합의 데이터를 변경하여서 새로운 데이터를 추가, 삭제할 수 있다. 즉 결과 집합에 변경이 가해지면 자동적으로 데이터베이스가 업데이트된다. 변경 가능한 결과 집합을 생성하기 위해서는 createStatement()메소드를 호출할 때 매개 변수로 ResultSet.CONCUR\_UPDATABLE를 주어야 한다.

|  |
| --- |
| Statement stmt = con.createStatement(ResultSet.TYPE\_SCROLL\_SENSITIVE,  ResultSet.CONCUR\_UPDATABLE); |

첫번째 매개변수는 결과집합이 스크롤 가능함을 나타내는데 즉 결과 집합 안에서 커서를 뒤쪽이나 앞쪽으로 이동할 수 있다. 두 번째 매개변수는 결과 집합이 변경 가능하고 결과 집합에 변경이 가해지면 데이터베이스도 변경된다는 것을 나타낸다.

|  |  |
| --- | --- |
| **메소드** | **설명** |
| void cancelRowUpdates() | 변경을 취소한다 |
| void insertRow() | 결과 집합에 현재 레코드의 내용을 추가한다. |
| void deleteRow() | 결과 집합에 현재 레코드를 삭제한다. |
| void moveToCurrentRow() | 커서를 현재 레코드로 이동한다. |
| void moveToInsertRow() | 커서를 삽입 레코드로 이동한다. |
| void refreshRow() | 현재의 레코드를 가장 최신값으로 갱신한다. |
| void updateRow() | 현재 레코드를 변경한다. |

**1) 레코드를 삭제하기**

결과 집합에서 레코드를 삭제하려면 결과 집합에서 삭제를 원하는 레코드로 이동한 후에 deleteRow() 메소드를 실행하면된다. 이들 문자들은 SQLException을 발생할 수 있으므로 try/catch 블록으로 감싸야 한다.

|  |
| --- |
| Statement stmt = con.createStatement(ResultSet.TYPE\_SCROLL\_SENSITIVE,  ResultSet.CONCUR\_UPDATABLE);  ResultSet rs = stmt.executeQuery("SELECT \* FROM books");  rs.absolute(2);  rs.deleteRow(); |

2) 레코드의 컬럼값 변경

특정한 레코드의 특정 컬럼값을 변경하려면 역시 변경을 원하는 레코드로 이동하여서 변경 메소드 중에서 하나를 호출하고 최종적으로 updateRow()를 호출하면 된다.

|  |
| --- |
| Statement stmt = con.createStatement(ResultSet.TYPE\_SCROLL\_SENSITIVE,  ResultSet.CONCUR\_UPDATABLE);  ResultSet rs = stmt.executeQuery("SELECT \* FROM books");  rs.absolute(3);  rs.updateInt("price", 26000);  rs.updateRow(); |

**3) 레코드를 추가하기**

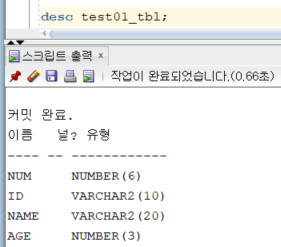
레코드를 추가하기 위해서는 삽입 레코드라고 불리는 특별한 레코드를 사용한다. 먼저 커서를 삽입 레코드로 이동하기 위하여 moveToInsertRow()를 호출한다. 다음에 updateXXX() 메소드를 이용하여서 삽입 레코드의 각 컬럼의 값을 설정한다. 이어서 insertRow() 메소드를 호출하면 삽입 레코드가 결과 집합으로 복사된다. 결과 집합은 다시 데이터베이스로 쓰여진다. 최종적으로 moveToCurrentRow()를 호출하면 이전의 위치로 되돌아간다.

|  |
| --- |
| Statement stmt = con.createStatement(ResultSet.TYPE\_SCROLL\_SENSITIVE,  ResultSet.CONCUR\_UPDATABLE);  ResultSet rs = stmt.executeQuery("SELECT \* FROM books");  // 하나의 레코드를 삽입한다.  rs.moveToInsertRow();  rs.updateString("title", "Introduction to Java");  rs.updateString("publisher", "Wiley");  rs.updateString("year", "2006");  rs.updateInt("price", 29000);  rs.insertRow();  rs.moveToCurrentRow(); |

**[테이블 생성 예제]**

|  |
| --- |
| import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.Statement;  public class DBTable {  public static void main(String args[]) {  Connection conn = null;  Statement stmt = null;  String url = "jdbc:oracle:thin:@127.0.0.1:1521:xe"; // DB Server 정보 셋팅  String user = "hr"; // 사용자 계정  String passwd = "1234"; // 비밀번호  try {  Class.forName("oracle.jdbc.driver.OracleDriver");  conn = DriverManager.getConnection(url, user, passwd);  } catch (ClassNotFoundException ex1) {  System.out.println("드라이버 로딩 에러 : " + ex1.toString());  return;  } catch (Exception ex2) {  System.out.println("DB 접속 실패 : " + ex2.toString());  return;  }  // Sql문 작성  String sql = "create table test01\_tbl(num number(6) , id varchar2(10), name varchar2(20), age number(3))";  try {  System.out.println("DB연결 성공");  stmt = conn.createStatement(); // 질의 객체 생성  stmt.executeUpdate(sql); // 질의 수행  } catch (Exception ex3) {  System.out.println("SQL 문 수행 오류 : " + ex3.toString());  return;  } finally {  try {  stmt.close(); // 객체 반환  conn.close(); // 연결 끊기  } catch (Exception ex4) {  }  }  System.out.println("DB 접속 및 SQL 문 수행 성공!");  }// main end  }// class end |

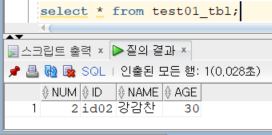
desc test01\_tbl;



**[데이터를 등록하는 예제]**

|  |
| --- |
| import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.Statement;  public class DBdata {  public static void main(String[] args) {  Connection conn = null;  Statement stmt = null;  String url = "jdbc:oracle:thin:@127.0.0.1:1521:xe";  String user = "hr";  String passwd = "1234";  try {  Class.forName("oracle.jdbc.driver.OracleDriver");  } catch (ClassNotFoundException ex1) {  System.out.println("드라이버 로딩 에러 : " + ex1.toString());  return;  } catch (Exception ex2) {  System.out.println("DB 접속 실패 : " + ex2.toString());  return;  }  try {  conn = DriverManager.getConnection(url, user, passwd);  System.out.println("DB연결 성공");  stmt = conn.createStatement();  String sql = "insert into test01\_tbl values( 2, 'id02', '강감찬', 30)";  System.out.println("SQL : " + sql);  int result = stmt.executeUpdate(sql); // 갱신문 수행  System.out.println(result + "개의 Data 입력 성공!!");  } catch (Exception e) {  System.out.println("Data 입력 실패!! : " + e.getMessage());  } finally {  try {  if (stmt != null)  stmt.close(); // 객체 반환  if (conn != null)  conn.close();  } catch (Exception ex) {  }  }  System.out.println("DB 접속 및 SQL 문 수행 성공!");  }  }// class end |

select \* from test01\_tbl;



**[데이터를 검색하는 예제]**

|  |
| --- |
| import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.ResultSet;  import java.sql.Statement;  public class DBSerch {  public static void main(String[] args) {  Connection conn = null;  Statement stmt = null;  String url = "jdbc:oracle:thin:@127.0.0.1:1521:xe";  String user = "hr";  String passwd = "1234";  try {  Class.forName("oracle.jdbc.driver.OracleDriver");  conn = DriverManager.getConnection(url, user, passwd);  } catch (ClassNotFoundException ex1) {  System.out.println("드라이버 로딩 에러 : " + ex1.toString());  return;  } catch (Exception ex2) {  System.out.println("DB 접속 실패 : " + ex2.toString());  return;  }  try {  stmt = conn.createStatement();  String sql = "SELECT \* FROM TEST01\_TBL"; // Select문 작성  ResultSet rs = stmt.executeQuery(sql); // 질의 실행  while (rs.next()) { // 결과셋을 가지고 있는 rs객체를 통해서 결과물 얻기  String s = rs.getInt("num") + "\t";  s += rs.getString("id") + "\t";  s += rs.getString("name") + "\t";  s += rs.getInt("age") + "\n";  System.out.println(s);  }  } catch (Exception e) {  System.out.println("SELECT 실패! : " + e.getMessage());  } finally {  try {  if (stmt != null)  stmt.close();  if (conn != null)  conn.close();  } catch (Exception ex) {  }  }  } // main end  }// class end |



Statement 인터페이스는 SQL문을 전송할 수 있는 객체다. 하지만 좀 더 효율적으로 DBMS에 질의를 할 수 있는 인터페이스를 제공하고 있다.

Statement의 서브 인터페이스로 PreparedStatement를 제공하고 있고, PreparedStatement의 서브 인터페이스로 CallableStatement를 제공하고 있다.

**- PreparedStatement**

**PreparedStement는 SQL문의 구조는 동일하나 조건이 다른 문장을 변수 처리함으로써 항상 SQL문을 동일하게 처리할 수 있는 인터페이스이다.**

PreparedStement로 SQL문을 처리하게 되면 LIBRARY CACHE에 저장된 세 가지 작업을 재사용 함으로써 수행 속도를 좀 더 향상시킬 수 있다.

이해를 돕기 위해 SQL문을 전송했을 경우 오라클은 내부적으로 어떻게 작동하는 보도록 하자.

SQL문 전송하게 되면 오라클은 내부적으로 PARSING → EXECUTE PLAN → FETCH 작업을 한다.

이런 3가지 작업을 한 후에 검색한 결과를 SGA 영역 안에 Data Buffer Cache영역에 Block 단위로 저장하게 된다. SQL문과 PARSING한 결과와 실행계획을 SHARED POOL안에 LIBRARY CACHE에 저장하게 된다.

똑같은 SQL문을 전송하면 LIBRARY CACHE에 저장된 SQL문과 비교하여 SQL문이 동일하다면 파싱한 결과와 EXECUTE PLAN을 그대로 사용하게 되며, 그 결과 또한 디스크에서 I/O로 읽어오지 않지 않고 데이터 버퍼 캐시에 저장된 실제 메모리에서 읽어 오기 때문에 결과를 가져오는 작업 또한 빠르다고 할 수 있다. 똑같은 SQL문이라도 대소문자가 하나라도 틀리거나 SQL문이 다르다면 LIBRARY CACHE에 저장된 3가지 작업을 재 사용할 수 없고 다시 PARSING → EXECUTE PLAN → FETCH 작업을 수행하게 된다.

- PreparedStatement 사용방법

|  |
| --- |
| String sql = “select age from test1 where id=**?**”;  **PreparedStatement** pstmt = con.prepareStatement(sql);  pstmt.setString(**1**, ”syh1011”);  ResultSet rs = pstmt.executeQuery(); |

**PreparedStatement는 SQL문을 작성할 때 컬럼 값을 실제로 지정하지 않고, 변수 처리함으로서 DBMS을 효율적으로 사용한다.**

**PreparedStatement의 SQL문은 SQL문의 구조는 같은데 조건이 수시로 변할 때 조건의 변수처리를 “?” 하는데 이를 바인딩 변수라 한다.**

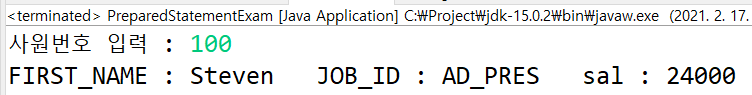
바인딩 변수는 반드시 컬럼 명이 아닌 컬럼 값이 와야 한다는 것이다.

바인딩 변수의 순서는 “?” 의 개수에 의해 결정이 되는데 시작 번호는 1부터 시작하게 된다.

바인딩 변수에 값을 저장하는 메서드는 오라클의 컬럼 타입에 따라 지정해 주면 된다. 전에 ResultSet의 getXXX() 메서드와 유사하게 PreparedStatement 인터페이스에는 바인딩 변수에 값을 저장하는 setXXX() 메서드를 제공하고 있다.

|  |
| --- |
| String sql = “insert into test1 values(id, regDate, age) values(**?**, sysdate, **?**)”;  **PreparedStatement** pstmt = con.prepareStatement(sql);  pstmt.setString(**1**, ”syh1011”);  pstmt.setString(**2**, 25);  int res = pstmt.executeUpdate(); |

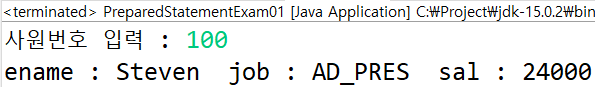
|  |
| --- |
| import java.io.BufferedReader;  import java.io.InputStreamReader;  import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.ResultSet;  import java.sql.Statement;  public class StatementExam {  public static void main(String[] args) {  Connection conn = null;  Statement stmt = null;  ResultSet rs = null;    String url = "jdbc:oracle:thin:@127.0.0.1:1521:xe";  String user = "hr";  String passwd = "1234";    BufferedReader in\_empno = new BufferedReader(new InputStreamReader(System.in));  try {  Class.forName("oracle.jdbc.driver.OracleDriver");  } catch (ClassNotFoundException ex1) {  System.out.println("드라이버 로딩 에러 : " + ex1.toString());  return;  } catch (Exception ex2) {  System.out.println("DB 접속 실패 : " + ex2.toString());  return;  }    try {  conn = DriverManager.getConnection(url, user, passwd);  String empno; // 사원 번호    System.out.print("사원번호 입력 : ");  empno = in\_empno.readLine();    String sql = "select FIRST\_NAME, JOB\_ID, SALARY from EMPLOYEES where EMPLOYEE\_ID = " **+ empno**;    stmt = conn.createStatement(); // SQL문 전송 객체 생성  rs = stmt.executeQuery(sql); // 질의문 수행  while (rs.next()) {  System.out.println("FIRST\_NAME : " + rs.getString(1) + " JOB\_ID : " + rs.getString(2) + " sal : " + rs.getString(3));  }  } catch (Exception e) {  System.out.println(" 실패!! : " + e.getMessage());  } finally {  try {  if (stmt != null)  stmt.close();  if (conn != null)  conn.close();  } catch (Exception ex) {  }  }  }  }// class end |



|  |
| --- |
| conn system/manager  select sql\_text, sharable\_mem, executions  from v$sqlarea  where sql\_text like 'select ename%'; |

각각의 변수들로 인해 다른 문장으로 구분되어 각각 따로따로 한 번씩 (execute 값) 실행되었음을 알 수 있다.

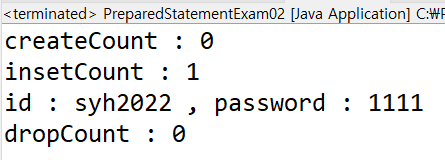
|  |
| --- |
| import java.io.BufferedReader;  import java.io.InputStreamReader;  import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.PreparedStatement;  import java.sql.ResultSet;  public class PreparedStatementExam01 {  public static void main(String[] args) {  Connection conn = null;  PreparedStatement pstmt = null;  ResultSet rs = null;    String url = "jdbc:oracle:thin:@127.0.0.1:1521:xe";  String user = "hr";  String passwd = "1234";    BufferedReader in\_empno = new BufferedReader(new InputStreamReader(System.in));  try {  Class.forName("oracle.jdbc.driver.OracleDriver");  } catch (ClassNotFoundException ex1) {  System.out.println("드라이버 로딩 에러 : " + ex1.toString());  return;  } catch (Exception ex2) {  System.out.println("DB 접속 실패 : " + ex2.toString());  return;  }  try {  conn = DriverManager.getConnection(url, user, passwd);  String empno; // 사원번호  System.out.print("사원번호 입력 : ");  empno = in\_empno.readLine();    String sql = "select FIRST\_NAME, JOB\_ID, SALARY from EMPLOYEES where EMPLOYEE\_ID = **?**";    pstmt = conn.prepareStatement(sql); // SQL문 전송 객체 생성  **pstmt.setString(1, empno);**  rs = pstmt.executeQuery(); // 질의문 수행    while (rs.next()) {  System.out.println("ename : " + rs.getString(1) + " job : " + rs.getString(2) + " sal : " + rs.getString(3));  }  } catch (Exception e) {  System.out.println(" 실패!! : " + e.getMessage());  } finally {  try {  if (pstmt != null)  pstmt.close();  if (conn != null)  conn.close();  } catch (Exception ex) {  }  }  }  }// class end |



|  |
| --- |
| conn system/manager  select sql\_text, sharable\_mem, executions  from v$sqlarea  where sql\_text like 'select ename%'; |

맨 마지막 문장으로 PreparedStatement의 변수는 바인드 변수로 처리됨으로써 같은 문장으로 인정되어 하나의 문장으로 두 번 실행되었음을 알 수 있다.

|  |
| --- |
| import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.PreparedStatement;  import java.sql.ResultSet;  import java.sql.SQLException;  public class PreparedStatementExam02 {  static {  try {  Class.forName("oracle.jdbc.driver.OracleDriver");  } catch (ClassNotFoundException cnfe) {  cnfe.printStackTrace();  }  }  public static void main(String arg[]) {  Connection con = null;  PreparedStatement pstmt = null;  ResultSet rs = null;  try {  con = DriverManager.getConnection("jdbc:oracle:thin:@127.0.0.1:1521:xe", "hr", "1234");    String **sql = "create table syh1012(id varchar2(10), password varchar2(10))";**  pstmt = con.prepareStatement(sql);  int updateCount = pstmt.executeUpdate();  System.out.println("createCount : " + updateCount);  **sql = "insert into syh1012 values(?, ?)";**  pstmt = con.prepareStatement(sql);  pstmt.setString(**1**, "syh2022");  pstmt.setString(**2**, "1111");  updateCount = pstmt.executeUpdate();  System.out.println("insetCount : " + updateCount);  **sql = "select \* from syh1012";**  pstmt = con.prepareStatement(sql);  rs = pstmt.executeQuery();  while (rs.next()) {  System.out.print("id : " + rs.getString(1));  System.out.println(" , password : " + rs.getString(2));  }  **sql = "drop table syh1012";**  pstmt = con.prepareStatement(sql);  updateCount = pstmt.executeUpdate();  System.out.println("dropCount : " + updateCount);  } catch (SQLException sqle) {  sqle.printStackTrace();  } finally {  try {  if (rs != null)  rs.close();  if (pstmt != null)  pstmt.close();  if (con != null)  con.close();  } catch (SQLException sqle) {  }  }  }  } |



**- CallableStatement 기능과 사용방법**

**CallableStatement는 DBMS의 저장 프로시저(Stored Procedure)를 호출할 수 있는 인터페이스이다. (PL/SQL[프로시저, 함수]를 Java언어에서 실행시키기 위해 사용되는 인터페이스이다.)**

저장 프로시저란 파라미터를 받을 수 있고, 다른 애플리케이션이나 PL/SQL 루틴에서 호출할 수 있는 이름을 가진 PL/SQL 블록이다. 즉, SQL문을 프로그램화 시켜 함수화시킨 스크립트 언어이다.

SQL문을 프로그램화 시켜 조건문, 반복문, 변수처리 등을 사용하기 때문에 일괄처리 및 조건에 따라 달라지는 SQL문을 작성할 때는 유리하다.

|  |
| --- |
| PL/SQL 객체를 사용하면 다음과 같은 장점을 얻을 수 있다.  - SQL문을 DBMS가 관리함으로 자바 코드가 간결해 Application의 부담이 덜어진다.  - 일반적인 방식은 SQL문을 DBMS에서 Network를 통해 전달한 후 DBMS가 SQL문을 받아서 분석, 컴파일, 처리하지만 PL/SQL 객체를 사용하게 되면 객체를 생성할 때 분석, 컴파일이 되므로 운영중에 SQL문 분석, 컴파일하는 시간을 절약 할 수 있게 된다.  - 쿼리문을 데이터베이스에 저장하고 함수처럼 호출해 사용하므로 네트워크로 오고 가는 쿼리문을 없애므로 Network 트래픽을 감소 시킬 수 있다.  - PreparedStatement의 장점을 가질 수 있다. 즉 Application에서 처리하는 번수에 대해 DBMS가 바인드 변수로 인식해 동일한 문장으로 처리함으로 SQL문 분석, 컴파일하는 횟수를 줄 일 수 있다. |

먼저 실행된 프로시저를 변경하거나 재생성하는 옵션

- 프로시저 예

|  |
| --- |
| [형식]  create [or replace] procedure 프로시저명  (매개변수 in datatype, 매개변수 out datatype...)  is  begin  실행문  end;  / |

Oracle SQL Developer에서 EMP1 테이블을 생성한다.

|  |
| --- |
| CREATE TABLE EMP1  AS  SELECT EMPLOYEE\_ID, FIRST\_NAME, DEPARTMENT\_ID, SALARY  FROM EMPLOYEES; |

|  |
| --- |
| --부서별로 SALARY 인상. 부서코드다 10이면 10% 인상. 20이면 20%인상 나머지는 동결하는 SQL문을 작성한다.  create or replace procedure pro01\_inmode  (v\_deptno in EMP1. DEPARTMENT\_ID%TYPE)  is  begin  update emp1 set SALARY = decode(v\_deptno, 10, SALARY \*1.1, 20, SALARY \*1.2, SALARY)  where DEPARTMENT\_ID=v\_deptno;  end pro01\_inmode;  /  --drop procedure pro01\_inmode  --값 확인  select EMPLOYEE\_ID, FIRST\_NAME, DEPARTMENT\_ID, SALARY  from EMP1  where DEPARTMENT\_ID=10;    --생성한 프로시저 실행. in mode이면 전달해야 할 값을 직접 명시한다.  exec pro01\_inmode(10);  --변경된 값 확인  select EMPLOYEE\_ID, FIRST\_NAME, DEPARTMENT\_ID, SALARY  from EMP1  where DEPARTMENT\_ID=10; |

이런 Stored Procedure 호출을 가능 하게해 줄 수 있는 인터페이스가 CallableStatement이다.

**CallableStatement는 Connection 인터페이스의 prepareCall(~)를 사용하면 된다.**

|  |
| --- |
| CallableStatement cstmt = con.prepareCall("{call 프로시저명(?, ?)}"); |

prepareCall(String procedure)의 프로시저는 2가지 형태를 가지고 있다.

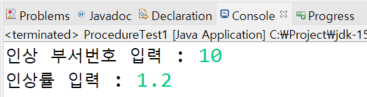
|  |
| --- |
| 1. {?= call <procedure-name>[<arg1>,<arg2>, ...]}  1번은 <arg1>,<arg2>,..은 PreparedStatement 처럼 바인딩 변수로 처리 하면 되고,  ?은 registerOutParameter(~)계열의 메서드를 사용하면 된다.  2. {call <procedure-name>[<arg1>,<arg2>, ...]}  2번은 <arg1>,<arg2>,..은 PreparedStatement 처럼 바인딩 변수로 처리 한다. |

|  |
| --- |
| create or replace procedure myproc  (v\_ename in varchar2, v\_sal out number)  is  begin  select SALARY into v\_sal  from EMP1  where FIRST\_NAME= v\_ename;  end myproc;  / |

|  |
| --- |
| CallableStatement cstmt; // CallableStatement 선언  cstmt = conn.prepareCall("{call myproc(?,?)}"); //저장프로시저 호출  cstmt.setString(1, "scott"); //in형 변수에 값을 지정해 줌.  /\* in으로 프로시저에 들어가는 변수는 변수의 datatype에 따라 setString, setInt, setDate등을 값을 입력해 줘야 한다. \*/  cstmt.registerOutParameter(2, Types.INTEGER); //out형 변수에 값을 받을 datatype을 지정.  /\* out을 프로시저를 실행한 결과물은 먼저 전달받을 변수의 datatype을 먼저 registerOutParameter 에 의해 지정받고 실행한 결과를 마지막 라인처럼 받아야 한다.\*/  cstmt.executeQuery(); //해당 프로시져를 실행  int sal = cstmt.getInt(2); //실행한 후 out에 의한 결과를 받음. |

|  |
| --- |
| create or replace procedure proc01  (v\_deptno in number, v\_increase\_rate in number)  is  begin  update emp1 set SALARY= SALARY\*v\_increase\_rate  where DEPARTMENT\_ID = v\_deptno;  end proc01;  / |

|  |
| --- |
| import java.io.BufferedReader;  import java.io.IOException;  import java.io.InputStreamReader;  import java.sql.CallableStatement;  import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.SQLException;  public class ProcedureTest1 {  static {  try {  Class.forName("oracle.jdbc.driver.OracleDriver");  } catch (ClassNotFoundException cnfe) {  cnfe.printStackTrace();  }  }  public static void main(String arg[]) {  Connection con = null;  CallableStatement cstmt = null;  String deptno;  String rate;  try {  con = DriverManager.getConnection("jdbc:oracle:thin:@127.0.0.1:1521:xe", "hr", "1234");  cstmt = con.prepareCall("{call proc01(?, ?)}");  // 실행할 프로시저 명시. ?는 매개변수이 개수만큼 입력  BufferedReader in\_value = new BufferedReader(new InputStreamReader(System.in));  System.out.print("인상 부서번호 입력 : ");  deptno = in\_value.readLine();  System.out.print("인상률 입력 : ");  rate = in\_value.readLine();  cstmt.setInt(1, Integer.parseInt(deptno));  cstmt.setFloat(2, Float.parseFloat(rate));  cstmt.executeUpdate();  } catch (SQLException ee) {  System.out.println("Connection Error");  ee.printStackTrace();  } catch (IOException ie) {  } finally {  try {  if (cstmt != null)  cstmt.close();  if (con != null)  con.close();  } catch (Exception e) {  }  }  }  } |

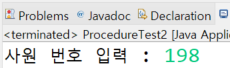


Oracle SQL Developer에서 결과 확인한다.

사원번호를 입력해서 그 사원이 속한 부서의 평균 SALARY를 구한 후 평균보다 높으면 10%인상을 평균보다 낮으면 20%인상을 하는 프로그램을 만든다.

|  |
| --- |
| create or replace procedure proc02  (v\_empno in number, msg out varchar2)  is  v\_avg\_sal number;  v\_sal number;  v\_ename varchar2(20);  v\_increase number;  begin  --사원의 이름 및 월급을 받음.  select FIRST\_NAME, SALARY into v\_ename, v\_sal  from EMP1  where EMPLOYEE\_ID= v\_empno;  --사원 부서의 평균 sal를 v\_avg\_sal에 받음  select avg(SALARY) into v\_avg\_sal  from EMP1  where DEPARTMENT\_ID=(select DEPARTMENT\_ID  from EMP1  where EMPLOYEE\_ID=v\_empno);  --부서 평균sal보다 자신의 sal가 많으면 인상률이 1.1이고 적으면 1.2  if v\_sal > v\_avg\_sal then  v\_increase := 1.1;  else  v\_increase := 1.2;  end if;  update EMP1 set SALARY = SALARY\*v\_increase  where EMPLOYEE\_ID = v\_empno;  msg := v\_ename||'사원의 '|| v\_sal || '월급이 ' || v\_increase ||'%월급 인상으로 '|| v\_sal\*v\_increase || '으로 인상';  end proc02;  / |

|  |
| --- |
| import java.io.BufferedReader;  import java.io.IOException;  import java.io.InputStreamReader;  import java.sql.CallableStatement;  import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.SQLException;  import java.sql.Types;  public class ProcedureTest2 {  static {  try {  Class.forName("oracle.jdbc.driver.OracleDriver");  } catch (ClassNotFoundException cnfe) {  cnfe.printStackTrace();  }  }  public static void main(String arg[]) {  Connection con = null;  CallableStatement cstmt = null;  try {  con = DriverManager.getConnection("jdbc:oracle:thin:@127.0.0.1:1521:xe", "hr", "1234");  cstmt = con.prepareCall("{call proc02(?,?)}");  // 실행할 프로시저 명시. ?는 매개변수이 개수만큼 입력  BufferedReader in\_value = new BufferedReader(new InputStreamReader(System.in));  System.out.print("사원 번호 입력 : ");  String v\_empno = in\_value.readLine();  cstmt.setInt(1, Integer.parseInt(v\_empno));  cstmt.registerOutParameter(2, Types.VARCHAR);  cstmt.executeUpdate();  System.out.println(cstmt.getString(2));  } catch (SQLException ee) {  System.out.println("Connection Error");  ee.printStackTrace();  } catch (IOException ie) {  } finally {  try {  if (cstmt != null)  cstmt.close();  if (con != null)  con.close();  } catch (Exception e) {  }  }  }  } |



Oracle SQL Developer에서 결과 확인한다.

트랜잭션이란 하나의 논리적 작업 단위로 수행되는 일련의 작업을 의미(= 일의 최소 단위)한다. 처리방식은 all or nothing(문제가 생기면 전부 되거나 아예 안되도록 한다는 것)이다.

트랜잭션은 하나의 작업 단위의 일들은 전체 작업이 모두 올바르게 수행되거나 또는 전체 작업이 모두 수행되지 않아야 한다.

|  |  |
| --- | --- |
| **특 성** | **설 명** |
| **원자성** | 트랜잭션은 더 이상 분리 될 수 없다. 쪼갤 수 없다는 표현이 적절할 것이다. 더 이상 쪼갤 수 없는 트랜잭션은 모든 것이 성공하거나, 또는 모든 것이 실패해야 한다는 원칙이 적용된다 |
| **일관성** | 하나의 트랜잭션이 완료되면, 완료 후의 데이터는 일관성을 유지해야 한다. 관계형 데이터 베이스에서는 트랜잭션 중 모든 규칙을 적용하여 데이터의 일관성을 유지해야 한다 |
| **격리성** | 두 개 이상의 트랜잭션은 서로 격리되어야 한다. 즉 트랙잭션 A와 트랜잭션 B는 서로 참조 할 수 없으며, 서로를 침해 할 수도 없다 |
| **영속성** | 트랜잭션이 완료된 데이터는 영속적이어야 한다. 즉 변화가 없다는 것을 확신할 수 있어야 한다 |

**Transaction에 사용되는 메소드**

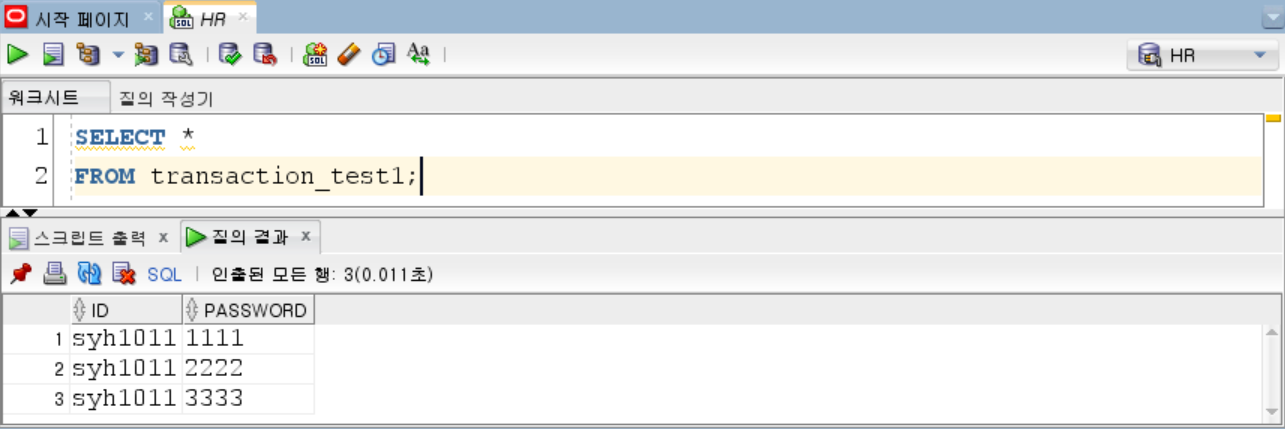
SQL\*PLUS에서도 여러분이 임의의 테이블에 insert를 수행했다고 실제 테이블에 삽입이 되는 것이 아니다. 물론 select문으로 조사해보면 그 결과는 나오지만 프로그램으로 조사해보면 어떤 데이터도 보이지 않게 된다. 테이블에 insert를 하고 commit 명령어를 내려야만 테이블에 삽입된다. 물론 테이블에 삽입하지 않기 위해서는 rollback명령어를 이용할 수 있다.

JDBC도 이와 같이 Connection 인터페이스에서 commit과 rollback을 제어하기 위해서 setAutoCommit()을 이용해서 commit의 true와 false를 세팅할 수 있다. setAutoCommit(false)로 세팅되어 있으면 모든 작업 내용은 트랜잭션되며, commit을 수행하기 전까지는 데이터베이스에 반영되지 않는다. 그리고 commit을 수행할 수 있는 메소드(commit()), rollback을 할 수 있는 메소드(rollback())을 제공하고 있다.

|  |
| --- |
| setAutoCommit(boolean autoCommit) – autoCommit이 true이면 트랜잭션을 시작하지 않겠다는 의미, false 이면 트랜잭션을 시작하겠다는 의미(다시말해 트랜잭션을 가능하게 함)이다.  commit() – setAutoCommit(false)와 commit() 사이에 있는 모든 operation를 수행하겠다는 의미  (setAutoCommit(false)이면 반드시 commit()을 수행해야만 데이터베이스에 반영된다)이다.  그리고 setAutoCommit(true)이면 쿼리의 수행과 동시에 데이터베이스에 반영한다. 그래서 기본값이  setAutoCommit(true)이기 때문에 true로 세팅할 필요는 없다  rollback() - setAutoCommit(false)와 rollback() 사이에 있는 모든 operation를 수행하지 않겠다는 의미(작업을 취소하고 싶을 때 수행하면 된다.)이다. |

|  |
| --- |
| import java.sql.Connection;  import java.sql.DriverManager;  import java.sql.PreparedStatement;  import java.sql.SQLException;  public class TransactionExam {  static {  try {  Class.forName("oracle.jdbc.driver.OracleDriver");  } catch (ClassNotFoundException cnfe) {  cnfe.printStackTrace();  }  }  public static void main(String[] args) {  Connection con = null;  PreparedStatement pstmt = null;  boolean success = false;  String sql;  try {  con = DriverManager.getConnection("jdbc:oracle:thin:@127.0.0.1:1521:xe", "hr", "1234");  sql = "create table transaction\_test1 (id varchar2(10), password varchar2(10))";  pstmt = con.prepareStatement(sql);  pstmt.executeUpdate();  **con.setAutoCommit(false);**  sql = "insert into transaction\_test1 values('syh1011','1111')";  pstmt = con.prepareStatement(sql);  pstmt.executeUpdate();  sql = "insert into transaction\_test1 values('syh1011','2222')";  pstmt = con.prepareStatement(sql);  pstmt.executeUpdate();  sql = "insert into transaction\_test1 values('syh1011','3333')";  pstmt = con.prepareStatement(sql);  pstmt.executeUpdate();  **success = true;**  } catch (Exception e) {  e.printStackTrace();  } finally {  try {  if (success) {  **con.commit();**  } else {  **con.rollback();**  }  if (pstmt != null)  pstmt.close();  if (con != null)  con.close();  } catch (SQLException sqle) {  sqle.printStackTrace();  }  }  }  } |

서로 다른 Connection 객체를 가지고 transaction을 만들면 당연이 작동하지 않는다. 정확히 트랜잭션을 하기 위해서는 한 개의 Connection 객체로 여러 가지 오퍼레이션을 수행해야 한다.



**Properties를 이용한 JDBC 연결과 ResultSetMetaData**

Properties클래스를 이용해서 특정 파일을 읽어온 후 드라이브 로드명, URL, USER, PASSWORD의 정보를 알아낸다. 이렇게 Properties 클래스를 이용하게 되면 DBMS가 바뀌거나 드라이브 로드명, URL, USER, PASSWORD가 변경되더라도 파일의 정보만 변경하고, 실제 프로그램을 수정하여 재컴파일을 할 필요가 없기 때문에 유지 보수에 상당히 좋다고 할 수 있다.

Properites 클래스는 properties파일의 집합을 추상화 클래스이다. 이 클래스는 Stream를 로드하여 저장할 수 있고, 이를 다시 Map 형태로 관리하여 Key를 알면 Key에 대한 Value을 얻어올 수 있는 클래스이다.

C:\jdbc.properties 파일을 아래와 같이 만들어보자.

|  |
| --- |
| driver = oracle.jdbc.driver.OracleDriver  url = jdbc:oracle:thin:@127.0.0.1:1521:xe  user = hr  password = 1234 |

**jdbc.properties 파일을 읽기 위해서는 스트림을 생성해야 한다.**

**스트림을 Properties 클래스에 로드(load())** 시키고, **Properties클래스의 getProperty(String key)메서드**를 이용해서 각각의 값을 얻어 올 수 있다.

|  |
| --- |
| try{  **FileInputStream fis = new FileInputStream("c:\\jdbc.properties");**  Properties pro = new Properties();  **pro.load(fis);**  String driver = pro.**getProperty("driver");**  String url = pro.getProperty("url");  String user = pro.getProperty("user");  String password = pro.getProperty("password");  }catch(IOException ee){  ee.printStackTrace();  } |

Properties 클래스를 이용하여 JDBC를 연결하게 되면 데이터베이스가 바뀌는 경우, 또는 데이터 베이스의 IP가 바뀌는 경우, 사용자가 바뀌는 경우에 jdbc.properties 파일만 수정하여 재컴파일 방지하고, 유지보수에 용이하게 함이다.

|  |
| --- |
| import java.sql.\*;  import java.io.\*;  import java.util.\*;  public class PropertieEx{  public static void main(String[] args){  Connection con=null;  PreparedStatement pstmt=null;  Properties pro=null;  FileInputStream fis=null;  try{  pro = new Properties();  fis = new FileInputStream("c:\\jdbc.properties");  pro.load(fis);  String driver = pro.getProperty("driver");  String url = pro.getProperty("url");  String user = pro.getProperty("user");  String password = pro.getProperty("password");  Class.forName(driver);  con = DriverManager.getConnection(url,user,password);  String sql = "create table properties(id varchar(10))";  pstmt = con.prepareStatement(sql);  pstmt.executeUpdate();  }catch(Exception e){  e.printStackTrace();  }finally{  try{  if(pstmt != null) pstmt.close();  if(con != null) con.close();  }catch(SQLException sqle){  }  }  }  } |

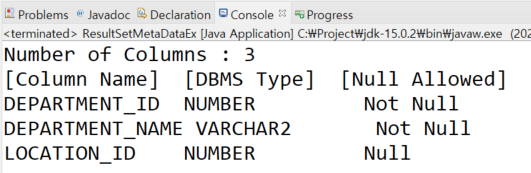
JDBC는 테이블의 컬럼에 대한 메타정보(컬럼의 수, 컬럼의 이름, 컬럼의 데이터 타입 등)을 얻어오기 위해 ResultSetMetaData라는 인터페이스를 제공하고 있다. Table명은 알고 있지만 Table를 구성하는 컬럼명이라든지, 컬럼명의 자료형등을 알기 위해서는 ResultSetMetaData를 이용해야 한다.

ResultSetMetaData는 ResultSet의 구성요소이다. **다시 말해서, SQL의 Table을 구성하는 모든 요소를 알아낼 수 있는 메서드를 제공하고 있다.**

ResultSetMetaData 객체를 생성하는 방법은 아래와 같다.

|  |
| --- |
| rs = pstmt.executeQuery() ;  **ResultSetMetaData rsmd = rs.getMetaData()** ; |

|  |
| --- |
| import java.sql.\*;  public class ResultSetMetaDataEx{  static{  try{  Class.forName("oracle.jdbc.driver.OracleDriver");  }catch(ClassNotFoundException cnfe){  cnfe.printStackTrace();  }  }  public static void main(String[] args){  Connection con=null;  PreparedStatement pstmt=null;  ResultSet rs=null;  String \_nullable=null;  try {  con = DriverManager.getConnection(  "jdbc:oracle:thin:@127.0.0.1:1521:xe", "hr", "1234");  String sql = " select department\_id, department\_name, location\_id from departments";  pstmt = con.prepareStatement(sql);  rs = pstmt.executeQuery() ;  ResultSetMetaData rsmd = rs.getMetaData() ;  int numberColumn = rsmd.getColumnCount() ;  System.out.println("Number of Columns : " + numberColumn) ;  System.out.println("[Column Name] [DBMS Type] [Null Allowed]");  for( int i=1 ; i<=numberColumn ; i++) {  String columnName = rsmd.getColumnName(i) ;  String dbmsType = rsmd.getColumnTypeName(i) ;  int isNull = rsmd.isNullable(i) ;  //null 허용 여부를 반환하는데 0일때는 not null, 1이면 null을 의미  if(isNull == 0)  \_nullable = "Not Null";  else if(isNull == 1)  \_nullable = "Null";  else  \_nullable = "UnKnown";  System.out.printf("%-14s %-14s %s %n",  columnName, dbmsType, \_nullable);  }  }catch(Exception e){  e.printStackTrace() ;  }finally{  try {  if(rs != null) rs.close();  if(pstmt != null) pstmt.close();  if(con != null) con.close();  }catch(SQLException se){  }  }  }  } |

****