**JDBC (Java Database Connectivity) Complete Notes**

**What is JDBC?**

* JDBC stands for **Java Database Connectivity**.
* It is a **Java API** used to connect and execute queries with a database.
* JDBC is part of the Java Standard Edition platform and provides methods to interact with relational databases such as MySQL, PostgreSQL, Oracle, and others.

**Architecture of JDBC**

JDBC consists of two main components:

1. **JDBC API**:
   * A set of interfaces and classes in java.sql and javax.sql packages.
   * Handles database connections, SQL execution, and result processing.
2. **JDBC Driver**:
   * A software component that translates Java calls into database-specific calls.
   * Types of drivers:
     + **Type 1**: JDBC-ODBC Bridge Driver
     + **Type 2**: Native-API Driver
     + **Type 3**: Network Protocol Driver
     + **Type 4**: Thin Driver (most commonly used)

**Steps to Connect a Java Program to a Database**

1. **Import JDBC Package**:

import java.sql.\*;

1. **Load and Register the Driver**:

Class.forName("com.mysql.cj.jdbc.Driver"); // For MySQL

1. **Establish Connection**:

Connection con = DriverManager.getConnection(

"jdbc:mysql://localhost:3306/db\_name", "username", "password");

1. **Create a Statement**:

Statement stmt = con.createStatement();

1. **Execute SQL Queries**:

ResultSet rs = stmt.executeQuery("SELECT \* FROM table\_name");

1. **Process the Results**:

while (rs.next()) {

System.out.println(rs.getInt(1) + " " + rs.getString(2));

}

1. **Close the Connection**:

con.close();

**JDBC Core Components**

**1. DriverManager**

* Manages database drivers.
* Provides a connection to the database.
* Example:

Connection con = DriverManager.getConnection(url, username, password);

**2. Connection**

* Represents the session with the database.
* Methods:
  + createStatement(): Creates a statement object.
  + prepareStatement(): Creates a precompiled SQL statement.
  + close(): Closes the connection.

**3. Statement**

* Executes SQL queries.
* Types:
  + **Statement**: For simple queries.
  + **PreparedStatement**: For parameterized queries.
  + **CallableStatement**: For executing stored procedures.

**4. ResultSet**

* Represents the result of a query.
* Provides methods to navigate and retrieve data.
* Example:

while (rs.next()) {

int id = rs.getInt("id");

String name = rs.getString("name");

}

**JDBC Statements**

**1. Statement**

* Used for executing static SQL queries.
* Example:

Statement stmt = con.createStatement();

ResultSet rs = stmt.executeQuery("SELECT \* FROM users");

**2. PreparedStatement**

* Used for precompiled SQL queries with dynamic parameters.
* Prevents SQL injection.
* Example:

PreparedStatement pstmt = con.prepareStatement("SELECT \* FROM users WHERE id = ?");

pstmt.setInt(1, 101);

ResultSet rs = pstmt.executeQuery();

**3. CallableStatement**

* Used for executing stored procedures.
* Example:

CallableStatement cstmt = con.prepareCall("{call getUser(?)}");

cstmt.setInt(1, 101);

ResultSet rs = cstmt.executeQuery();

**JDBC Exceptions**

* **SQLException** is the primary exception class in JDBC.
* Common methods in SQLException:
  + getMessage(): Retrieves the error message.
  + getErrorCode(): Retrieves the vendor-specific error code.
  + getSQLState(): Retrieves the SQL state string.

**Transaction Management in JDBC**

* Transactions are used to group multiple operations into a single logical unit.
* By default, JDBC auto-commits each SQL statement.

**Methods:**

1. Disable Auto-Commit:
2. con.setAutoCommit(false);
3. Commit a Transaction:
4. con.commit();
5. Rollback a Transaction:
6. con.rollback();

**Example:**

try {

con.setAutoCommit(false);

Statement stmt = con.createStatement();

stmt.executeUpdate("INSERT INTO accounts VALUES (1, 'Alice', 1000)");

stmt.executeUpdate("UPDATE accounts SET balance = balance - 100 WHERE id = 1");

con.commit();

} catch (SQLException e) {

con.rollback();

e.printStackTrace();

}

**Batch Processing in JDBC**

* Used to execute multiple SQL statements in one batch for efficiency.

**Steps**:

1. Add queries to the batch:
2. stmt.addBatch("INSERT INTO users VALUES (1, 'Alice')");
3. stmt.addBatch("INSERT INTO users VALUES (2, 'Bob')");
4. Execute the batch:
5. int[] results = stmt.executeBatch();
6. Handle results:
7. for (int result : results) {
8. System.out.println("Rows affected: " + result);
9. }

**JDBC Drivers**

| **Type** | **Description** | **Advantages** | **Disadvantages** |
| --- | --- | --- | --- |
| Type 1 | JDBC-ODBC Bridge | Easy to use | Dependent on ODBC; not portable |
| Type 2 | Native-API | Faster than Type 1 | Platform-dependent |
| Type 3 | Network Protocol | Portable | Requires a middleware server |
| Type 4 | Thin Driver (pure Java driver) | Platform-independent; faster | None |

**Key Features of JDBC**

1. **Portability**: Works across all platforms that support Java.
2. **Supports SQL**: Works with any relational database supporting SQL.
3. **Integration**: Can be used with other APIs like Hibernate, JPA, etc.
4. **Extensibility**: Supports custom drivers for additional databases.

**Common JDBC Operations**

**Inserting Data:**

PreparedStatement pstmt = con.prepareStatement("INSERT INTO users (id, name) VALUES (?, ?)");

pstmt.setInt(1, 101);

pstmt.setString(2, "John");

pstmt.executeUpdate();

**Updating Data:**

PreparedStatement pstmt = con.prepareStatement("UPDATE users SET name = ? WHERE id = ?");

pstmt.setString(1, "Jane");

pstmt.setInt(2, 101);

pstmt.executeUpdate();

**Deleting Data:**

PreparedStatement pstmt = con.prepareStatement("DELETE FROM users WHERE id = ?");

pstmt.setInt(1, 101);

pstmt.executeUpdate();

**Best Practices for JDBC**

1. Always close Connection, Statement, and ResultSet to free resources.
2. Use try-with-resources for auto-closing resources.
3. Use PreparedStatement to prevent SQL injection.
4. Batch updates for bulk operations.
5. Use connection pooling for better performance in production.