### ***JDBC NOTES***

### **🚀 Welcome to JDBC: Your Gateway to Databases!**

Have you ever wondered how websites store our login details, shopping carts, or even high scores in a game like stuffs ? That’s where **databases** come in! Exactly In databases we store these huge data . But how does a Java program talk to a database like MySQL?

**JDBC (Java Database Connectivity)** – is the **bridge** 🏗️ that allows Java applications to connect with databases, store information, and retrieve it whenever needed.

Think of JDBC as a **universal translator** 🗣️ that helps Java and MySQL understand each other. Whether you’re building a simple login system or a full-fledged e-commerce platform, JDBC is the key 🔑 to making it happen!

In this guide, we’ll start from the basics and gradually move towards advanced concepts so that, by the end, you’ll be confidently working with Java and MySQL like a pro! 💡🔥

### **🔹 What is JDBC? (Simple Explanation)**

* **JDBC (Java Database Connectivity)** is a Java API that allows Java applications to interact with databases.
* It acts as a **bridge** between Java programs and databases like MySQL, PostgreSQL, or Oracle.
* Using JDBC, Java applications can **store, retrieve, update, and delete data** from databases.

### **🔹 Why Do We Need JDBC with MySQL?**

* Java programs cannot communicate with databases **directly**—JDBC provides the necessary connection.
* It allows developers to **store user data, retrieve records, and manage databases dynamically**.
* Works with **multiple databases**, but MySQL is one of the most popular choices due to its speed and open-source nature.

**Real-World Applications of JDBC & MySQL**

✅ **Web Applications** – Storing user accounts, login credentials, and preferences.  
✅ **E-Commerce Platforms** – Managing product catalogs, orders, and payments.  
✅ **Banking Systems** – Handling customer accounts, transactions, and reports.  
✅ **Enterprise Software** – Storing employee records, attendance, and payroll information.

**Key points about JDBC include:**

1. Connectivity: It allows Java applications to connect to a database using a standard interface.
2. Drivers: JDBC requires a driver to connect to the database. There are four types of JDBC drivers:
   * Type 1: JDBC-ODBC Bridge Driver
   * Type 2: Native-API Driver
   * Type 3: Network Protocol Driver
   * Type 4: Thin Driver
3. SQL Execution: JDBC provides a standard mechanism to execute SQL queries and retrieve results.
4. Database Independence: It allows developers to write database-independent Java applications, meaning the same code can work with different databases with minimal changes.
5. API Components: The main components of JDBC are:
   * DriverManager: Manages a list of database drivers.
   * Connection: Represents a connection to a specific database.
   * Statement: Used to execute the SQL queries.
   * ResultSet: Represents the result set of a query.
   * PreparedStatement: Extends Statement for SQL statements that are executed multiple times.
   * CallableStatement: Used to execute stored procedures in the database.

JDBC is widely used in enterprise applications where interaction with databases is required.

### **SET UP**

### **🔹 Step 1: Install MySQL Database**

* Download MySQL from the official site: [**MySQL Download**](https://dev.mysql.com/downloads/installer/)
* Follow the installation steps and **remember your root password**.
* Start MySQL Server and open **MySQL Workbench** or **Command Line Client**.

### **🔹 Step 2: Create a Sample Database & Table**

Open MySQL Workbench or CLI and run the following SQL commands:

| **CREATE** **DATABASE** database1; **USE** database1;  **CREATE** **TABLE** **users** (  **id** INT AUTO\_INCREMENT PRIMARY **KEY**,  **name** VARCHAR(50),  email VARCHAR(100) ); |
| --- |

* This creates a **database (database1)** and a **table (users)** to store user details.

### **🔹 Step 3: Download MySQL JDBC Driver**

* JDBC requires a **driver** to connect Java with MySQL.
* Download the **MySQL Connector/J** from: [**MySQL Connector/J**](https://dev.mysql.com/downloads/connector/j/)
* Add the JAR file to your Java project:
  + **IntelliJ:** File → Project Structure → Libraries → Add JAR
  + **Eclipse:** Right-click project → Build Path → Add External JARs

Create a **Java project** in your IDE (**IntelliJ/Eclipse**).

Add the **MySQL Connector/J JAR** to the project.

* + **VSCode:**

1.Extract the .zip file and copy **mysql-connector-java-<version>.jar**

2.**Open VS Code** and create a **new Java project** (File → Open Folder).

3.**Add a new folder** in your project named lib/ and place the **MySQL JDBC JAR** inside.

4.**Create a new Java file** (Main.java).

5.Open **settings.json** (Ctrl + Shift + P → Search "Settings JSON") and ensure Java is configured properly.

6.Add the JAR file to the **classpath**:

7.

| **json**  "java.project.referencedLibraries": [  "lib/mysql-connector-java-<version>.jar" ] |
| --- |

8.Open Main.java and write:

### **🔹 Step 4: Connect DataBase**

Write a simple Java program to **test the connection**:

| **import** java.sql.\*; **public** class MySQLConnectionTest {  **public** **static** **void** main(**String**[] args) {  **String** url = "jdbc:mysql://localhost:3306/database1";  **String** user = "root";  **String** password = "yourpassword";  try (Connection con = DriverManager.getConnection(url, user, password)) {  System.out.println("Connected to MySQL successfully!");  } catch (SQLException e) {  e.printStackTrace();  }  } } |
| --- |

* Run the program! If successful, it will print:  
   **"Connected to MySQL successfully!"** 🎉

This completes the basic **JDBC & MySQL setup**! 🚀

### **DriverManager and Connection Objects in JDBC**

When working with **JDBC (Java Database Connectivity)**, we need a way to establish a connection between our **Java application** and a **database** (like MySQL). This is where **DriverManager** and **Connection** objects come in.

## **1️⃣ DriverManager (Class) – Manages Database Drivers**

* DriverManager is a **built-in JDBC class** that **loads the database driver** and **establishes a connection** to the database.
* It **selects the appropriate driver** based on the JDBC URL provided in getConnection().

### **Example Usage of DriverManager**

Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/mydatabase", "root", "password");

💡 **What happens here?**

* DriverManager checks for the **MySQL JDBC driver**.
* It uses the JDBC **URL, username, and password** to connect to the database.
* If successful, it returns a **Connection object** to interact with the database.

## **2️⃣ Connection (Interface) – Represents the Database Connection**

* Connection is an **interface** that represents an **active connection** to the database.
* Once connected, it allows Java to **send queries, retrieve data, and perform transactions**.
* The Connection object is used to **create statements, prepare queries, and manage transactions**.

### **Example Usage of Connection**

| try {  Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/mydatabase", "root", "password");  System.out.println("Connected to the database!");    // Performing database operations here...   con.close(); // Always close the connection! } catch (SQLException e) {  e.printStackTrace(); } |
| --- |

💡 **Key Points:** ✔️ DriverManager.getConnection() returns a **Connection object**.  
 ✔️ We use this object to **execute queries** and **manage transactions**.  
 ✔️ Always **close the connection** (con.close()) to free up resources.

## **🔹 Summary**

| **Feature** | **DriverManager** | **Connection** |
| --- | --- | --- |
| Type | Class | Interface |
| Purpose | Loads the JDBC driver & manages connections | Represents an active connection to the database |
| Key Method | getConnection() | createStatement(), prepareStatement(), close() |
| Usage | DriverManager.getConnection(url, user, password); | con.createStatement().executeQuery("SELECT \* FROM users"); |

These two objects **work together** to enable database interactions in Java applications! 🚀

### **Performing CRUD Operations with JDBC**

Once we have established a **database connection**, the next step is to **perform CRUD operations** using JDBC. CRUD stands for:  
 ✅ **C** - Create (Insert data)  
 ✅ **R** - Read (Retrieve data)  
 ✅ **U** - Update (Modify existing data)  
 ✅ **D** - Delete (Remove data)

Let’s go through each operation step by step using JDBC with **MySQL**.

## **1️⃣ CREATE – Insert Data into Database**

To insert data into a table, we use the **INSERT INTO** SQL statement.

### **Example: Insert a New User**

| **import** java.sql.\*;  **public** class InsertData {  **public** **static** **void** main(**String**[] args) {  **String** url = "jdbc:mysql://localhost:3306/mydatabase";  **String** user = "root";  **String** password = "yourpassword";   **String** sql = "INSERT INTO users (name, email) VALUES (?, ?)";   try (Connection con = DriverManager.getConnection(url, user, password);  PreparedStatement pstmt = con.prepareStatement(sql)) {   pstmt.setString(1, "Alice");  pstmt.setString(2, "alice@example.com");  pstmt.executeUpdate();   System.out.println("User inserted successfully!");   } catch (SQLException e) {  e.printStackTrace();  }  } } |
| --- |

✅ **Key Points:**

* Use **PreparedStatement** to prevent SQL injection.
* setString(1, "Alice") replaces ? with actual values.
* executeUpdate() is used for **INSERT, UPDATE, DELETE** operations.

## **2️⃣ READ – Retrieve Data from Database**

To fetch records from a table, we use the **SELECT** SQL statement.

### **Example: Fetch & Display Users**

| **import** java.sql.\*;  **public** class ReadData {  **public** **static** **void** main(**String**[] args) {  **String** url = "jdbc:mysql://localhost:3306/mydatabase";  **String** user = "root";  **String** password = "yourpassword";   **String** sql = "SELECT \* FROM users";   try (Connection con = DriverManager.getConnection(url, user, password);  Statement stmt = con.createStatement();  ResultSet rs = stmt.executeQuery(sql)) {   while (rs.next()) {  **int** id = rs.getInt("id");  **String** name = rs.getString("name");  **String** email = rs.getString("email");  System.out.println("ID: " + id + ", Name: " + name + ", Email: " + email);  }   } catch (SQLException e) {  e.printStackTrace();  }  } } |
| --- |

✅ **Key Points:**

* executeQuery() is used for **SELECT** queries.
* ResultSet stores and processes the retrieved data.
* rs.next() moves to the next row in the result set.

## **3️⃣ UPDATE – Modify Existing Records**

To update records, we use the **UPDATE** SQL statement.

### **Example: Update User Email**

| **import** java.sql.\*;  **public** class UpdateData {  **public** **static** **void** main(**String**[] args) {  **String** url = "jdbc:mysql://localhost:3306/mydatabase";  **String** user = "root";  **String** password = "yourpassword";   **String** sql = "UPDATE users SET email = ? WHERE name = ?";   try (Connection con = DriverManager.getConnection(url, user, password);  PreparedStatement pstmt = con.prepareStatement(sql)) {   pstmt.setString(1, "newalice@example.com");  pstmt.setString(2, "Alice");  **int** rowsUpdated = pstmt.executeUpdate();   if (rowsUpdated > 0) {  System.out.println("User updated successfully!");  } else {  System.out.println("No user found with that name.");  }   } catch (SQLException e) {  e.printStackTrace();  }  } } |
| --- |

✅ **Key Points:**

* Use **UPDATE** to modify records.
* executeUpdate() returns the **number of rows affected**.

## **4️⃣ DELETE – Remove Data from Database**

To delete records, we use the **DELETE** SQL statement.

### **Example: Delete a User**

| **import** java.sql.\*;  **public** class DeleteData {  **public** **static** **void** main(**String**[] args) {  **String** url = "jdbc:mysql://localhost:3306/mydatabase";  **String** user = "root";  **String** password = "yourpassword";   **String** sql = "DELETE FROM users WHERE name = ?";   try (Connection con = DriverManager.getConnection(url, user, password);  PreparedStatement pstmt = con.prepareStatement(sql)) {   pstmt.setString(1, "Alice");  **int** rowsDeleted = pstmt.executeUpdate();   if (rowsDeleted > 0) {  System.out.println("User deleted successfully!");  } else {  System.out.println("No user found with that name.");  }   } catch (SQLException e) {  e.printStackTrace();  }  } } |
| --- |

✅ **Key Points:**

* **DELETE** removes records permanently.
* Always be careful when using **DELETE** without a **WHERE** condition to avoid deleting all records.

### **🔹 Summary of CRUD Operations**

| **Operation** | **SQL Command** | **JDBC Method** |
| --- | --- | --- |
| **Create** | INSERT INTO table\_name VALUES (...) | executeUpdate() |
| **Read** | SELECT \* FROM table\_name | executeQuery() |
| **Update** | UPDATE table\_name SET column = value WHERE condition | executeUpdate() |
| **Delete** | DELETE FROM table\_name WHERE condition | executeUpdate() |

### **📌 Final Notes:**

🔹 **Always close database connections** after performing operations.  
 🔹 Use **Prepared Statements** to prevent SQL Injection.  
 🔹 Handle **exceptions properly** using try-catch blocks.

Now you can **perform CRUD operations** in Java using JDBC & MySQL! 🚀

### **Advanced Concepts (Moving to Pro Level)**

Once you're comfortable with basic CRUD operations, it's time to explore **advanced JDBC concepts** that improve security, efficiency, and scalability.

### **1️⃣ Using Prepared Statements (Preventing SQL Injection) 🔒**

* **Prepared Statements** prevent SQL Injection by separating SQL queries from user input.
* Unlike regular statements, placeholders (?) are used to bind values safely.

**Example:**

| String sql = "SELECT \* FROM users WHERE email = ?"; PreparedStatement pstmt = con.prepareStatement(sql); pstmt.setString(1, "user@example.com"); ResultSet rs = pstmt.executeQuery(); |
| --- |

✅ **Prevents attackers from injecting harmful SQL code**.

### **2️⃣ Transaction Management (Commit & Rollback) 🔄**

* Transactions **group multiple operations** into a single unit.
* If one operation fails, all changes can be **rolled back** to maintain database integrity.

**Example:**

| **con**.setAutoCommit(false); // Start transaction **try** {  **stmt**.executeUpdate("UPDATE accounts SET balance = balance - 500 WHERE id = 1");  **stmt**.executeUpdate("UPDATE accounts SET balance = balance + 500 WHERE id = 2");  **con**.commit(); // Commit transaction } **catch** (SQLException e) {  **con**.rollback(); // Rollback if an error occurs } |
| --- |

✅ Ensures **data consistency** in critical operations (e.g., banking transactions).

### **3️⃣ Connection Pooling (Boosting Performance) 🚀**

* **Opening and closing connections repeatedly is slow**.
* **Connection pooling** reuses database connections, improving speed in **large-scale applications**.
* **HikariCP** and **Apache DBCP** are popular JDBC connection pool libraries.

**Example (Using HikariCP):**

| HikariConfig config = new HikariConfig(); config.setJdbcUrl("jdbc:mysql://localhost:3306/mydatabase"); config.setUsername("root"); config.setPassword("password"); HikariDataSource dataSource = new HikariDataSource(config); |
| --- |

✅ **Reduces latency** and **improves efficiency** in high-traffic apps.

### **4️⃣ Using JDBC with ORM Frameworks (Intro to Hibernate) 🏗️**

* **JDBC requires writing SQL manually**, but **ORM (Object-Relational Mapping) frameworks** like **Hibernate** simplify this.
* Hibernate **maps Java objects to database tables**, reducing boilerplate code.

**Example (Hibernate Entity Class):**

| @Entity @Table(name = "users") public class User {  @Id @GeneratedValue(strategy = GenerationType.IDENTITY)  private int id;  **private** **String** **name**;  **private** **String** **email**; } |
| --- |

✅ **Less SQL, more object-oriented programming!**

### **🔹 Summary**

| **Concept** | **Benefit** |
| --- | --- |
| **Prepared Statements** | Prevents SQL Injection |
| **Transaction Management** | Ensures data consistency with Commit & Rollback |
| **Connection Pooling** | Improves performance by reusing connections |
| **JDBC + ORM (Hibernate)** | Reduces SQL code, making development easier |

Master these concepts to **write secure, scalable, and high-performance JDBC applications!** 🚀

### **Best Practices & Common Mistakes**

To write **efficient, secure, and maintainable** JDBC applications, follow these best practices:

### **✅ Best Practices**

✔️ **Always Close the Database Connection** 🔄

* Open connections **consume resources**; always close them after use.
* Use con.close() to release resources.

✔️ **Use Try-With-Resources (Automatic Resource Management)** 🛠️

* Ensures **connections, statements, and result sets** are **closed automatically**.

**Example:**

| **try** (Connection con = DriverManager.getConnection(url, user, password);  PreparedStatement pstmt = con.prepareStatement("SELECT \* FROM users")) {  ResultSet rs = pstmt.executeQuery();  **while** (rs.**next**()) {  System.out.**println**(rs.getString("name"));  } } **catch** (SQLException e) {  e.printStackTrace(); } |
| --- |

✅ No need for manual close() calls!

✔️ **Use Environment Variables or Properties Files for Credentials** 🔐

* **Never hardcode credentials** in code—it’s a security risk!
* Store them in a .properties file or environment variables.

**Example (Loading from a Properties File):**

| Properties props = new Properties(); props.load(new FileInputStream("config.properties")); String dbUser = props.getProperty("db.user"); String dbPassword = props.getProperty("db.password"); |
| --- |

✅ Protects **sensitive information** and makes configuration flexible.

### **❌ Common Mistakes to Avoid**

🚫 **Not closing connections** → Can lead to **memory leaks**.  
🚫 **Using plain Statement instead of PreparedStatement** → **SQL Injection risk**.  
🚫 **Hardcoding credentials** → Security vulnerability.

By following these **best practices**, you ensure your JDBC application is **secure, efficient, and scalable**! 🚀

### **🔸 Conclusion**

In this guide, we covered the basics of **JDBC (Java Database Connectivity)**, including:

* **Connecting to MySQL** with DriverManager and Connection objects.
* Performing **CRUD operations** (Create, Read, Update, Delete) using SQL statements.
* Advanced concepts like **Prepared Statements**, **Transaction Management**, **Connection Pooling**, and integrating JDBC with **ORM frameworks** like **Hibernate**.
* Best practices for **closing connections**, using **try-with-resources**, and securely handling **database credentials**.

### **🚀 Next Steps**

Now that you've learned the basics of JDBC, it's time to **try the code yourself**! Set up your MySQL database, run the examples, and start exploring more advanced topics.

Next learning steps:

* **Hibernate**: Learn how to use **ORM** to simplify database operations and map Java objects to database tables.
* **Spring JDBC**: Use the **Spring Framework** to manage database operations more easily and efficiently.

By mastering these concepts, you'll be ready to develop **secure, scalable, and efficient** database-driven Java applications! 🚀

**SUMMARY**

**JDBC Key Points for Viva (With Explanations)**

### **1️⃣ What is JDBC?**

* **JDBC (Java Database Connectivity)** is an API in Java that allows applications to connect and interact with databases.
* It provides a standard way to execute SQL queries from Java programs.
* **Example:** Fetching user details from a MySQL database using SQL queries.

### **2️⃣ Why Do We Need JDBC?**

* Java applications cannot directly interact with databases. JDBC acts as a **bridge** between Java code and databases.
* It enables operations like **inserting, updating, deleting, and retrieving data**.
* **Example:** A web application storing user details in MySQL using JDBC.

### **3️⃣ JDBC Architecture (Components of JDBC)**

1. **JDBC API** - Interfaces like Connection, Statement, ResultSet, etc.
2. **JDBC Driver** - Converts Java calls into database-specific calls.
3. **Database** - The storage system (e.g., MySQL, PostgreSQL, Oracle).

**Flow:** Java Application → JDBC API → JDBC Driver → Database → Response

### **4️⃣ Types of JDBC Drivers**

1. **Type 1 (JDBC-ODBC Bridge Driver)** – Uses ODBC driver (slow, not recommended).
2. **Type 2 (Native-API Driver)** – Uses database-specific native APIs.
3. **Type 3 (Network Protocol Driver)** – Uses middleware server for database communication.
4. **Type 4 (Thin Driver / Pure Java Driver)** – Directly interacts with the database (Best for modern applications).

✅ **MySQL uses Type 4 driver (com.mysql.cj.jdbc.Driver).**

### **5️⃣ Establishing a Database Connection in JDBC**

| **To** connect Java with MySQL: Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/mydatabase", "root", "password"); |
| --- |

✅ **DriverManager** loads the driver and establishes a connection.

### **6️⃣ JDBC CRUD Operations**

✅ **Create (Insert Data)**

| PreparedStatement pstmt = con.prepareStatement("INSERT INTO users (name, email) VALUES (?, ?)"); pstmt.setString(1, "Alice"); pstmt.setString(2, "alice@example.com"); pstmt.executeUpdate(); |
| --- |

✅ **Read (Fetch Data)**

| **Statement** stmt = con.createStatement(); **ResultSet** rs = stmt.executeQuery("SELECT \* FROM users"); **while** (rs.next()) {  **System**.**out**.println(rs.getString("name")); } |
| --- |

✅ **Update (Modify Data)**

| PreparedStatement pstmt = con.prepareStatement("UPDATE users SET email = ? WHERE name = ?"); pstmt.setString(1, "newalice@example.com"); pstmt.setString(2, "Alice"); pstmt.executeUpdate(); |
| --- |

✅ **Delete (Remove Data)**

| PreparedStatement pstmt = con.prepareStatement("**DELETE** FROM users WHERE name = ?"); pstmt.setString(1, "Alice"); pstmt.executeUpdate(); |
| --- |

### **7️⃣ Statement vs. PreparedStatement vs. CallableStatement**

| **Type** | **Use Case** | **SQL Injection Protection** | **Performance** |
| --- | --- | --- | --- |
| Statement | Simple SQL queries | ❌ No | 🔴 Slower |
| PreparedStatement | Parameterized queries | ✅ Yes | 🟢 Faster |
| CallableStatement | Stored procedures | ✅ Yes | 🟢 Optimized |

✅ **Use PreparedStatement to prevent SQL Injection.**

### **8️⃣ ResultSet and Its Types**

* **ResultSet** stores the result of an executed query.
* Types:
  + **Forward-Only (TYPE\_FORWARD\_ONLY)** – Default, can only move forward.
  + **Scroll-Sensitive (TYPE\_SCROLL\_SENSITIVE)** – Reflects real-time DB changes.
  + **Scroll-Insensitive (TYPE\_SCROLL\_INSENSITIVE)** – Does not reflect changes.

✅ rs.next() moves the cursor to the next record.

### **9️⃣ Using Transactions (Commit & Rollback)**

✅ **By default, each query is auto-committed. To control transactions:**

| **con**.setAutoCommit(false); // Disable auto-commit  **try** {  **stmt**.executeUpdate("UPDATE accounts SET balance = balance - 500 WHERE id = 1");  **stmt**.executeUpdate("UPDATE accounts SET balance = balance + 500 WHERE id = 2");  **con**.commit(); // Commit both queries } **catch** (SQLException e) {  **con**.rollback(); // Undo changes if an error occurs } |
| --- |

✅ **Ensures data consistency (e.g., banking transactions).**

### **🔟 Connection Pooling (Performance Optimization)**

* **Instead of opening/closing connections repeatedly, reuse existing ones.**
* Improves performance in high-load applications.
* **Example (Using HikariCP):**

| HikariConfig config = new HikariConfig(); config.setJdbcUrl("jdbc:mysql://localhost:3306/mydatabase"); config.setUsername("root"); config.setPassword("password"); HikariDataSource ds = new HikariDataSource(config); |
| --- |

✅ **Efficient for large-scale applications.**

### **1️⃣1️⃣ Integrating JDBC with Hibernate (ORM Frameworks)**

* **JDBC requires manual SQL writing**, while **Hibernate** automates database operations.
* Uses **Entity Classes** to map Java objects to database tables.
* **Example (Hibernate Entity Class):**

| @Entity @Table(name = "users") public class User {  @Id @GeneratedValue(strategy = GenerationType.IDENTITY)  private int id;  **private** **String** **name**;  **private** **String** **email**; } |
| --- |

✅ **Less SQL, more Java-oriented development.**

### **1️⃣2️⃣ Best Practices & Common Mistakes**

✅ **Best Practices:**

* Always **close the database connection** (con.close()).
* Use **try-with-resources** for automatic connection closing.
* Store **database credentials securely** in environment variables or .properties files.
* Use **connection pooling** for better performance.
* Always use **Prepared Statements** to prevent SQL injection.

🚫 **Common Mistakes:**

* Forgetting to **close connections** → Memory leaks.
* Using **plain Statement instead of PreparedStatement** → Security risk.
* Hardcoding **database credentials** in the code.

### **📝 Quick Revision Table for Viva**

| **Concept** | **Explanation** |
| --- | --- |
| **JDBC** | Java API for database interaction |
| **DriverManager** | Loads database drivers |
| **Connection** | Establishes a link with DB |
| **Statement** | Executes SQL queries |
| **PreparedStatement** | Prevents SQL injection |
| **ResultSet** | Stores query results |
| **Transaction Management** | Ensures data consistency |
| **Connection Pooling** | Reuses connections for performance |
| **Hibernate** | Simplifies database operations |

Now you're **fully prepared** for your viva! 🎤🔥 Let me know if you need more details! 🚀

## **JDBC Interview Questions (Bonus for Viva) 🎤**

### **1️⃣ What is JDBC, and why is it used?**

✅ **JDBC (Java Database Connectivity)** is an API that allows Java applications to interact with databases.  
✅ It is used to **execute SQL queries**, **retrieve data**, **update records**, and **manage transactions**.  
✅ It provides a standard interface for connecting different types of databases (e.g., MySQL, PostgreSQL, Oracle).

### **2️⃣ What are the four types of JDBC drivers? Which one is best?**

🔹 **Type 1 – JDBC-ODBC Bridge Driver** (Deprecated, slow, not recommended)  
🔹 **Type 2 – Native-API Driver** (Uses database-specific native libraries, platform-dependent)  
🔹 **Type 3 – Network Protocol Driver** (Uses middleware server, rarely used)  
🔹 **Type 4 – Thin Driver (Pure Java Driver)** (Directly communicates with the database, platform-independent, ✅ **Best choice** ✅)

✅ **Type 4 Driver (com.mysql.cj.jdbc.Driver) is the best and most widely used.**

### **3️⃣ How do you establish a JDBC connection?**

✅ To establish a connection with a MySQL database:

| **import** java.sql.\*; **public** class JDBCExample {  **public** **static** **void** main(**String**[] args) {  **String** url = "jdbc:mysql://localhost:3306/mydatabase";  **String** user = "root";  **String** password = "password";  try (Connection con = DriverManager.getConnection(url, user, password)) {  System.out.println("Connected to the database!");  } catch (SQLException e) {  e.printStackTrace();  }  } } |
| --- |

✅ **DriverManager.getConnection(url, user, password)** is used to establish a connection.

### **4️⃣ What is the difference between Statement, PreparedStatement, and CallableStatement?**

| **Type** | **Usage** | **Security** | **Performance** |
| --- | --- | --- | --- |
| **Statement** | Executes static SQL queries | ❌ Vulnerable to SQL Injection | ❌ Slow |
| **PreparedStatement** | Executes parameterized queries | ✅ Prevents SQL Injection | ✅ Faster |
| **CallableStatement** | Calls stored procedures | ✅ Secure | ✅ Optimized for batch execution |

✅ **Always prefer PreparedStatement to avoid SQL Injection.**

### **5️⃣ How do you prevent SQL Injection in JDBC?**

✅ **Use PreparedStatement instead of Statement**

❌ **Bad Practice (Vulnerable to SQL Injection):**

| Statement stmt = con.createStatement(); String query = "SELECT \* FROM users WHERE name = '" + userInput + "'"; ResultSet rs = stmt.executeQuery(query); |
| --- |

✅ **Good Practice (Safe from SQL Injection):**

| PreparedStatement pstmt = con.prepareStatement("**SELECT** \* **FROM** **users** **WHERE** **name** = ?"); pstmt.setString(1, userInput); ResultSet rs = pstmt.executeQuery(); |
| --- |

✅ **Prevents attackers from injecting malicious SQL commands.**

**6️⃣ What is ResultSet, and how do you use it?**

✅ ResultSet is used to store and retrieve data from SQL queries.

**Example:** Fetching user data:

| **Statement** stmt = con.createStatement(); **ResultSet** rs = stmt.executeQuery("SELECT \* FROM users");  **while** (rs.next()) {  **System**.**out**.println(rs.getString("name") + " - " + rs.getString("email")); } |
| --- |

✅ rs.next() moves the cursor to the next row in the result set.

### **7️⃣ What is transaction management in JDBC? Explain with an example.**

✅ **Transaction Management** ensures that multiple queries execute **as a single unit**.  
✅ If one query fails, **rollback** ensures database consistency.

**Example: Bank Transaction (Transferring ₹500 from Account A to B)**

| **con**.setAutoCommit(false); // Disable auto-commit  **try** {  **stmt**.executeUpdate("UPDATE accounts SET balance = balance - 500 WHERE id = 1");  **stmt**.executeUpdate("UPDATE accounts SET balance = balance + 500 WHERE id = 2");  **con**.commit(); // Commit both queries } **catch** (SQLException e) {  **con**.rollback(); // Undo all changes if an error occurs } |
| --- |

✅ **Ensures either both queries succeed or none is executed.**

**8️⃣ How does Connection Pooling improve performance?**

✅ **Opening & closing database connections repeatedly is slow.**✅ **Connection Pooling** reuses connections instead of creating a new one every time.  
✅ Libraries like **HikariCP, Apache DBCP, and C3P0** manage connection pools.

**Example (Using HikariCP for Connection Pooling):**

| HikariConfig config = new HikariConfig(); config.setJdbcUrl("jdbc:mysql://localhost:3306/mydatabase"); config.setUsername("root"); config.setPassword("password"); HikariDataSource ds = new HikariDataSource(config); |
| --- |

✅ **Improves performance in large-scale applications.**

### 

### **9️⃣ How can we store JDBC configuration securely?**

🚫 **Never hardcode database credentials!**

✅ **Use a .properties file:**

| db.url=jdbc:mysql://localhost:3306/mydatabase db.user=root db.password=securepassword |
| --- |

✅ **Load from Java code:**

| db.url=jdbc:mysql://localhost:3306/mydatabase db.user=root db.password=securepassword |
| --- |

✅ **Use environment variables for extra security.**

### 

### 

### **🔟 What is the difference between JDBC and Hibernate?**

| **Feature** | **JDBC** | **Hibernate** |
| --- | --- | --- |
| **Query Execution** | SQL queries written manually | Uses HQL (Hibernate Query Language) |
| **Object Mapping** | Manual conversion between Java objects & database tables | Automatically maps Java objects to database tables |
| **Code Complexity** | High (Boilerplate code required) | Low (Less code, more abstraction) |
| **Performance** | Fast for simple queries | Optimized for large-scale applications |
| **Caching** | No built-in caching | Supports first-level & second-level caching |

✅ **Hibernate** simplifies database operations by reducing manual SQL coding.  
✅ **JDBC** is useful for direct SQL execution in smaller projects.

### **🎯 Quick Revision Table for Viva**

| **Question** | **Quick Answer** |
| --- | --- |
| **What is JDBC?** | Java API for database interaction |
| **Types of JDBC Drivers?** | Type 1 (Bridge), Type 2 (Native), Type 3 (Network), Type 4 (Thin - Best) |
| **How to connect to MySQL?** | DriverManager.getConnection(url, user, password) |
| **Statement vs. PreparedStatement vs. CallableStatement?** | PreparedStatement prevents SQL Injection |
| **Prevent SQL Injection?** | Use PreparedStatement |
| **What is ResultSet?** | Stores query results, use rs.next() to iterate |
| **What is Transaction Management?** | Ensures atomic execution (commit & rollback) |
| **Why use Connection Pooling?** | Improves performance by reusing connections |
| **How to store database credentials securely?** | Use .properties files or environment variables |
| **JDBC vs. Hibernate?** | JDBC requires SQL, Hibernate automates ORM |