Agenda

- JDBC

JDBC

- RDBMS understand SQL language only.
- JDBC driver converts Java requests in database understandable form and database response in Java understandable form.
- JDBC drivers are of 4 types

1. Type I - Jdbc Odbc Bridge driver

    - ODBC is standard of connecting to RDBMS (by Microsoft).
    - Needs to create a DSN (data source name) from the control panel.
    - From Java application JDBC Type I driver can communicate with that ODBC driver (DSN).
    - The driver class: sun.jdbc.odbc.JdbcOdbcDriver -- built-in in Java.
    - database url: jdbc:odbc:dsn
    - Advantages:
    - Can be easily connected to any database.
    - Disadvantages:
    - Slower execution (Multiple layers).
    - The ODBC driver needs to be installed on the client machine.

2. Type II - Partial Java/Native driver

    - Partially implemented in Java and partially in C/C++. Java code calls C/C++ methods via JNI.
    - Different driver for different RDBMS. Example: Oracle OCI driver.
    - Advantages:
    - Faster execution
    - Disadvantages:
        - Partially in Java (not truely portable)
        - Different driver for Different RDBMS

3. Type III - Middleware/Network driver

    - Driver communicate with a middleware that in turn talks to RDBMS.
    - Example: WebLogic RMI Driver
    - Advantages:
        - Client coding is easier (most task done by middleware)
    - Disadvantages:
        - Maintaining middleware is costlier
        - Middleware specific to database

4. Type IV

    - Database specific driver written completely in Java.
    - Fully portable.
    - Most commonly used.

  - Example: Oracle thin driver, MySQL Connector/J, ...

MySQL Programming Steps

- step 0: Add JDBC driver into project/classpath. In Eclipse, project -> right click -> properties -> java build path -> libraries -> Add external jars -> select mysql driver jar.
- step 1: Load and register JDBC driver class. These drivers are auto-registered when loaded first time in JVM. This step is optional in Java SE applications from JDBC 4 spec.

```
Class.forName("com.mysql.cj.jdbc.Driver");
// for Oracle: Use driver class oracle.jdbc.driver.OracleDriver
```

- step 2: Create JDBC connection using helper class DriverManager.

```
// db url = jdbc:dbname://db-server:port/database
Connection con =
DriverManager.getConnection("jdbc:mysql://localhost:3306/classwork", "root",
"manager");
// for Oracle: jdbc:oracle:thin:@localhost:1521:sid
```

- step 3: Create the statement.

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

- step 4: Execute the SQL query using the statement and process the result.

```
String sql = "non-select query";
int count = stmt.executeUpdate(sql); // returns number of rows affected
OR
String sql = "select query";
ResultSet rs = stmt.executeQuery(sql);
while(rs.next()) // fetch next row from db(return false when all rows completed)
{
  x = rs.getInt("col1");
  // get first column from the current row
  y = rs.getString("col2");
  // get second column from the current row
  z = rs.getDouble("col3");
  // get third column from the current row
  // process/print the result
}
rs.close();
```

- step 5: Close statement and connection.

```
con.close();
stmt.close();
```

MySQL Driver Download

https://mvnrepository.com/artifact/com.mysql/mysql-connector-j/8.1.0

SQL Injection

- Building queries by string concatenation is inefficient as well as insecure.
- Example:

```
dno = sc.nextLine();
sql = "SELECT * FROM emp WHERE deptno="+dno;
```

- If user input "10", then effective SQL will be "SELECT _ FROM emp WHERE deptno=10". This will select all emps of deptno 10 from the RDBMS.
- If user input "10 OR 1", then effective SQL will be "SELECT _ FROM emp WHERE deptno=10 OR 1". Here "1" represent true condition and it will select all rows from the RDBMS.
- In Java, it is recommeded NOT to use "Statement" and building SQL by string concatenation. Instead use PreparedStatement.

PreparedStatement

- PreparedStatement represents parameterized queries.

```
String sql = "SELECT * FROM students WHERE name=?";
PreparedStatement stmt = con.prepareStatement(sql);

System.out.print("Enter name to find: ");
String name = sc.next();

stmt.setString(1, name);
ResultSet rs = stmt.executeQuery();

while(rs.next()) {
  int roll = rs.getInt("roll");
  String name = rs.getString("name");
  double marks = rs.getDouble("marks");
  System.out.printf("%d, %s, %.2f\n", roll, name, marks);
}
```

- The same PreparedStatement can be used for executing multiple queries. There is no syntax checking repeated. This improves the performance.

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
// ...
}
```

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 = "root";
public static final String DB_PASSSWD = "root";
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 (using the POJO and DBUtil same as Example 2)

~~~
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();
  }
```