Core Java

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.

MySQL Programming steps -- PreparedStatement

1. Add JDBC driver into project/classpath.
    - Java project -> Properties -> Java Build Path -> Libraries -> Add External Jars -> select MySQL JDBC driver jar -> Apply and Close.
2. Load and register driver class.

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

3. Create database connection.

```
Connection con =
DriverManager.getConnection("jdbc:mysql://localhost:3306/dbname", "dbuser",
"dbpassword");
```

4. Create PreparedStatement with (paramterized) SQL query.

```
String sql = "sql query with ?";
PreparedStatement stmt = con.prepareStatement(sql);
```

5. Set param values, execute the query and process the result.

```
stmt.setInt(1, val1); // set 1st param ? value
stmt.setString(2, val2); // set 2nd param ? value
```

```
// for non-SELECT queries
int count = stmt.executeUpdate();
```

```
// for SELECT queries
ResultSet rs = stmt.executeQuery();
while(rs.next()) {
    int val1 = rs.getInt("col1");
    String val2 = rs.getString("col2");
    // ...
}
rs.close();
```

6. Close statement and connection.

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

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();
```

Call Stored Procedure using JDBC (without OUT parameters)

- Stored Procedure - Change price of given book id.

```
DELIMITER //

CREATE PROCEDURE sp_updateprice(IN p_id INT, IN p_price DOUBLE)
BEGIN
    UPDATE books SET price=p_price WHERE id=p_id;
END;
//

DELIMITER ;
```

```
CALL sp_updateprice(22, 543.21);
```

- 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 title and price of given book id.

```
DELIMITER //

CREATE PROCEDURE sp_gettitleprice(IN p_id INT, OUT p_name CHAR(40), OUT
p_price DOUBLE)
BEGIN
    SELECT name INTO p_name FROM books WHERE id=p_id;
    SELECT price INTO p_price FROM books WHERE id=p_id;
END;
//

DELIMITER ;
```

```
CALL sp_gettitleprice(22, @p_name, @p_price);

SELECT @p_name, @p_price;
```

- 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-4000 WHERE id=1;
UPDATE accounts SET balance=balance+4000 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
```

###Read Uncommitted 1.This is the lowest isolation level. 2.A transaction can read uncommitted changes made by other transactions. 3.This means it can see data that other transactions have modified but not yet committed. 4.This level allows for dirty reads, where a transaction reads data that might later be rolled back. 5.Typically used when performance is the highest priority, and data consistency is less critical (e.g., logging systems).

###Read Committed(Non-repeatable Reads) 1.A transaction can only read committed data (i.e., data that has been committed by other transactions). 2.Dirty reads are prevented, but it can still encounter non-repeatable reads. 3.A non-repeatable read happens when the value of a record is changed by another transaction after the current transaction has read it.

###Repeatable Read(Phantom Reads) 1.A transaction can read data repeatedly, and the data it reads will not change during the transaction, even if other transactions modify it. 2.This level prevents dirty reads and non-repeatable reads. 3.However, phantom reads are still possible (new rows might appear or disappear between reads in the same transaction).

###Serializable 1.This is the highest isolation level. 2.It ensures complete isolation by serializing transactions — meaning they are executed as if they were happening one after the other, with no overlapping. 3.It prevents dirty reads, non-repeatable reads, and phantom reads.