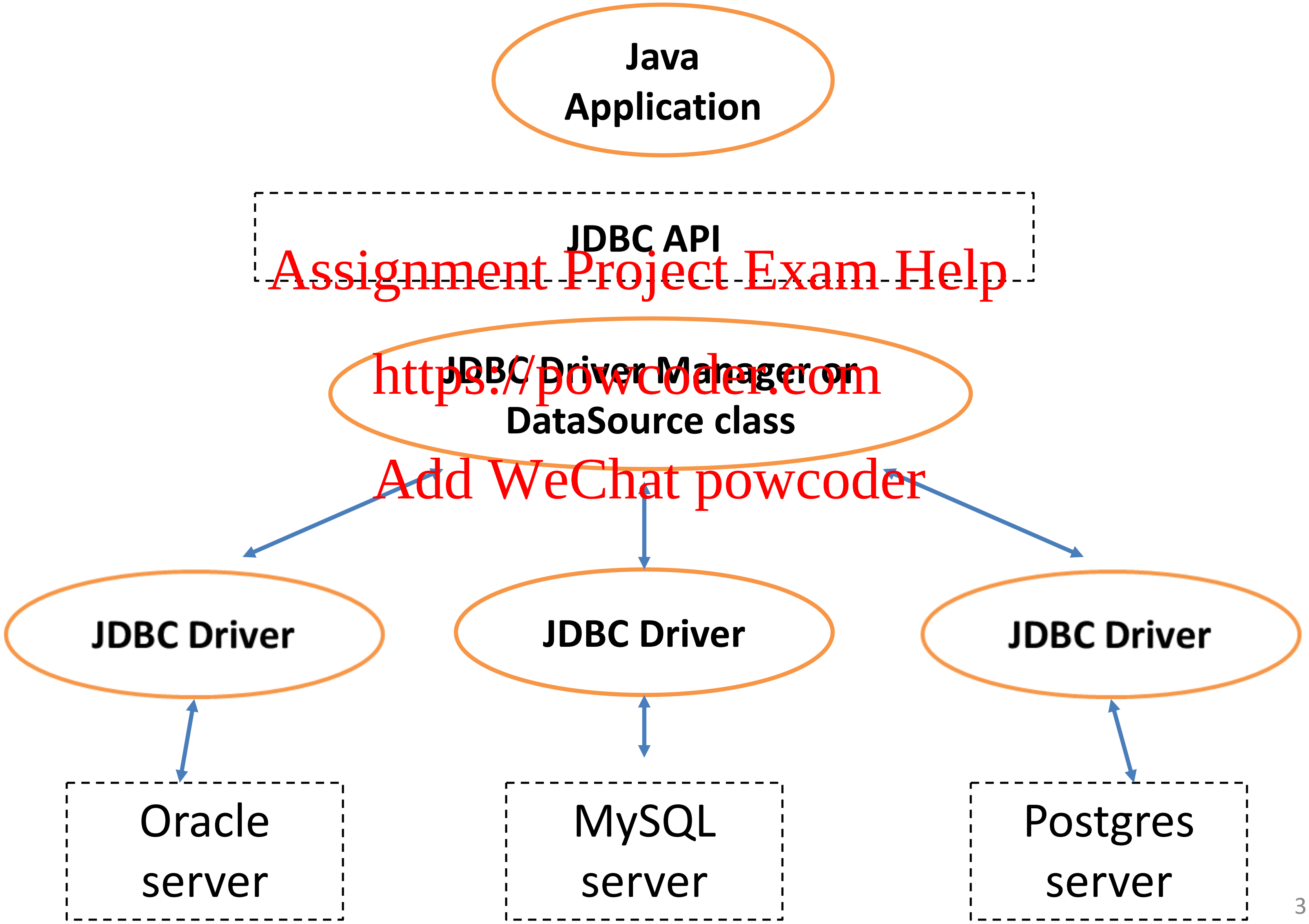
CS157A: Introduction to Database Management Systems

JDBC
(**J**ava **D**ata**B**ase **C**onnectivity)
Suneuy Kim

JDBC

- http://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/
- JDBC API : The industry standard for database-independent connectivity between Java and a SQL database
  - Establish a connection with a database or access any tabular data source
  - Send SQL statements
  - Process the results

JDBC Architecture



Establishing a Connection

Typically, a JDBC application connects to a target data source using one of two classes:

- DriverManager and DataSource
- My examples use the DriverManager class instead of the DataSource because it is easier to use and the examples do not require the features of the DataSource class.

Processing SQL Statements with JDBC

- `Statement`

To submit the SQL statements to the database.

- `ResultSet`

Holds results of SQL statements. It acts as an iterator to allow you to iterate over its data.

- `SQLException`

Handles any errors that occur in a database application.

Creating JDBC Application

- Import the packages

  `e.g.) import java.sql.*`
- Register the JDBC driver (automatically done since JDBC 4.0)

  `Class.forName("com.mysql.jdbc.Driver");`
- Open a connection

  `Connection conn = DriverManager.getConnection();`
- Create a Statement

  `Statement  statementObject = conn.createStatement();`
- Execute a query

  `statementObject.execute(sq);`
- Extract data from result set .

  Appropriate `ResultSet.getXXX()` method
- Clean up the environment

  `conn.close();`

Source: JDBCExample.java

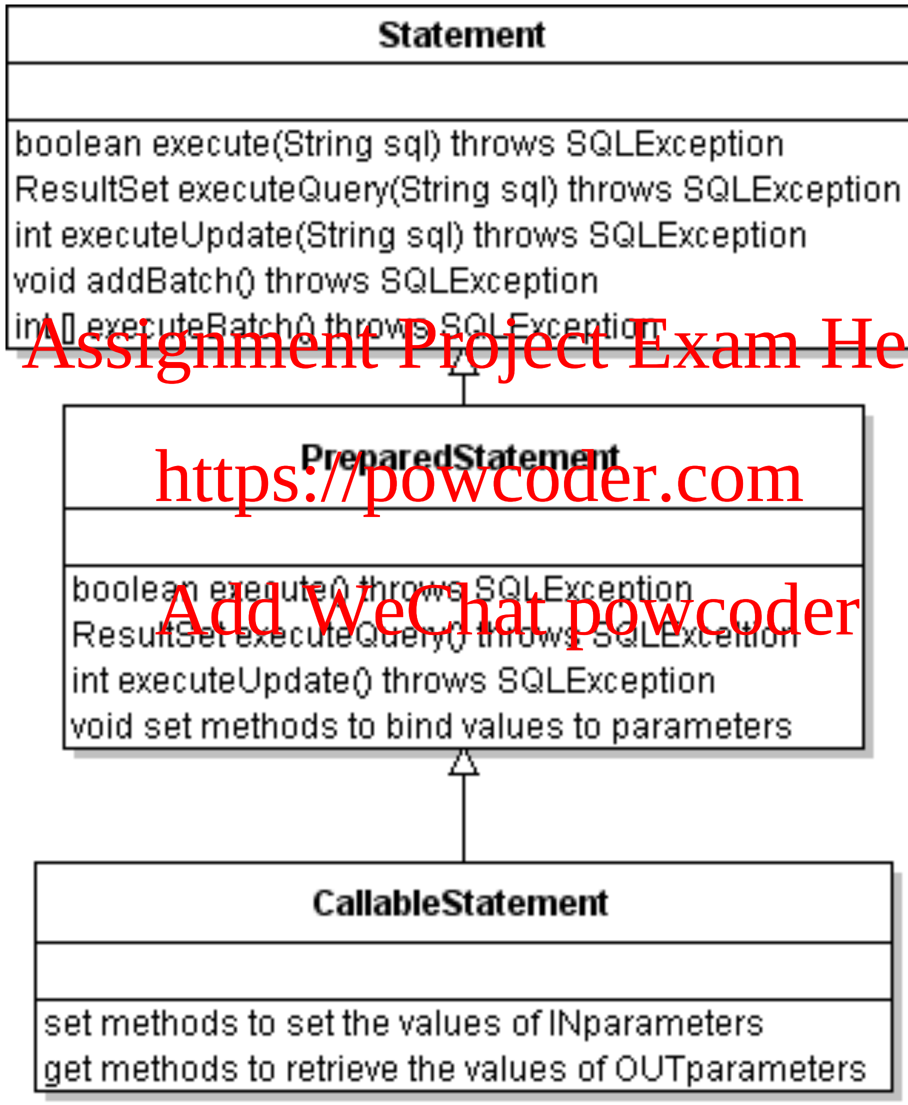
Example Source Codes

- DriverManagerTester.java
- JDBCExample.java : Use this as a template to create your JDBC applications.

Statements

- Statement: Used to implement simple SQL statements with no parameters.
- PreparedStatement: Used for precompiling SQL statements that might contain parameters in a form of ?.
- CallableStatement: Used to execute stored procedures that may contain both input and output parameters.

JDBC Statements


Statement
boolean execute(String sql) throws SQLException
ResultSet executeQuery(String sql) throws SQLException
int executeUpdate(String sql) throws SQLException
void addBatch() throws SQLException
int[] executeBatch() throws SQLException
PreparedStatement
boolean execute() throws SQLException
ResultSet executeQuery() throws SQLException
int executeUpdate() throws SQLException
void set methods to bind values to parameters
CallableStatement
set methods to set the values of INparameters
get methods to retrieve the values of OUTparameters

Statement

- To execute a simple SQL statement with no parameters.

```
Statement stmt = conn.createStatement();
String sql = "INSERT INTO BOOK VALUES ('B', 'A', 10)";
stmt.executeUpdate(sql);
```

- Execute Methods

- Create, insert, update or delete: `stmt.executeUpdate(sql);`
- Select query that returns one ResultSet: `stmt.executeQuery(sql);`

- Query that might returns multiple ResultSets:

```
  stmt.execute(sql);
  ReseultSet rs = stmt.getResultSet();
```

- See pp. 30 to see a SQL statement with parameters.

Create/Drop Database

- JDBC - Create Database

```
String sql =
          "CREATE DATABASE STUDENTS";
```

- JDBC - Drop Database

```
String sql =
          "DROP DATABASE STUDENTS";
```

Create/Drop Tables

- JDBC - Create Tables

```
String sql = "CREATE TABLE REGISTRATION "
           + "(id INTEGER not NULL, "
           + " first VARCHAR(255), "
           + " last VARCHAR(255), "
           + " age INTEGER, "
           + " PRIMARY KEY ( id ))";
```

- JDBC - Drop Tables

```
String sql = "DROP TABLE REGISTRATION ";
```

Modification

- JDBC - Insert Records

```
String sql="INSERT INTO Registration "
+ "VALUES (100, 'Zara', 'Ali', 18)";
```

- JDBC - Update Records

```
String sql = "UPDATE Registration " +
"SET age = 30 WHERE id in (100, 101)";
```

- JDBC - Delete Records

```
String sql = "DELETE FROM Registration "
+ "WHERE id = 101";
```

Select-From-Where clause

```
String sql =
   "SELECT id, first, last, age
    FROM Registration";
String sql =
  "SELECT id, first, last, age
   FROM Registration" +
" WHERE id >= 101 ";
```

Like/Order by

JDBC - Like Clause

```
sql = "SELECT id, first, last, age
FROM Registration" + " WHERE first
LIKE '%za%' ";
```

JDBC –Order by Clause

```
String sql = "SELECT id, first, last,
age FROM Registration" + " ORDER BY
first ASC";
```

A SQL statement that returns multiple result sets

```
String createProcedure = "CREATE PROCEDURE doSomething() "+
                                   "BEGIN SELECT * FROM Students ; "+
                                  "SELECT * FROM Students where age < 20 ; END" ;
statement.executeUpdate(createProcedure);

boolean hasResults = statement.execute("{CALL doSomething()}");
 while (hasResults)
 {      ResultSet rs = statement.getResultSet();
        while (rs.next())
        {
        System.out.print("id:" + rs.getInt("id"));
        System.out.print("name:" + rs.getString("name"));
        System.out.print("age:" + rs.getInt("age"));
        }
        hasResults = statement.getMoreResults();
}
```

Batch Update
(from JDBCStatementExample.java)

```
conn.setAutoCommit(false);

statement = conn.createStatement();
statement.addBatch("INSERT INTO Students " +
"VALUES (495, 'Robert Cliff', 22)");
statement.addBatch("INSERT INTO Students " +
"VALUES (333, 'Toni Smith', 27)");
statement.addBatch("INSERT INTO Students " +
"VALUES (555, 'Robert E.Laskey', 25)");

int [] updateCounts = statement.executeBatch();
conn.commit();
```

```
int [] updateCounts =
statement.executeBatch();
```

- Each executed statement returns a update count indicating how many rows are affected by this statement.
- In previous example, the executeBatch returns an array containing three 1s.
- You can use executeBatch if a statement return a update count
  - insert, update and delete: n >= 0
  - create and drop: 0
- Otherwise, executeBatch can't be used (e.g. select)

Example: JDBC Statement

- JDBCStatementExample.java

JDBC PreparedStatement

- This extended statement gives you the flexibility of supplying arguments dynamically.
- All parameters in JDBC are represented by the parameter symbol **?**.
- Each parameter marker is referred to by its ordinal position, starting at 1.
- The `setXXX()` methods bind values to the parameters, where represents the Java data type of the value

JDBC PreparedStatement

```
String SQL =
   "Update Employees SET age = ? WHERE id = ?";
PreparedStatement pstmt =
                  conn.prepareStatement(SQL);
pstmt.setInt(1, 35);
pstmt.setInt(2, 111);
pstmt.executeUpdate();

pstmt.setInt(1, 40);
pstmt.setInt(2, 222);
pstmt.executeUpdate();

Note: execute()/executeQuery()as needed.
```

JDBC PreparedStatement: Example

JDBCPreparedStatementExample.java

JDBC CallableStatement: Steps

1. Prepare the callable statement by using Connection.prepareCall().
2. Register the output parameters (if any exist)
3. Set the input parameters (if any exist)
4. Execute the CallableStatement, and retrieve any result sets or output parameters.

StoredProcedure with IN Parameter

```
DROP PROCEDURE IF EXISTS getFacultyByName;
DELIMITER //
CREATE PROCEDURE getFacultyByName(IN facultyName
VARCHAR(50))
BEGIN
SELECT *
FROM Faculty
WHERE name=facultyName;
END//
DELIMITER ;
---------------------------------------
call getFacultyByName('Dennis Chien');
+-----+--------------+------+
| id  | name         | age  |
+-----+--------------+------+
| 848 | Dennis Chien |   52 |
+-----+--------------+------+
```

To call a stored procedure with IN parameter

```
String sql = "{call getFacultyByName(?)}";
CallableStatement cstmt = conn.prepareCall(sql);
cstmt.setString(1,"James Sobrier");
boolean hasResult = cstmt.execute();
```

Stored Procedure with OUT parameter

```
DROP PROCEDURE IF EXISTS countByAge;
DELIMITER //
CREATE PROCEDURE countByAge(IN retirementAge INT,
                            OUT total INT)
BEGIN
  SELECT count(*) INTO total
  FROM Faculty
  WHERE retirementAge < age;
END//

DELIMITER ;
----------------------------------------------------
CALL countByAge(50, @result);
SELECT @result;
```

To call a stored procedure with OUT parameter

```
CallableStatement cs =
  conn.prepareCall("{CALL countByAge(?, ?)}");
cs.setInt(1,50);
cs.registerOutParameter(2, Types.INTEGER);
boolean hasResult = cs.execute(); // false
System.out.println(cs.getInt(2));//by index
System.out.println(cs.getInt("total"));//  by
name
```

Stored Procedure returning a relation without OUT parameter

```
DROP PROCEDURE IF EXISTS countByAge2;
DELIMITER //
CREATE PROCEDURE countByAge2(IN retirementAge
INT)
BEGIN
SELECT count(*)
FROM Faculty
WHERE retirementAge < age;
END//
DELIMITER ;
----------------------------------------
CALL countByAge2(50);
```

To call a stored procedure returning a relation without OUT paramter

```
CallableStatement cs =
   conn.prepareCall("{CALL countByAge2(?)}");
cs.setInt(1,50);
boolean hasResult2= cs.execute();
if (hasResult2)
{   rs = cs.getResultSet(); rs.next();
    System.out.println(rs.getInt(1));
}
```

To get multiple ResultSets from a stored procedure

```
CallableStatement cs = …
boolean hasResults = cs.execute();
while (hasResults)
{   System.out.println("Result Set:");
    ResultSet rs = cs.getResultSet();
    printResultSet(rs);
    rs.close();
    hasResults = cs.getMoreResults();
}
```

JDBC CallableStatement: Example

- JDBCCallableStatementExample.java
- MultipleResultSets.java

ResultSet

- Represents a table of data returned by executing query statement.
- A ResultSet maintains a cursor.
- Initially the cursor is positioned before the first row.
- The navigational methods move the cursor in ResultSet. It returns false if there is no rows in the ResultSet. `while(rs.next()){ }`

ResultSet

```
try
{
Statement stmt = conn.createStatement(
          ResultSet.TYPE_FORWARD_ONLY,
          ResultSet.CONCUR_READ_ONLY);
ResultSet rs =
stmt.executeQuery("SELECT * from User");

}
catch(SQLException ex) { .... }
finally { .... }
```

ResultSet Type

- `ResultSet.TYPE_FORWARD_ONLY`
- `ResultSet.TYPE_SCROLL_INSENSITIVE`
- `ResultSet.TYPE_SCROLL_SENSITIVE`

Notes:

- `FORWARD_ONLY vs. SCROLL`

- `INSENSITIVE vs. SENSITIVE`: the result set is (in) sensitive to changes made after the result set was created.

ResultSet type (MySQL)

`DatabaseMetaData.supportsResultSetType(int)`

returns true if the specified `ResultSet` type is supported and false otherwise.

```
DatabaseMetaData dmd = conn.getMetaData();

dmd.supportsResultSetType(ResultSet.TYPE_FORWARD_ONLY);
// false
dmd.supportsResultSetType(ResultSet.TYPE_SCROLL_INSENSIT
IVE); // true
dmd.supportsResultSetType(ResultSet.TYPE_SCROLL_SENSITIV
E); // false
```

Updatable ResultSet

This option indicates if the ResultSet is updatable or not.

- `ResultSet.CONCUR_READ_ONLY`
- `ResultSet.CONCUR_UPDATABLE:`

`DatabaseMetaData.supportsResultSetConcurrency(int, int)` returns true if the specified concurrency level is supported by the driver and false otherwise.

ResultSet Concurrency (MySQL)

```
DatabaseMetaData dmd = conn.getMetaData();

dmd.supportsResultSetConcurrency(ResultSet.T
YPE_SCROLL_INSENSITIVE,ResultSet.CONCUR_READ
_ONLY); // true

dmd.supportsResultSetConcurrency(ResultSet.T
YPE_SCROLL_INSENSITIVE,ResultSet.CONCUR_UPDA
TABLE); // true
```

ResultSet methods

A ResultSet object maintains a cursor that points to the current row in the result set.

- **Navigational methods:** used to move the cursor around.
- **Get methods:** used to view the data in the **columns** of the current row
- **Update methods:** used to update the data in the columns of the current row.

Note: the current row is the tuple the cursor just jumped over by next() or previous().

Navigation Methods

`beforeFirst() vs first()`

- `beforeFirst()` moves the cursor to the front of this ResultSet object, just before the first row. A subsequent next() call makes the first row the current row.

- `first()` The first row becomes the current row. A subsequent next() makes the second the current row.

first () = beforeFirst() + next()

Navigation Methods

`beforeFirst() vs first()`

Suppose rows A, B, C are in the ResultSet.

```
rs.first()
while (rs.next())
{ // get and print the columns of
the current row}
```

Without rs.first(): A, B, C

With rs.first(): B, C

Navigation Methods

`next()` and `previous()`

- `next()`

- Moves the cursor forward one row from its current position.
- A cursor is initially positioned before the first row;
- the first call to the method next makes the first row the current row;
- When a call to the next method returns false, the cursor is positioned after the last row. (No hasNext() unlike Java)

- `previous()`

- Moves the cursor to the previous row in this ResultSet object.
- When the cursor is before the first row, the previous method returns false, the cursor is positioned before the first row. (No hasPrevious() unlike Java)

ResultSet: Update

Note: The type of ResultSet should be ResultSet.***CONCUR_UPDATABLE***

(1) To update the current row.

while (rs.next())

{ int id = rs.getInt("id");

 String name=rs.getString("name");

 int age = rs.getInt("age");

 rs.**updateInt**("age", age * 10); // update the row in ResultSet

 rs.**updateRow**(); // update the row in the database

}

ResultSet: Update

(2) To update a row at an absolute position:

```
rs.absolute(2); // the 2nd row will
become the current row.
rs.updateInt("id", 890);
rs.updateString("name", "Smith");
rs.updateInt("age", 43);
```

To cancel update

- `cancelRowUpdates()` cancels the updates made to the current row in this ResultSet object.

```
rs.updateInt("age", age * 10);
// Updating the ResultSet
```

```
rs.cancelRowUpdates();

rs.updateRow();
```

Note: Should be called before rs.updateRow() to be effective.

ResultSet:Insert

Use a staging tuple

```
rs.moveToInsertRow();
rs.updateInt("id", 890);
rs.updateString("name", "Smith");
rs.updateInt("age", 43);

rs.insertRow(); // into this ResultSet and into
the database; cursor is after the last element.
rs.beforeFirst(); //move the cursor to a
desired position.
```

ResultSet: Delete

```
rs.first(); // the 1st row becomes the
current row.
rs.deleteRow(); //Deletes the current
row from this ResultSet and also from
the underlying database.
```

Example:ResultSet

- JDBCResultSet.java

SQLExceptions

A SQLException object contains

A description of the error	Methods of SQLException class
A SQLState code	getSQLState()
An error code	getErrorCode() (vendor specific error code)
A cause	getCause()
A reference to any *chained* exceptions	getNextException()

Example: SQLException Handling

- ExceptionExample.java
- **Mapping MySQL Error Numbers to JDBC SQLState Codes**

https://docs.oracle.com/cd/E17952_01/connector-j-8.0-en/connector-j-reference-error-sqlstates.html

SQLStates

- SQL State (SQLSTATE) Error Codes are defined by the ISO/ANSI and Open Group (X/Open) SQL Standards.

- List of SQLStates (SQLStates.txt)

  A complete list of the SQLSTATE error codes can be found in the documentations of the ISO/ANSI and Open Group (X/Open) SQL Standards.

- Mapping MySQL Error Numbers to JDBC SQLState Codes: https://docs.oracle.com/cd/E17952_01/connector-j-8.0-en/connector-j-reference-error-sqlstates.html

Some popular JDBC drivers

RDBMS	JDBC Driver Name
MySQL	Driver Name com.mysql.jdbc.Driver Database URL format: jdbc:mysql//hostname/databaseName
Oracle	Driver Name: oracle.jdbc.driver.OracleDriver Database URL format: jdbc:oracle:thin@hostname:portnumber:databaseName
DB2	Driver Name: COM.ibm.db2.jdbc.net.DB2Driver Database URL format: jdbc:db2:hostname:portnumber/databaseName
Access	Driver Name: com.jdbc.odbc.JdbcOdbcDriver Database URL format: jdbc:odbc:databaseName