

UNIT 13

Database Programming using JDBC

(2 Hours) — Object Oriented Programming in Java (CAC204)

Syllabus: Using Connection, Statement & ResultSet interfaces for manipulating data with the databases.

13.1 Introduction to JDBC

JDBC (Java Database Connectivity) is a standard Java API that lets a Java program connect to a relational database, send SQL statements, and process the results. It is defined in the **java.sql** package (and **javax.sql** for advanced features). JDBC is a set of *interfaces*; the actual implementation of these interfaces is supplied by the database vendor as a **JDBC driver**.

Because the application is written only against the JDBC interfaces, the same Java code can work with MySQL, Oracle, PostgreSQL or any other database by simply changing the driver and the connection URL.

13.1.1 Why JDBC?

- **Database independence** – one API for many databases.
- **Platform independence** – pure Java code runs anywhere a JVM and driver are available.
- **Standard SQL access** – execute SELECT, INSERT, UPDATE, DELETE and DDL statements from Java.
- **Object-oriented design** – Connection, Statement and ResultSet are all interfaces, so implementations can be swapped.

13.1.2 Important Classes and Interfaces of java.sql

Type	Kind	Purpose
DriverManager	Class	Manages drivers and creates Connection objects.
Connection	Interface	Represents an open session with a database.
Statement	Interface	Executes static SQL statements.
PreparedStatement	Interface	Executes precompiled, parameterized SQL statements.
CallableStatement	Interface	Calls stored procedures in the database.
ResultSet	Interface	Holds the rows returned by a SELECT query.

Type	Kind	Purpose
SQLException	Class	Reports any database access error.

13.2 JDBC Architecture

The application talks to the **JDBC API**. The API passes requests to the **DriverManager**, which selects a suitable **driver**. The driver converts the JDBC calls into the database's own protocol.

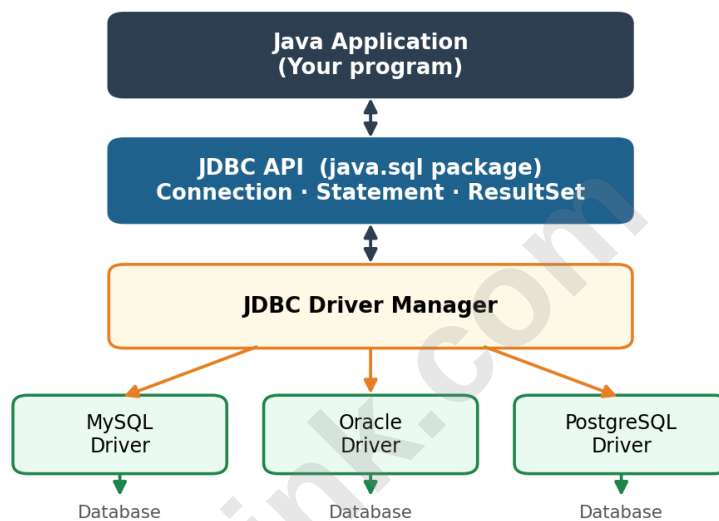


Figure 13.1: JDBC architecture — application, API, driver manager and drivers

13.2.1 Types of JDBC Drivers

Type	Name	Description
Type 1	JDBC-ODBC Bridge	Translates JDBC calls into ODBC calls. Slow and platform dependent; removed from Java 8 onwards.
Type 2	Native-API Driver	Uses the database's native client library. Needs native code installed on the client.
Type 3	Network-Protocol Driver	Sends calls to a middleware server, which talks to the database.
Type 4	Thin (Pure Java) Driver	Converts JDBC calls directly to the database protocol. Fast, portable and the most widely used (e.g. MySQL Connector/J).

Note: Type 4 is the recommended driver for most applications because it needs no extra software on the client.

13.3 Steps to Connect Java with a Database

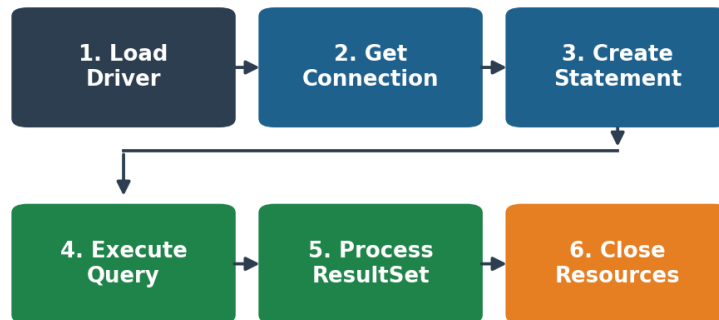


Figure 13.2: Six steps of a JDBC program

- **Step 1 – Load the driver.** `Class.forName("com.mysql.cj.jdbc.Driver");` (optional in JDBC 4.0+ if the driver JAR is on the classpath).
- **Step 2 – Establish the connection** using `DriverManager.getConnection()`.
- **Step 3 – Create a Statement** object from the connection.
- **Step 4 – Execute the SQL query** with `executeQuery()` or `executeUpdate()`.
- **Step 5 – Process the ResultSet** (for SELECT queries) row by row.
- **Step 6 – Close the resources** (ResultSet, Statement, Connection).

13.3.1 Connection URL Formats

Database	Driver class	URL format
MySQL	<code>com.mysql.cj.jdbc.Driver</code>	<code>jdbc:mysql://localhost:3306/dbname</code>
Oracle	<code>oracle.jdbc.OracleDriver</code>	<code>jdbc:oracle:thin:@localhost:1521:xe</code>
PostgreSQL	<code>org.postgresql.Driver</code>	<code>jdbc:postgresql://localhost:5432/dbname</code>
SQLite	<code>org.sqlite.JDBC</code>	<code>jdbc:sqlite:mydb.db</code>

13.4 The Connection Interface

A **Connection** object represents a session with a specific database. All SQL statements are executed and their results returned within the context of a connection. It is obtained from `DriverManager.getConnection(url, user, password)`.

Method	Description
Statement createStatement()	Creates a Statement object for sending SQL to the database.
PreparedStatement prepareStatement(String sql)	Creates a precompiled, parameterized statement.
CallableStatement prepareCall(String sql)	Creates an object for calling stored procedures.
void setAutoCommit(boolean b)	Turns automatic commit on or off (default is on).
void commit() / void rollback()	Makes changes permanent / undoes changes since last commit.
boolean isClosed()	Checks whether the connection is closed.
void close()	Closes the connection and releases resources.

Program 13.1: Establishing a connection

```
import java.sql.*;

public class ConnectDemo {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/college";
        String user = "root";
        String pass = "password";
        try (Connection con = DriverManager.getConnection(url, user, pass)) {
            System.out.println("Connected successfully!");
        } catch (SQLException e) {
            System.out.println("Connection failed: " + e.getMessage());
        }
    }
}
```

13.5 The Statement Interface

A **Statement** is used to execute a static SQL query and return its result. It is created by `con.createStatement()`. Three main execution methods are provided:

Method	Used for	Returns
executeQuery(String sql)	SELECT statements	ResultSet containing the rows
executeUpdate(String sql)	INSERT, UPDATE, DELETE, CREATE, DROP	int – number of rows affected (0 for DDL)
execute(String sql)	Any SQL (unknown type)	boolean – true if the result is a ResultSet

13.5.1 PreparedStatement

PreparedStatement extends **Statement**. The SQL is precompiled with **?** placeholders whose values are supplied by setter methods such as `setInt(1, value)` and `setString(2, value)`. It is faster when the same query runs repeatedly and, more importantly, it **prevents SQL injection**.

Statement vs PreparedStatement

```
// Statement - unsafe if 'name' comes from user input
stmt.executeQuery("SELECT * FROM student WHERE name='" + name + "'");

// PreparedStatement - safe and precompiled
PreparedStatement ps = con.prepareStatement("SELECT * FROM student WHERE name=?");
ps.setString(1, name);
ResultSet rs = ps.executeQuery();
```

Note: Always prefer PreparedStatement when the query contains values entered by the user.

13.6 The ResultSet Interface

A **ResultSet** holds the table of data produced by a **SELECT** query. It maintains a **cursor** that initially points *before the first row*. Each call to `next()` moves the cursor forward one row and returns **false** when no rows remain.

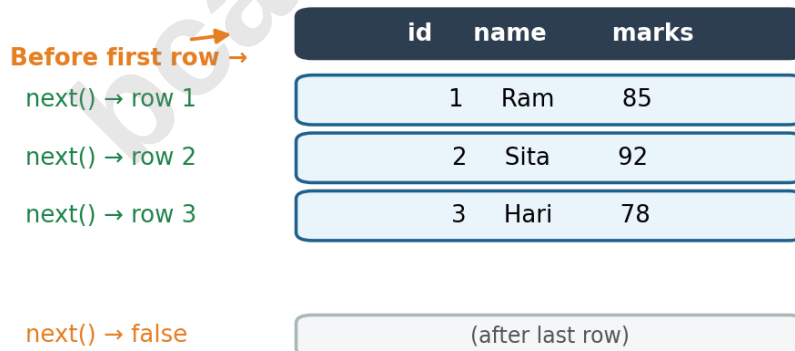


Figure 13.3: ResultSet cursor movement using `next()`

Method	Description
<code>boolean next()</code>	Moves cursor to the next row; false if there are no more rows.
<code>int getInt(int col / String name)</code>	Reads an INT column of the current row.
<code>String getString(int col / String name)</code>	Reads a VARCHAR/CHAR column as a String.
<code>double getDouble(...)</code> / <code>Date getDate(...)</code>	Reads DOUBLE / DATE columns.

Method	Description
boolean previous(), first(), last()	Cursor movement (only for scrollable ResultSets).
void close()	Releases the ResultSet resources.

Note: Column indexes in JDBC start from 1, not 0.

13.6.1 ResultSet Types

Constant	Meaning
TYPE_FORWARD_ONLY	Default. Cursor moves forward only.
TYPE_SCROLL_INSENSITIVE	Cursor can move forward and backward; does not reflect changes made by others.
TYPE_SCROLL_SENSITIVE	Scrollable and reflects changes made to the data by others.
CONCUR_READ_ONLY / CONCUR_UPDATABLE	Whether the ResultSet can be used to update the database.

13.7 Manipulating Data: Complete CRUD Example

Consider a table **student(id INT PRIMARY KEY, name VARCHAR(50), marks INT)** in the database **college**. The following examples perform Create, Read, Update and Delete operations.

Example 13.1: Insert and display records

Write a JDBC program to insert a student (1, "Ram", 85) into the table and then display all students.

```
import java.sql.*;

public class InsertSelect {
    public static void main(String[] args) throws Exception {
        Class.forName("com.mysql.cj.jdbc.Driver");
        Connection con = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/college", "root", "password");
        Statement st = con.createStatement();

        // INSERT
        int n = st.executeUpdate("INSERT INTO student VALUES (1, 'Ram', 85)");
        System.out.println(n + " row inserted");

        // SELECT
        ResultSet rs = st.executeQuery("SELECT * FROM student");
        while (rs.next()) {
            System.out.println(rs.getInt("id") + " "
                + rs.getString("name") + " " + rs.getInt("marks"));
        }
    }
}
```

```

    }
    rs.close(); st.close(); con.close();
}
}

```

Output

1 row inserted

1 Ram 85

Example 13.2: Update and delete using PreparedStatement

Update the marks of the student with id 1 to 90, then delete the student with id 1.

```

String upd = "UPDATE student SET marks = ? WHERE id = ?";
PreparedStatement ps = con.prepareStatement(upd);
ps.setInt(1, 90);           // first ? -> marks
ps.setInt(2, 1);           // second ? -> id
System.out.println(ps.executeUpdate() + " row updated");

String del = "DELETE FROM student WHERE id = ?";
ps = con.prepareStatement(del);
ps.setInt(1, 1);
System.out.println(ps.executeUpdate() + " row deleted");
ps.close();

```

Output

1 row updated

1 row deleted

13.8 Exception Handling and Closing Resources

Almost every JDBC method throws the checked exception **SQLException**, so JDBC code must be placed in a try-catch block or the method must declare *throws SQLException*. Useful methods of **SQLException** are `getMessage()`, `getSQLState()` and `getErrorCode()`.

Connections, statements and result sets consume database resources, so they must always be closed. Since Java 7, the **try-with-resources** statement closes them automatically, even when an exception occurs.

Program 13.2: try-with-resources

```

String sql = "SELECT id, name, marks FROM student WHERE marks > ?";
try (Connection con = DriverManager.getConnection(url, user, pass);
    PreparedStatement ps = con.prepareStatement(sql)) {

```

```

ps.setInt(1, 80);
try (ResultSet rs = ps.executeQuery()) {
    while (rs.next()) {
        System.out.println(rs.getInt(1) + " "
            + rs.getString(2) + " " + rs.getInt(3));
    }
}
} catch (SQLException e) {
    System.out.println("SQL State: " + e.getSQLState());
    System.out.println("Error      : " + e.getMessage());
}

```

13.8.1 Transaction Management (Brief)

By default a connection is in **auto-commit** mode: each SQL statement is committed immediately. To group several statements into one unit of work, call `con.setAutoCommit(false)`, then `commit()` on success or `rollback()` on failure.

```

con.setAutoCommit(false);
try {
    st.executeUpdate("UPDATE account SET bal = bal - 500 WHERE id = 1");
    st.executeUpdate("UPDATE account SET bal = bal + 500 WHERE id = 2");
    con.commit();           // both succeed
} catch (SQLException e) {
    con.rollback();         // undo both on error
}

```

13.9 Statement vs PreparedStatement

Feature	Statement	PreparedStatement
SQL type	Static SQL string	Precompiled SQL with ? parameters
Performance	Compiled every time	Compiled once, reused
SQL injection	Vulnerable	Protected
Parameters	Concatenated into the string	Set using setXxx() methods
Best used for	Simple, fixed queries	Repeated or user-input queries

13.10 Important Exam Questions

Practice Questions

1. What is JDBC? Explain its architecture with a neat diagram.
2. Explain the different types of JDBC drivers.
3. List and explain the steps required to connect a Java program to a database.
4. Differentiate between Statement and PreparedStatement.
5. Explain the role of Connection, Statement and ResultSet interfaces with examples.
6. Write a JDBC program to insert, update, delete and display records of a table.

Key Takeaways

- ✓ JDBC is a Java API (java.sql) for connecting to relational databases using SQL.
- ✓ A driver (preferably Type 4) implements the JDBC interfaces for a specific database.
- ✓ Six steps: load driver → connect → create statement → execute → process result → close.
- ✓ Connection = session; Statement = sends SQL; ResultSet = holds rows returned by SELECT.
- ✓ executeQuery() is for SELECT; executeUpdate() is for INSERT/UPDATE/DELETE/DDL.
- ✓ ResultSet cursor starts before the first row; call next() to move; column indexes start at 1.
- ✓ Use PreparedStatement to prevent SQL injection and improve performance.
- ✓ Always close resources — use try-with-resources — and handle SQLException.

— End of Unit 13 —