Core Java

JDBC

- Q & A
- JDBC DAOs
- Calling Stored Procedure
- Transaction Management
- ResultSet

Q & A

- Assignments discussion

JDBC

DAO class

- In enterprise applications, there are multiple tables and frequent data transfer from database is needed.
- Instead of writing a JDBC code in multiple Java files of the application (as and when needed), it is good practice to keep all the JDBC code in a centralized place -- in a single application layer.
- DAO (Data Access Object) class is standard way to implement all CRUD operations specific to a table. It is advised to create different DAO for different table.
- DAO classes makes application more readable/maintainable.
- Example 1:

```
class StudentDao implements AutoClosable {
    private Connection con;
```

```
    public StudentDao() throws Exception {
        con = DriverManager.getConnection(DbUtil.DB_URL, DbUtil.DB_USER, DbUtil.DB_PASSWORD);
    }
    public void close() {
        try{
            if(con != null)
                con.close();
        } catch(Exception ex) {
        }
    }
    public int update(Student s) throws Exception {
        int count = 0;
        String sql = "UPDATE students SET name=?, marks=? WHERE roll=?"
        try(PreparedStatement stmt = con.prepareStatement(sql)) {
            // optionally you may create PreparedStatement in constructor (as implemented)
            stmt.setString(1, s.getName());
            stmt.setDouble(2, s.getMarks());
            stmt.setInt(3, s.getRoll());
            count = stmt.executeUpdate();
        }
        return count;
    }
}
```

```
// in main()
try(StudentDao dao = new StudentDao()) {
    System.out.print("Enter roll to be updated: ");
    int roll = sc.nextInt();
    System.out.print("Enter new name: ");
    String name = sc.next();
    System.out.print("Enter new marks: ");
    double marks = sc.next();
    Student s = new Student(roll, name, marks);
    int cnt = dao.update(s);
```

```
    System.out.println("Rows updated: " + cnt);
} // dao.close()
catch(Exception ex) {
    ex.printStackTrace();
}
```

- Example 2:

```
// POJO (Entity)
class Emp {
    private int empno;
    private String ename;
    private Date hire;
    // ...
}
```

```
class DbUtil {
    public static final String DB_DRIVER = "com.mysql.cj.jdbc.Driver";
    public static final String DB_URL = "jdbc:mysql://localhost:3306/test";
    public static final String DB_USER = "nilesh";
    public static final String DB_PASSSWD = "nilesh";

    static {
        try {
            Class.forName(DB_DRIVER);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
            System.exit(0);
        }
    }

    public static Connection getConnection() throws Exception {
```

```
        return DriverManager.getConnection(DB_URL, DB_USER, DB_PASSSWD);
    }
}
```

```
class EmpDao implements AutoClosable {
    private Connection con;
    public EmpDao() throws Exception {
        con = DbUtil.getConnection();
    }
    public void close() {
        try {
            if(con != null)
                con.close();
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }
    public int update(Emp e) throws Exception {
        String sql = "UPDATE emp SET ename=?, hire=? WHERE id=?";
        try(PreparedStatement stmt = con.prepareStatement(sql)) {
            stmt.setString(1, e.getEname());
            java.util.Date uDate = e.getHire();
            java.sql.Date sDate = new java.sql.Date(uDate.getTime());
            stmt.setDate(2, sDate);
            stmt.setInt(3, e.getEmpno());
            int cnt = stmt.executeUpdate();
            return cnt;
        } // stmt.close();
    }
    // ...
}
```

```
// in main()
try(EmpDao dao = new EmpDao()) {
    Emp e = new Emp();
    // input emp data from end user (Scanner)
    /*
    String dateStr = sc.next(); // dd-MM-yyyy
    SimpleDateFormat sdf = new SimpleDateFormat("dd-MM-yyyy");
    java.util.Date uDate = sdf.parse(dateStr);
    e.setHire(uDate);
    */
    int cnt = dao.update(e);
    System.out.println("Emps updated: " + cnt);
} // dao.close();
catch(Exception ex) {
    ex.printStackTrace();
}
```

- Example 3:

```
// POJO (Entity)
class Emp {
    private int empno;
    private String ename;
    private Date hire;
    // ...
}
```

```
class DbUtil {
    public static final String DB_DRIVER = "com.mysql.cj.jdbc.Driver";
    public static final String DB_URL = "jdbc:mysql://localhost:3306/test";
    public static final String DB_USER = "nilesh";
```

```
    public static final String DB_PASSSWD = "nilesh";

    static {
        try {
            Class.forName(DB_DRIVER);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
            System.exit(0);
        }
    }

    public static Connection getConnection() throws Exception {
        return DriverManager.getConnection(DB_URL, DB_USER, DB_PASSSWD);
    }
}
```

```
class EmpDao implements AutoClosable {
    private Connection con;
    private PreparedStatement stmtFindById;
    // ...
    public EmpDao() throws Exception {
        con = DbUtil.getConnection();
        String sql = "SELECT * FROM emp WHERE empno=?";
        stmtFindById = con.prepareStatement(sql);
        // ...
    }
    public void close() {
        try {
            // ...
            if(stmtFindById != null)
                stmtFindById.close();
            if(con != null)
                con.close();
        } catch(Exception ex) {
```

```
            ex.printStackTrace();
        }
    }
    public Emp findById(int empno) throws Exception {
        stmtFindById.setInt(1, empno);
        try(ResultSet rs = stmtFindById.executeQuery()) {
            if(rs.next()) {
                int empno = rs.getInt("empno");
                String ename = rs.getString("ename");
                java.sql.Date sDate = rs.getDate("hire");
                // ...
                java.util.Date uDate = new java.util.Date( sDate.getTime() );
                Emp e = new Emp(empno, ename, uDate);
                return e;
            }
        } // rs.close();
        return null;
    }
}
```java
// in main()
try(EmpDao dao = new EmpDao()) {
    System.out.print("Enter empno to find: ");
    id = sc.nextInt();
    e = dao.findById(id);
    System.out.println("Found: " + e);

    System.out.print("Enter empno to find: ");
    id = sc.nextInt();
    e = dao.findById(id);
    System.out.println("Found: " + e);

    System.out.print("Enter empno to find: ");
    id = sc.nextInt();
    e = dao.findById(id);
    System.out.println("Found: " + e);
```
```

```
}
catch(Exception ex) {
    ex.printStackTrace();
}
```

Call Stored Procedure using JDBC (without OUT parameters)

- Stored Procedure - Increment votes of candidate with given id

```
DELIMITER //

CREATE PROCEDURE sp_incrementvotes(IN p_id INT)
BEGIN
    UPDATE candidates SET votes=votes+1 WHERE id=p_id;
END;
//

DELIMITER ;
```

```
CALL sp_incrementvotes(10);
```

- JDBC use CallableStatement interface to invoke the stored procedures.
- CallableStatement interface is extended from PreparedStatement interface.
- Steps to call Stored procedure are same as PreparedStatement.
    - Create connection.
    - Create CallableStatement using con.prepareCall("CALL ...").
    - Set IN parameters using stmt.setXYZ(...);

- Execute the procedure using stmt.executeQuery() or stmt.executeUpdate().
- Close statement & connection.

- To invoke stored procedure, in general stmt.execute() is called. This method returns true, if it is returning ResultSet (i.e. multi-row result). Otherwise it returns false, if it is returning update/affected rows count.

```
boolean isResultSet = stmt.execute();
if(isResultSet) {
    ResultSet rs = stmt.getResultSet();
    // process the ResultSet
}
else {
    int count = stmt.getUpdateCount();
    // process the count
}
```

Call Stored Procedure using JDBC (with OUT parameters)

- Stored Procedure - Get quote and author of given quote id -- using OUT parameters.

```
DELIMITER //

CREATE PROCEDURE sp_getpartyvotes(IN p_party CHAR(40), OUT p_votes INT)
BEGIN
    SELECT SUM(votes) INTO p_votes FROM candidates WHERE party=p_party;
END;
//

DELIMITER ;
```

```
CALL sp_getpartyvotes('BJP', @votes);

SELECT @votes;
```

- Steps to call Stored procedure with out params.
    - Create connection.
    - Create CallableStatement using con.prepareCall("CALL ...").
    - Set IN parameters using stmt.setXYZ(...) and register out parameters using stmt.registerOutParam(...).
    - Execute the procedure using stmt.execute().
    - Get values of out params using stmt.getXYZ(paramNumber).
    - Close statement & connection.

Transaction Management

- RDBMS Transactions
    - Transaction is set of DML operations to be executed as a single unit. Either all queries in tx should be successful or all should be discarded.
    - The transactions must be atomic. They should never be partial.

```
CREATE TABLE accounts(id INT, type CHAR(30), balance DOUBLE);
INSERT INTO accounts VALUES (1, 'Saving', 30000.00);
INSERT INTO accounts VALUES (2, 'Saving', 2000.00);
INSERT INTO accounts VALUES (3, 'Saving', 10000.00);

SELECT * FROM accounts;

START TRANSACTION;
--SET @@autocommit=0;

UPDATE accounts SET balance=balance-3000 WHERE id=1;
UPDATE accounts SET balance=balance+3000 WHERE id=2;
```

```
SELECT * FROM accounts;

COMMIT;
-- OR
ROLLBACK;
```

- JDBC transactions (Logical code)

```
try(Connection con = DriverManager.getConnection(DB_URL, DB_USER, DB_PASSWORD)) {
    con.setAutoCommit(false); // start transaction
    String sql = "UPDATE accounts SET balance=balance+? WHERE id=?";
    try(PreparedStatement stmt = con.prepareStatement(sql)) {
        stmt.setDouble(1, -3000.0); // amount=3000.0
        stmt.setInt(2, 1); // accid = 1
        cnt1 = stmt.executeUpdate();
        stmt.setDouble(1, +3000.0); // amount=3000.0
        stmt.setInt(2, 2); // accid = 2
        cnt2 = stmt.executeUpdate();
        if(cnt1 == 0 || cnt2 == 0)
            throw new RuntimeException("Account Not Found");
    }
    con.commit(); // commit transaction
}
catch(Exception e) {
    e.printStackTrace();
    con.rollback(); // rollback transaction
}
```

ResultSet

- ResultSet types
  - TYPE_FORWARD_ONLY -- default type
    - next() -- fetch the next row from the db and return true. If no row is available, return false.

```
while(rs.next()) {
    // ...
}
```

- TYPE_SCROLL_INSENSITIVE
  - next() -- fetch the next row from the db and return true. If no row is available, return false.
  - previous() -- fetch the previous row from the db and return true. If no row is available, return false.
  - absolute(rownum) -- fetch the row with given row number and return true. If no row is available (of that number), return false.
  - relative(rownum) -- fetch the row of next rownum from current position and return true. If no row is available (of that number), return false.
  - first(), last() -- fetch the first/last row from db.
  - beforeFirst(), afterLast() -- set ResultSet to respective positions.
  - INSENSITIVE -- After taking ResultSet if any changes are done in database, those will NOT be available/accessible using ResultSet object. Such ResultSet is INSENSITIVE to the changes (done externally).
- TYPE_SCROLL_SENSITIVE
  - SCROLL -- same as above.
  - SENSITIVE -- After taking ResultSet if any changes are done in database, those will be available/accessible using ResultSet object. Such ResultSet is SENSITIVE to the changes (done externally).

- ResultSet concurrency
  - CONCUR_READ_ONLY -- Using this ResultSet one can only read from db (not DML operations). This is default concurrency.
  - CONCUR_UPDATABLE -- Using this ResultSet one can read from db as well as perform INSERT, UPDATE and DELETE operations on database.

```
String sql = "SELECT roll, name, marks FROM students";
stmt = con.prepareStatement(sql, ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
rs = stmt.executeQuery();
```

```
rs.absolute(2); // moves the cursor to the 2nd row of rs
rs.updateString("name", "Bill"); // updates the 'name' column of row 2 to be Bill
```

```
rs.updateDouble("marks", 76.32); // updates the 'marks' column of row 2 to be 76.32
rs.updateRow(); // updates the row in the database
```

```
rs.moveToInsertRow(); // moves cursor to the insert row -- is a blank row
rs.updateInt(1, 9); // updates the 1st column (roll) to be 9
rs.updateString(2, "AINSWORTH"); // updates the 2nd column (name) of to be AINSWORTH
rs.updateDouble(3, 76.23); // updates the 3rd column (marks) to true 76.23
rs.insertRow(); // inserts the row in the database
rs.moveToCurrentRow();
```

```
rs.absolute(2); // moves the cursor to the 2nd row of rs
rs.deleteRow(); // deletes the current row from the db
```

Assignments

1. Complete CandidateDAO class. Refer the comments "TODO". Implement one more method given below.

```
List<PartyVotes> getPartywiseVotes(); // get partywise total votes.
    // create a POJO class "PartyVotes" with two fields "party" and "votes".
    // no separate db table to be created.
```

2. Create an UserDAO class. Implement CRUD operations (similar to CandidateDAO).