

JDBC Questions and Answers

Q. What is DAO factory design pattern in Java?

Data Access Object Pattern or DAO pattern is used to separate low level data accessing API or operations from high level business services.

DAO pattern is based on abstraction and encapsulation design principles and shields rest of application from any change in the persistence layer e.g. change of database from Oracle to MySQL, change of persistence technology e.g. from File System to Database.

Step 1: Create Value Object [Student.java]

```
public class Student {
    private String name;
    private int rollNo;

    Student(String name, int rollNo){
        this.name = name;
        this.rollNo = rollNo;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getRollNo() {
        return rollNo;
    }

    public void setRollNo(int rollNo) {
        this.rollNo = rollNo;
    }
}
```

Step 2: Create Data Access Object Interface [StudentDao.java]

```
import java.util.List;

public interface StudentDao {
    public List<Student> getAllStudents();
    public Student getStudent(int rollNo);
    public void updateStudent(Student student);
}
```

```

    public void deleteStudent(Student student);
}

```

Step 3: Create concrete class implementing above interface [StudentDaoImpl.java]

```

import java.util.ArrayList;
import java.util.List;

public class StudentDaoImpl implements StudentDao {

    // list is working as a database
    List<Student> students;

    public StudentDaoImpl(){
        students = new ArrayList<Student>();
        Student student1 = new Student("Robert",0);
        Student student2 = new Student("John",1);
        students.add(student1);
        students.add(student2);
    }

    @Override
    public void deleteStudent(Student student) {
        students.remove(student.getRollNo());
        System.out.println("Student: Roll No " + student.getRollNo() + ", deleted from d
atabase");
    }

    // retrieve list of students from the database
    @Override
    public List<Student> getAllStudents() {
        return students;
    }

    @Override
    public Student getStudent(int rollNo) {
        return students.get(rollNo);
    }

    @Override
    public void updateStudent(Student student) {
        students.get(student.getRollNo()).setName(student.getName());
        System.out.println("Student: Roll No " + student.getRollNo() + ", updated in the
database");
    }
}

```

Step 4: Use the StudentDao to demonstrate Data Access Object pattern usage [DaoPatternDemo.java]

```

public class DaoPatternDemo {
    public static void main(String[] args) {
        StudentDao studentDao = new StudentDaoImpl();
    }
}

```

```

        // print all students
        for (Student student : studentDao.getAllStudents()) {
            System.out.println("Student: [RollNo : " + student.getRollNo() + ", Name : "
+ student.getName() + " ]");
        }

        // update student
        Student student =studentDao.getAllStudents().get(0);
        student.setName("Michael");
        studentDao.updateStudent(student);

        // get the student
        studentDao.getStudent(0);
        System.out.println("Student: [RollNo : " + student.getRollNo() + ", Name : " + s
tudent.getName() + " ]");
    }
}

```

Output:

```



Student: [RollNo : 0, Name : Robert ]
Student: [RollNo : 1, Name : John ]
Student: Roll No 0, updated in the database
Student: [RollNo : 0, Name : Michael ]



```

Q. What are the differences between ResultSet and RowSet?

A **ResultSet** maintains a connection to a database and because of that it can't be serialized and also we cant pass the Resultset object from one class to other class across the network.

RowSet is a disconnected, serializable version of a JDBC ResultSet and also the RowSet extends the ResultSet interface so it has all the methods of ResultSet. The RowSet can be serialized because it doesn't have a connection to any database and also it can be sent from one class to another across the network.

|  ResultSet |  RowSet |
|---|--|
| <u>A ResultSet always maintains connection with the database.</u> | A RowSet can be connected, disconnected from the database. |
| <u>It cannot be serialized.</u> | A RowSet object can be serialized. |
| <u>ResultSet object cannot be passed other over network.</u> | You can pass a RowSet object over the network. |

|  ResultSet |  RowSet |
|---|--|
| <u>ResultSet object is not a JavaBean object You can create/get a result set using the executeQuery() method.</u> | ResultSet Object is a JavaBean object. You can get a RowSet using the RowSetProvider.newFactory().createJdbcRowSet() method. |
| <u>By default, ResultSet object is not scrollable or, updatable.</u> | By default, RowSet object is scrollable and updatable. |

Q. How can we execute stored procedures using CallableStatement?

CallableStatement interface in java is used to call stored procedure from java program. **Stored Procedures** are group of statements that we compile in the database for some task. Stored procedures are beneficial when we are dealing with multiple tables with complex scenario and rather than sending multiple queries to the database, we can send required data to the stored procedure and have the logic executed in the database server itself.

A CallableStatement object provides a way to call stored procedures using JDBC. Connection.prepareCall() method provides you CallableStatement object.

```
create or replace procedure "INSERTUSERS"
(id IN NUMBER,
name IN VARCHAR2)
is
begin
insert into users values(id,name);
end;
/
```

```
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;

/**
 *
 * A Simple example to use CallableStatement in Java Program.
 */
public class Proc {

    public static void main(String[] args) throws Exception{
        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            Connection con=DriverManager.getConnection(
                "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
```

```

    } catch (Exception e) {
        e.printStackTrace();
    }

    String SQL = "{call INSERTUSERS(?,?)}";
    CallableStatement stmt = con.prepareCall(SQL);
    stmt.setInt(1,1011);
    stmt.setString(2,"Alex");
    ResultSet rs = stmt.executeQuery();

    while(rs.next()){
        System.out.println(rs.getString(1));
    }

    rs.close();
}
}

```

Q. What are the differences between Statement and PreparedStatement interface?

JDBC API provides 3 different interfaces to execute the different types of SQL queries. They are,

- **Statement** – Used to execute normal SQL queries.
- **PreparedStatement** – Used to execute dynamic or parameterized SQL queries.
- **CallableStatement** – Used to execute the stored procedures.

1. Statement

Statement interface is used to execute normal SQL queries. We can't pass the parameters to SQL query at run time using this interface. This interface is preferred over other two interfaces if we are executing a particular SQL query only once. The performance of this interface is also very less compared to other two interfaces. In most of time, Statement interface is used for DDL statements like **CREATE**, **ALTER**, **DROP** etc.

```

/** Creating The Statement Object */
Statement stmt = con.createStatement();

/** Executing The Statement */
stmt.executeUpdate("CREATE TABLE STUDENT(ID NUMBER NOT NULL, NAME VARCHAR)");

```

2. PreparedStatement

PreparedStatement is used to execute dynamic or parameterized SQL queries. PreparedStatement extends Statement interface. We can pass the parameters to SQL query at run time using this interface. It is recommended to use PreparedStatement if we are executing a particular SQL query multiple times. It gives better performance than Statement interface. Because, PreparedStatement are precompiled and the query plan is created only once irrespective of how many times we are executing that query.

```
/** Creating PreparedStatement object */
PreparedStatement pstmt = con.prepareStatement("update STUDENT set NAME = ? where ID = ?");

/** Setting values to place holders using setter methods of PreparedStatement object */
pstmt.setString(1, "MyName");    /** Assigns "MyName" to first place holder */

pstmt.setInt(2, 111);           /** Assigns "111" to second place holder */

/** Executing PreparedStatement */
pstmt.executeUpdate();
```

3. CallableStatement

CallableStatement is used to execute the stored procedures. CallableStatement extends PreparedStatement. Using CallableStatement, we can pass 3 types of parameters to stored procedures. They are : **IN** – used to pass the values to stored procedure, **OUT** – used to hold the result returned by the stored procedure and **IN OUT** – acts as both IN and OUT parameter. Before calling the stored procedure, we must register OUT parameters using **registerOutParameter()** method of CallableStatement. The performance of this interface is higher than the other two interfaces. Because, it calls the stored procedures which are already compiled and stored in the database server.

```
/** Creating CallableStatement object */
CallableStatement cstmt = con.prepareCall("{call anyProcedure(?, ?, ?)}");

/** Use cstmt.setter() methods to pass IN parameters */

/** Use cstmt.registerOutParameter() method to register OUT parameters */

/** Executing the CallableStatement */

cstmt.execute();



/** Use cstmt.getter() methods to retrieve the result returned by the stored procedure */
```

Q. What are the different types of locking in JDBC?

The types of locks in JDBC:

- 1. Row and Key Locks:** Useful when updating the rows (update, insert or delete operations), as they increase concurrency.
- 2. Page Locks:** Locks the page when the transaction updates or inserts or deletes rows or keys. The database server locks the entire page that contains the row. The lock is made only once by database server, even more rows are updated. This lock is suggested in the situation where large number of rows is to be changed at once.
- 3. Table Locks:** Utilizing table locks is efficient when a query accesses most of the tables of a table. These are of two types:
 - a) Shared lock:** One shared lock is placed by the database server, which prevents other to perform any update operations.
 - b) Exclusive lock:** One exclusive lock is placed by the database server, irrespective of the number of the rows that are updated.
- 4. Database Lock:** In order to prevent the read or update access from other transactions when the database is open, the database lock is used.

Q. What are the differences between Stored Procedure and functions?

|  Functions |  Procedures |
|--|--|
| <u>A function has a return type and returns a value.</u> | A procedure does not have a return type. But it returns values using the OUT parameters. |
| <u>You cannot use a function with Data Manipulation queries. Only Select queries are allowed in functions.</u> | You can use DML queries such as insert, update, select etc... with procedures. |
| <u>A function does not allow output parameters</u> | A procedure allows both input and output parameters. |
| <u>You cannot manage transactions inside a function.</u> | You can manage transactions inside a function. |
| <u>You cannot call stored procedures from a function</u> | You can call a function from a stored procedure. |

| | |
|--|--|
| Aa Functions | ☰ Procedures |
| <u>You can call a function using a select statement.</u> | You cannot call a procedure using select statements. |

Q. What is batch processing and how to perform batch processing in JDBC?

Batch Processing allows to group related SQL statements into a batch and submit them with one call to the database. The `java.sql.Statement` and `java.sql.PreparedStatement` interfaces provide methods for batch processing.

- **addBatch():** The `addBatch()` method of `Statement`, `PreparedStatement`, and `CallableStatement` is used to add individual statements to the batch.
- **executeBatch():** The `executeBatch()` is used to start the execution of all the statements grouped together. The `executeBatch()` returns an array of integers, and each element of the array represents the update count for the respective update statement.
- **clearBatch():** This method removes all the statements added with the `addBatch()` method.

Batching with Statement Object

```
// Create statement object
Statement stmt = conn.createStatement();

// Set auto-commit to false
conn.setAutoCommit(false);

// Create SQL statement
String SQL = "INSERT INTO Employees (id, first, last, age) " +
            "VALUES(200,'Zia', 'Ali', 30)";
// Add above SQL statement in the batch.
stmt.addBatch(SQL);

// Create one more SQL statement
String SQL = "INSERT INTO Employees (id, first, last, age) " +
            "VALUES(201,'Raj', 'Kumar', 35)";
// Add above SQL statement in the batch.
stmt.addBatch(SQL);

// Create one more SQL statement
String SQL = "UPDATE Employees SET age = 35 " +
            "WHERE id = 100";
// Add above SQL statement in the batch.
stmt.addBatch(SQL);

// Create an int[] to hold returned values
```



```
int[] count = stmt.executeBatch();

//Explicitly commit statements to apply changes
conn.commit();
```

Batching with PreparedStatement Object

```
// Create SQL statement
String SQL = "INSERT INTO Employees (id, first, last, age) " +
    "VALUES(?, ?, ?, ?)";

// Create PreparedStatement object
PreparedStatement pstmt = conn.prepareStatement(SQL);

//Set auto-commit to false
conn.setAutoCommit(false);

// Set the variables
pstmt.setInt( 1, 400 );
pstmt.setString( 2, "Pappu" );
pstmt.setString( 3, "Singh" );
pstmt.setInt( 4, 33 );
// Add it to the batch
pstmt.addBatch();

// Set the variables
pstmt.setInt( 1, 401 );
pstmt.setString( 2, "Pawan" );
pstmt.setString( 3, "Singh" );
pstmt.setInt( 4, 31 );
// Add it to the batch
pstmt.addBatch();

//add more batches
.
.
.
.
// Create an int[] to hold returned values
int[] count = stmt.executeBatch();

// Explicitly commit statements to apply changes
conn.commit();
```

Q. What is database connection pooling? What are the advantages of using a connection pool?

Connection pooling means that connections are reused rather than created each time a connection is requested. To facilitate connection reuse, a memory cache of

database connections, called a connection pool, is maintained by a connection pooling module as a layer on top of any standard JDBC driver product.

Connection pooling is performed in the background and does not affect how an application is coded; however, the application must use a DataSource object (an object implementing the DataSource interface) to obtain a connection instead of using the DriverManager class.

JDBC 3.0 API Framework

The JDBC 3.0 API specifies a ConnectionEvent class and the following interfaces as the hooks for any connection pooling implementation:

- ConnectionPoolDataSource
- PooledConnection
- ConnectionEventListener

Example: JDBC Connection Pool

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>JdbcPool</groupId>
  <artifactId>JdbcPool</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>
  <dependencies>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>5.1.41</version>
    </dependency>
    <dependency>
      <groupId>commons-dbcp</groupId>
      <artifactId>commons-dbcp</artifactId>
      <version>1.4</version>
    </dependency>
  </dependencies>
</project>
```

ConnectionPool.java

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
```

```

import javax.sql.DataSource;
import org.apache.commons.dbcp.ConnectionFactory;
import org.apache.commons.dbcp.DriverManagerConnectionFactory;
import org.apache.commons.dbcp.PoolableConnectionFactory;
import org.apache.commons.dbcp.PoolingDataSource;
import org.apache.commons.pool.impl.GenericObjectPool;

public class ConnectionPool {

    // JDBC Driver Name & Database URL
    static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
    static final String JDBC_DB_URL = "jdbc:mysql://localhost:3306/tutorialDb";

    // JDBC Database Credentials
    static final String JDBC_USER = "root";
    static final String JDBC_PASS = "admin@123";

    private static GenericObjectPool gPool = null;

    @SuppressWarnings("unused")
    public DataSource setUpPool() throws Exception {
        Class.forName(JDBC_DRIVER);

        // Creates an Instance of GenericObjectPool That Holds Our Pool of Connections
        Object!
        gPool = new GenericObjectPool();
        gPool.setMaxActive(5);

        // Creates a ConnectionFactory Object Which Will Be Use by the Pool to Create
        the Connection Object!
        ConnectionFactory cf = new DriverManagerConnectionFactory(JDBC_DB_URL, JDBC_US
        ER, JDBC_PASS);

        // Creates a PoolableConnectionFactory That Will Wraps the Connection Object C
        reated by the ConnectionFactory to Add Object Pooling Functionality!
        PoolableConnectionFactory pcf = new PoolableConnectionFactory(cf, gPool, null,
        null, false, true);
        return new PoolingDataSource(gPool);
    }

    public GenericObjectPool getConnectionPool() {
        return gPool;
    }

    // This Method Is Used To Print The Connection Pool Status
    private void printDbStatus() {
        System.out.println("Max.: " + getConnectionPool().getMaxActive() + "; Active:
        " + getConnectionPool().getNumActive() + "; Idle: " + getConnectionPool().getNumIdle
        ());
    }

    public static void main(String[] args) {
        ResultSet rsObj = null;
        Connection connObj = null;
        PreparedStatement pstmtObj = null;
        ConnectionPool jdbcObj = new ConnectionPool();
        try {
            DataSource dataSource = jdbcObj.setUpPool();

```

```

        jdbcObj.printDbStatus();

        // Performing Database Operation!
        System.out.println("\n\n====Making A New Connection Object For Db Transact
ion====\n\n");
        connObj = dataSource.getConnection();
        jdbcObj.printDbStatus();

        pstmtObj = connObj.prepareStatement("SELECT * FROM technical_editors");
        rsObj = pstmtObj.executeQuery();
        while (rsObj.next()) {
            System.out.println("Username: " + rsObj.getString("tech_username"));
        }
        System.out.println("\n\n====Releasing Connection Object To Pool====\n\n");
    } catch (Exception sqlException) {
        sqlException.printStackTrace();
    } finally {
        try {
            // Closing ResultSet Object
            if(rsObj != null) {
                rsObj.close();
            }
            // Closing PreparedStatement Object
            if(pstmtObj != null) {
                pstmtObj.close();
            }
            // Closing Connection Object
            if(connObj != null) {
                connObj.close();
            }
        } catch (Exception sqlException) {
            sqlException.printStackTrace();
        }
    }
    jdbcObj.printDbStatus();
}
}
}

```

Q. What is JDBC Driver?

JDBC Driver is a software component that enables java application to interact with the database. There are 4 types of JDBC drivers:

- JDBC-ODBC bridge driver
- Native-API driver (partially java driver)
- Network Protocol driver (fully java driver)
- Thin driver (fully java driver)

1. JDBC-ODBC bridge driver

The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. This is now discouraged because of thin driver.

Oracle does not support the JDBC-ODBC Bridge from Java 8. Oracle recommends that you use JDBC drivers provided by the vendor of your database instead of the JDBC-ODBC Bridge.

2. Native-API driver

The Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API. It is not written entirely in java.

3. Network Protocol driver

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java.

4. Thin driver

The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java language.

Q. What are the JDBC API components?

The JDBC API provides the following interfaces and classes –

- **DriverManager:** This class manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication sub protocol. The first driver that recognizes a certain subprotocol under JDBC will be used to establish a database Connection.
- **Driver:** This interface handles the communications with the database server. You will interact directly with Driver objects very rarely. Instead, you use DriverManager objects, which manages objects of this type. It also abstracts the details associated with working with Driver objects.
- **Connection:** This interface with all methods for contacting a database. The connection object represents communication context, i.e., all communication with database is through connection object only.
- **Statement:** You use objects created from this interface to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.

- **ResultSet:** These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.
- **SQLException:** This class handles any errors that occur in a database application.

Q. What is JDBC ResultSetMetaData interface?

ResultSetMetaData is an interface in **java.sql** package of JDBC API which is used to get the metadata about a ResultSet object. Whenever we query the database using SELECT statement, the result will be stored in a ResultSet object. Every ResultSet object is associated with one ResultSetMetaData object. This object will have all the meta data about a ResultSet object like schema name, table name, number of columns, column name, datatype of a column etc. We can get this ResultSetMetaData object using **getMetaData()** method of ResultSet.

```
import java.sql.*;
class Rsmd {

    public static void main(String args[]) {

        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            Connection con = DriverManager.getConnection(
                "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");

            PreparedStatement ps = con.prepareStatement("select * from emp");
            ResultSet rs = ps.executeQuery();
            ResultSetMetaData rsmd = rs.getMetaData();

            System.out.println("Total columns: "+rsmd.getColumnCount());
            System.out.println("Column Name of 1st column: "+rsmd.getColumnName(1));
            System.out.println("Column Type Name of 1st column: "+rsmd.getColumnTypeName
(1));



            con.close();
        } catch (Exception e){
            System.out.println(e);
        }
    }
}
```

Output

```
Total columns: 2
Column Name of 1st column: ID
```

Q. What is JDBC DatabaseMetaData interface?

DatabaseMetaData interface provides methods to get meta data of a database such as database product name, database product version, driver name, name of total number of tables, name of total number of views etc.

|  Methods |  Description |
|---|---|
| <u>getDriverName():</u> | It returns the name of the JDBC driver. |
| <u>getDriverVersion():</u> | It returns the version number of the JDBC driver. |
| <u>getUserName():</u> | It returns the username of the database. |
| <u>getDatabaseProductName():</u> | It returns the product name of the database. |
| <u>getDatabaseProductVersion():</u> | It returns the product version of the database. |
| <u>getTables():</u> | It returns the description of the tables of the specified catalog. The table type can be TABLE, VIEW, ALIAS, SYSTEM TABLE, SYNONYM etc. |

```
import java.sql.*;
class Dbmd {

    public static void main(String args[]) {

        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");

            Connection con = DriverManager.getConnection(
                "jdbc:oracle:thin:@localhost:1521:xe", "system", "oracle");
            DatabaseMetaData dbmd = con.getMetaData();

            System.out.println("Driver Name: "+dbmd.getDriverName());
            System.out.println("Driver Version: "+dbmd.getDriverVersion());
            System.out.println("UserName: "+dbmd.getUserName());
            System.out.println("Database Product Name: "+dbmd.getDatabaseProductName());
            System.out.println("Database Product Version: "+dbmd.getDatabaseProductVersion
            ());

            con.close();
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

Output

```
Driver Name: Oracle JDBC Driver
Driver Version: 10.2.0.1.0XE
Database Product Name: Oracle
Database Product Version: Oracle Database 10g Express Edition
Release 10.2.0.1.0 -Production
```

Q. How can we set null value in JDBC PreparedStatement?

Use the **setNull()** method to bind null to the parameter. The setNull() method accepts two parameter, index and the sql type as arguments.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;

public class Main {

    public static void main(String[] args) throws Exception {

        Connection conn = getConnection();
        PreparedStatement pstmt = null;

        conn = getConnection();
        String query = "insert into nullable_table(id,string_column, int_column) values(?,
?, ?)";

        // create PreparedStatement object
        pstmt = conn.prepareStatement(query);
        pstmt.setString(1, "0001");
        pstmt.setNull(2, java.sql.Types.VARCHAR);
        pstmt.setNull(3, java.sql.Types.INTEGER);

        // execute query, and return number of rows created
        int rowCount = pstmt.executeUpdate();
        System.out.println("rowCount=" + rowCount);

        conn.close();
    }

    private static Connection getConnection() throws Exception {
        Class.forName("org.hsqldb.jdbcDriver");
        String url = "jdbc:hsqldb:mem:data/tutorial";
        return DriverManager.getConnection(url, "sa", "");
    }
}
```


Some more JDBC Questions you can prepare.

Q. What are the differences between execute, executeQuery, and executeUpdate?

Q. Which interface is responsible for transaction management in JDBC?

Q. How can we maintain the integrity of a database by using JDBC?

Q. What is the major difference between java.util.Date and java.sql.Date data type?

Q. How do you handle error condition while writing stored procedure or accessing stored procedure from java?

Q. How do you iterate ResultSet in the reverse order?

Q. What is the use of setAutoCommit() method?

Q. What is a “dirty read”?

Q. What are the ways to load or register driver?

Q. How to get the Database server details in java program?

Q. What is the use of getGeneratedKeys() method in Statement?

Q. What is the use of setFetchSize() and setMaxRows() methods in Statement?

Q. What is JDBC Savepoint? How to use it?

Q. What are CLOB and BLOB data types in JDBC?

Q. Explain optimistic and pessimistic locking in JDBC?

Q. How can we store and retrieve images from the database?

Q. How can we store and retrieve the file in the Oracle database?