Core Java

Agenda

- JDBC
  - Driver types
  - Jdbc Interfaces
  - Call stored procedure
  - Transaction management
  - DAO
  - ResultSet -- types

JDBC

- JDBC is a specification -- Interfaces and Helper classes.
- This specs is implemented by RDBMS drivers e.g. Oracle driver, MySQL driver, ...
- Programming steps
  - step 0: Add JDBC driver into the project classpath.
  - step 1: Load and register Driver class (One-time).
  - step 2: Create the connection using DriverManager.
  - step 3: Create the PreparedStatement with SQL (parameterized) query.
  - step 4: Execute the query (executeUpdate() or executeQuery()) and process the result.
  - step 5: Close statement & connection.

JDBC concepts

java.sql.Driver

- Implemented in JDBC drivers.
  - MySQL: com.mysql.cj.jdbc.Driver
  - Oracle: oracle.jdbc.OracleDriver

    - Postgres: org.postgresql.Driver
- Driver needs to be registered with DriverManager before use.
- When driver class is loaded, it is auto-registered (Class.forName()).
- Driver object is responsible for establishing database "Connection" with its connect() method.
- This method is called from DriverManager.getConnection().

java.sql.Connection

- Connection object represents database socket connection.
- All communication with db is carried out via this connection.
- Connection functionalities:
    - Connection object creates a Statement.
    - Transaction management.

java.sql.Statement

- Represents SQL statement/query.
- To execute the query and collect the result.

```
Statement stmt = con.createStatement();
```

```
ResultSet rs = stmt.executeQuery(selectQuery);
```

```
int count = stmt.executeUpdate(nonSelectQuery);
```

- Since query built using string concatenation, it may cause SQL injection.

java.sql.PreparedStatement

- Inherited from java.sql.Statement.
- Represents parameterized SQL statement/query.
- The query parameters (?) should be set before executing the query.
- Same query can be executed multiple times, with different parameter values.
- This speed up execution, because query syntax checking is done only once.

```
PreparedStatement stmt = con.prepareStatement(query);
```

```
stmt.setInt(1, intValue);
stmt.setString(2, stringValue);
stmt.setDouble(3, doubleValue);
stmt.setDate(4, dateObject); // java.sql.Date
stmt.setTimestamp(5, timestampObject); // java.sql.Timestamp
```

```
ResultSet rs = stmt.executeQuery();
// OR
int count = stmt.executeUpdate();
```

java.sql.ResultSet

- ResultSet represents result of SELECT query. The result may have one/more rows and one/more columns.
- Can access only the columns fetched from database in SELECT query (projection).

```
// SELECT id, quote, created_at FROM quotes
ResultSet rs = stmt.executeQuery();
```

```
while(rs.next()) {
    int id = rs.getInt("id");
    String quote = rs.getString("quote");
    Timestamp createdAt = rs.getTimestamp("created_at"); // java.sql.Timestamp
    // ...
}
```

```
// SELECT id, quote, created_at FROM quotes
ResultSet rs = stmt.executeQuery();
while(rs.next()) {
    int id = rs.getInt(1);
    String quote = rs.getString(2);
    Timestamp createdAt = rs.getTimestamp(3); // java.sql.Timestamp
    // ...
}
```

Handling Date types

- EMP table: empno INT, ename VARCHAR(20), job VARCHAR(10), mgr INT, hire DATE, sal DOUBLE, comm DOUBLE, deptno INT
- Emp POJO:

```
public class Emp {
    // ...
    private java.util.Date hire;
}
```

- Get date from the db and set in pojo.

```

```

```
// SELECT empno, ename, hire FROM emp
ResultSet rs = stmt.executeQuery();
while(rs.next()) {
    Emp e = new Emp();
    java.sql.Date hireDate = rs.getDate("hire");
    java.util.Date hire = new Date(hireDate.getTime());
    e.setHire(hire);
}
```

- Get date from pojo and set in stmt.

```
// UPDATE emp SET hire=? WHERE empno=?
java.util.Date hire = e.getHire();
java.sql.Date hireDate = new Date(hire.getTime());
stmt.setDate(1, hireDate);
```

- QUOTES table: id INT, quote VARCHAR(500), author VARCHAR(40), user_id INT, created_at TIMESTAMP
- Quote POJO:

```
public class Quote {
    // ...
    private java.util.Date createdDate;
}
```

- Get timestamp from the db and set in pojo.

```
// SELECT id,quote,created_at FROM quotes
ResultSet rs = stmt.executeQuery();
while(rs.next()) {
```

```
    Quote q = new Quote();
    java.sql.Timestamp createdTs = rs.getTimestamp("hire");
    java.util.Date createdAt = new Date(createdTs.getTime());
    q.setCreatedAt(createdAt);
}
```

- Get timestamp from pojo and set in stmt.

```
// UPDATE quotes SET created_at=? WHERE id=?
java.util.Date createdAt = e.getCreatedAt();
java.sql.Timestamp createdTs = new Date(createdAt.getTime());
stmt.setTimestamp(1, createdTs);
```

Call Stored Procedure using JDBC (without OUT parameters)

- Stored Procedure - Get all quotes along with user name.

```
DELIMITER //

CREATE PROCEDURE sp_getquotes(IN p_userid INT)
BEGIN
    SELECT q.quote, q.author, q.created_at, u.first_name, u.last_name FROM quotes q INNER JOIN users u ON
q.user_id = u.id WHERE u.id=p_userid;
END;
//

DELIMITER ;
```

```
CALL sp_getquotes(1);
```

- 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_getquote_details(IN p_id INT, OUT p_quote VARCHAR(500), OUT p_author VARCHAR(40)) BEGIN SELECT quote INTO p_quote FROM quotes WHERE id=p_id; SELECT author INTO p_author FROM quotes WHERE id=p_id; END; //

DELIMITER ; SQL CALL sp_getquote_details(1, @quote, @author); SELECT @quote, @author; ```

- 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
}
```

DAO class

- In enterprise applications there are multiple tables and frequent data transfer from database is needed.
- Instead of writing 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.

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
```

Quick Revision

Statements

- interface Statement: executing SQL queries
  - Drawback: Prepare queries by String concatenation. May cause SQL injection.
- interface PreparedStatement extends Statement: executing parameterized SQL queries
  - Prevent SQL injection
  - Efficient execution if same query is to be executed repeatedly.
- interface CallableStatement extends PreparedStatement: executing stored procedures in db.

- Prevent SQL injection
- More efficient execution if same query is to be executed repeatedly.

Executing statements

- Load and register class. In JDBC 4, this step is automated in Core Java applications (provided class is available in classpath).

```
static {
    try {
        Class.forName(DB_DRIVER);
    }
    catch(Exception ex) {
        ex.printStackTrace();
        System.exit(0);
    }
}
```

- Executing SELECT statements

```
try(Connection con = DriverManager.getConnection(DB_URL, DB_USER, DB_PASSWORD)) {
    String sql = "SELECT * FROM students WHERE marks > ?";
    try(PreparedStatement stmt = con.prepareStatement(sql)) {
        stmt.setDouble(1, marks);
        try(ResultSet rs = stmt.executeQuery()) {
            while(rs.next()) {
                int roll = rs.getInt("roll");
                String name = rs.getString("name");
                double smarks = rs.getDouble("marks");
                Student s = new Student(roll, name, marks);
                System.out.println(s);
            }
        } // rs.close()
    } // stmt.close()
```

```
} // con.close()
catch(Exception ex) {
    ex.printStackTrace();
}
```

- Executing non-SELECT statements

```
try(Connection con = DriverManager.getConnection(DB_URL, DB_USER, DB_PASSWORD)) {
    String sql = "DELETE FROM students WHERE marks > ?";
    try(PreparedStatement stmt = con.prepareStatement(sql)) {
        stmt.setDouble(1, marks);
        int count = stmt.executeUpdate();
        System.out.println("Rows Deleted: " + count);
    } // stmt.close()
} // con.close()
catch(Exception ex) {
    ex.printStackTrace();
}
```

DAO class

```
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();
}
```

JDBC Tutorial (Refer after Lab time - If required)

- JDBC 1 - Getting Started : https://youtu.be/SgAVBLZ_rww
- Jdbc 2 - PreparedStatement and CallableStatement : https://youtu.be/GzSUyiep7Mw
- Jdbc 3 - Transaction Management : https://youtu.be/Wh6nrkB_o8c

Assignment

1. Complete Quotes application with proper use of DAO and Service layer.

Quotes application development

- After sign-in current user can be saved. It can be used for other functionalities.

1. Create database tables. May enter some dummy data.
2. Create Java project with a Main class.
3. DbUtil class and Pojo classes
4. UserDao class -- con field, ctor to get connection, close() from AutoClosable
5. UserDao class -- int save(User u);
6. UserService class -- void signUp();
7. Main class -- Menu driven
8. Main class menu -- call userService.signUp();
9. Execute and test.
10. UserDao class -- User findByEmail(String email);
11. UserService class -- void signIn(); -- If sign-in success, store current user object into Main.curUser (public static field).
12. Main class menu -- call userService.signIn();
13. Execute and test.
14. UserDao class -- int changePassword(int userId, String newPasswd);
15. UserService class -- void changePassword(); take userid from Main.curUser.getId() and new password from user.
16. Main class menu -- call userService.changePassword();
17. Execute and test.
18. UserDao class -- int changeProfile(User user);
19. UserService class -- void changeProfile(); take userid from Main.curUser.getId() and take modified fields from user and set in Main.curUser.
20. Main class menu -- call userService.changeProfile();
21. Execute and test.

22. QuoteDao class -- con field, ctor to get connection, close()
23. QuoteDao class -- `List<Quote> findAll()`;
24. QuoteService class -- void findAll();
25. Main class menu -- call quoteService.displayAllQuotes();
26. Execute and test.
27. QuoteDao class -- `List<Quote> findByUserId(int userId)`;
28. QuoteService class -- void displayQuotesOfUser(); -> call quoteDao.findByUserId() method with Main.curUser.getId() parameter.
29. Main class menu -- call quoteService.findByUserId();
30. Execute and test.
31. ...